



# Application Fundamentals

## Quickview

- Android applications are composed of one or more application components (activities, services, content providers, and broadcast receivers)
- Each component performs a different role in the overall application behavior, and each one can be activated individually (even by other applications)
- The manifest file must declare all components in the application and should also declare all application requirements, such as the minimum version of Android required and any hardware configurations required
- Non-code application resources (images, strings, layout files, etc.) should include alternatives for different device configurations (such as different strings for different languages and different layouts for different screen sizes)

## In this document

1. [Application Components](#)
  1. [Activating components](#)
2. [The Manifest File](#)
  1. [Declaring components](#)
  2. [Declaring application requirements](#)
3. [Application Resources](#)

Android applications are written in the Java programming language. The Android SDK tools compile the code—along with any data and resource files—into an *Android package*, an archive file with an .apk suffix. All the code in a single .apk file is considered to be one application and is the file that Android-powered devices use to install the application.

Once installed on a device, each Android application lives in its own security sandbox:

- The Android operating system is a multi-user Linux system in which each application is a different user.
- By default, the system assigns each application a unique Linux user ID (the ID is used only by the system and is unknown to the application). The system sets permissions for all the files in an application so that only the user ID assigned to that application can access them.
- Each process has its own virtual machine (VM), so an application's code runs in isolation from other applications.
- By default, every application runs in its own Linux process. Android starts the process when any of the application's components need to be executed, then shuts down the process when it's no longer needed or when the system must recover memory for other applications.

In this way, the Android system implements the *principle of least privilege*. That is, each application, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an application cannot access parts of the system for which it is not given permission.

However, there are ways for an application to share data with other applications and for an application to access system services:

- It's possible to arrange for two applications to share the same Linux user ID, in which case they are able to access each other's files. To conserve system resources, applications with the same user ID can

also arrange to run in the same Linux process and share the same VM (the applications must also be signed with the same certificate).

- An application can request permission to access device data such as the user's contacts, SMS messages, the mountable storage (SD card), camera, Bluetooth, and more. All application permissions must be granted by the user at install time.

That covers the basics regarding how an Android application exists within the system. The rest of this document introduces you to:

- The core framework components that define your application.
- The manifest file in which you declare components and required device features for your application.
- Resources that are separate from the application code and allow your application to gracefully optimize its behavior for a variety of device configurations.

## Application Components

Application components are the essential building blocks of an Android application. Each component is a different point through which the system can enter your application. Not all components are actual entry points for the user and some depend on each other, but each one exists as its own entity and plays a specific role—each one is a unique building block that helps define your application's overall behavior.

There are four different types of application components. Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed.

Here are the four types of application components:

### Activities

An *activity* represents a single screen with a user interface. For example, an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email application, each one is independent of the others. As such, a different application can start any one of these activities (if the email application allows it). For example, a camera application can start the activity in the email application that composes new mail, in order for the user to share a picture.

An activity is implemented as a subclass of [Activity](#) and you can learn more about it in the [Activities](#) developer guide.

### Services

A *service* is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. 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. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it.

A service is implemented as a subclass of [Service](#) and you can learn more about it in the [Services](#) developer guide.

### Content providers

A *content provider* manages a shared set of application data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your application can access. Through the content provider, other applications can query or even modify the data (if the content provider allows it). For example, the Android system provides a content provider that manages the user's contact information. As such, any application with the proper permissions can query part of the content provider (such as [ContactsContract.Data](#)) to read and write information about a particular person.

Content providers are also useful for reading and writing data that is private to your application and not shared. For example, the [Note Pad](#) sample application uses a content provider to save notes.

A content provider is implemented as a subclass of [ContentProvider](#) and must implement a standard set of APIs that enable other applications to perform transactions. For more information, see the [Content Providers](#) developer guide.

## Broadcast receivers

A *broadcast receiver* is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Applications can also initiate broadcasts—for example, to let other applications know that some data has been downloaded to the device and is available for them to use. Although broadcast receivers don't display a user interface, they may [create a status bar notification](#) to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event.

A broadcast receiver is implemented as a subclass of [BroadcastReceiver](#) and each broadcast is delivered as an [Intent](#) object. For more information, see the [BroadcastReceiver](#) class.

A unique aspect of the Android system design is that any application can start another application's component. For example, if you want the user to capture a photo with the device camera, there's probably another application that does that and your application can use it, instead of developing an activity to capture a photo yourself. You don't need to incorporate or even link to the code from the camera application. Instead, you can simply start the activity in the camera application that captures a photo. When complete, the photo is even returned to your application so you can use it. To the user, it seems as if the camera is actually a part of your application.

When the system starts a component, it starts the process for that application (if it's not already running) and instantiates the classes needed for the component. For example, if your application starts the activity in the camera application that captures a photo, that activity runs in the process that belongs to the camera application, not in your application's process. Therefore, unlike applications on most other systems, Android applications don't have a single entry point (there's no `main()` function, for example).

Because the system runs each application in a separate process with file permissions that restrict access to other applications, your application cannot directly activate a component from another application. The Android system, however, can. So, to activate a component in another application, you must deliver a message to the system that specifies your *intent* to start a particular component. The system then activates the component for you.

## Activating Components

Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an *intent*. Intents bind individual components to each other at runtime (you can think of them as the messengers that request an action from other components), whether the component belongs to your application or another.

An intent is created with an [Intent](#) object, which defines a message to activate either a specific component or a specific *type* of component—an intent can be either explicit or implicit, respectively.

For activities and services, an intent defines the action to perform (for example, to "view" or "send" something) and may specify the URI of the data to act on (among other things that the component being started might need to know). For example, an intent might convey a request for an activity to show an image or to open a web page. In some cases, you can start an activity to receive a result, in which case, the activity also returns the result in an [Intent](#) (for example, you can issue an intent to let the user pick a personal contact and have it returned to you—the return intent includes a URI pointing to the chosen contact).

For broadcast receivers, the intent simply defines the announcement being broadcast (for example, a broadcast to indicate the device battery is low includes only a known action string that indicates "battery is low").

The other component type, content provider, is not activated by intents. Rather, it is activated when targeted by a request from a [ContentResolver](#). The content resolver handles all direct transactions with the content provider so that the component that's performing transactions with the provider doesn't need to and instead calls methods on the [ContentResolver](#) object. This leaves a layer of abstraction between the content provider and the component requesting information (for security).

There are separate methods for activating each type of component:

- You can start an activity (or give it something new to do) by passing an [Intent](#) to [startActivity\(\)](#) or [startActivityForResult\(\)](#) (when you want the activity to return a result).
- You can start a service (or give new instructions to an ongoing service) by passing an [Intent](#) to [startService\(\)](#). Or you can bind to the service by passing an [Intent](#) to [bindService\(\)](#).
- You can initiate a broadcast by passing an [Intent](#) to methods like [sendBroadcast\(\)](#), [sendOrderedBroadcast\(\)](#), or [sendStickyBroadcast\(\)](#).
- You can perform a query to a content provider by calling [query\(\)](#) on a [ContentResolver](#).

For more information about using intents, see the [Intents and Intent Filters](#) document. More information about activating specific components is also provided in the following documents: [Activities](#), [Services](#), [BroadcastReceiver](#) and [Content Providers](#).

## The Manifest File

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

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

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

## Declaring components

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

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

```
</application>  
</manifest>
```

In the [`<application>`](#) element, the `android:icon` attribute points to resources for an icon that identifies the application.

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

You must declare all application components this way:

- [`<activity>`](#) elements for activities
- [`<service>`](#) elements for services
- [`<receiver>`](#) elements for broadcast receivers
- [`<provider>`](#) elements for content providers

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

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

## Declaring component capabilities

As discussed above, in [Activating Components](#), you can use an [Intent](#) to start activities, services, and broadcast receivers. You can do so by explicitly naming the target component (using the component class name) in the intent. However, the real power of intents lies in the concept of intent actions. With intent actions, you simply describe the type of action you want to perform (and optionally, the data upon which you'd like to perform the action) and allow the system to find a component on the device that can perform the action and start it. If there are multiple components that can perform the action described by the intent, then the user selects which one to use.

The way the system identifies the components that can respond to an intent is by comparing the intent received to the *intent filters* provided in the manifest file of other applications on the device.

When you declare a component in your application's manifest, you can optionally include intent filters that declare the capabilities of the component so it can respond to intents from other applications. You can declare an intent filter for your component by adding an [`<intent-filter>`](#) element as a child of the component's declaration element.

For example, an email application with an activity for composing a new email might declare an intent filter in its manifest entry to respond to "send" intents (in order to send email). An activity in your application can then create an intent with the "send" action ([ACTION\\_SEND](#)), which the system matches to the email application's "send" activity and launches it when you invoke the intent with [`startActivity\(\)`](#).

For more about creating intent filters, see the [Intents and Intent Filters](#) document.

## Declaring application requirements

There are a variety of devices powered by Android and not all of them provide the same features and capabilities. In order to prevent your application from being installed on devices that lack features needed by your ap-

plication, it's important that you clearly define a profile for the types of devices your application supports by declaring device and software requirements in your manifest file. Most of these declarations are informational only and the system does not read them, but external services such as Google Play do read them in order to provide filtering for users when they search for applications from their device.

For example, if your application requires a camera and uses APIs introduced in Android 2.1 ([API Level 7](#)), you should declare these as requirements in your manifest file. That way, devices that do *not* have a camera and have an Android version *lower* than 2.1 cannot install your application from Google Play.

However, you can also declare that your application uses the camera, but does not *require* it. In that case, your application must perform a check at runtime to determine if the device has a camera and disable any features that use the camera if one is not available.

Here are some of the important device characteristics that you should consider as you design and develop your application:

## Screen size and density

In order to categorize devices by their screen type, Android defines two characteristics for each device: screen size (the physical dimensions of the screen) and screen density (the physical density of the pixels on the screen, or dpi—dots per inch). To simplify all the different types of screen configurations, the Android system generalizes them into select groups that make them easier to target.

The screen sizes are: small, normal, large, and extra large.

The screen densities are: low density, medium density, high density, and extra high density.

By default, your application is compatible with all screen sizes and densities, because the Android system makes the appropriate adjustments to your UI layout and image resources. However, you should create specialized layouts for certain screen sizes and provide specialized images for certain densities, using alternative layout resources, and by declaring in your manifest exactly which screen sizes your application supports with the [`<supports-screens>`](#) element.

For more information, see the [Supporting Multiple Screens](#) document.

## Input configurations

Many devices provide a different type of user input mechanism, such as a hardware keyboard, a trackball, or a five-way navigation pad. If your application requires a particular kind of input hardware, then you should declare it in your manifest with the [`<uses-configuration>`](#) element. However, it is rare that an application should require a certain input configuration.

## Device features

There are many hardware and software features that may or may not exist on a given Android-powered device, such as a camera, a light sensor, bluetooth, a certain version of OpenGL, or the fidelity of the touch-screen. You should never assume that a certain feature is available on all Android-powered devices (other than the availability of the standard Android library), so you should declare any features used by your application with the [`<uses-feature>`](#) element.

## Platform Version

Different Android-powered devices often run different versions of the Android platform, such as Android 1.6 or Android 2.3. Each successive version often includes additional APIs not available in the previous version. In order to indicate which set of APIs are available, each platform version specifies an [API Level](#) (for example, Android 1.0 is API Level 1 and Android 2.3 is API Level 9). If you use any APIs that were added to the platform after version 1.0, you should declare the minimum API Level in which those APIs were introduced using the [`<uses-sdk>`](#) element.

It's important that you declare all such requirements for your application, because, when you distribute your application on Google Play, the store uses these declarations to filter which applications are available on each device. As such, your application should be available only to devices that meet all your application requirements.

For more information about how Google Play filters applications based on these (and other) requirements, see the [Filters on Google Play](#) document.

## Application Resources

An Android application is composed of more than just code—it requires resources that are separate from the source code, such as images, audio files, and anything relating to the visual presentation of the application. For example, you should define animations, menus, styles, colors, and the layout of activity user interfaces with XML files. Using application resources makes it easy to update various characteristics of your application without modifying code and—by providing sets of alternative resources—enables you to optimize your application for a variety of device configurations (such as different languages and screen sizes).

For every resource that you include in your Android project, the SDK build tools define a unique integer ID, which you can use to reference the resource from your application code or from other resources defined in XML. For example, if your application contains an image file named `logo.png` (saved in the `res/drawable/` directory), the SDK tools generate a resource ID named `R.drawable.logo`, which you can use to reference the image and insert it in your user interface.

One of the most important aspects of providing resources separate from your source code is the ability for you to provide alternative resources for different device configurations. For example, by defining UI strings in XML, you can translate the strings into other languages and save those strings in separate files. Then, based on a language *qualifier* that you append to the resource directory's name (such as `res/values-fr/` for French string values) and the user's language setting, the Android system applies the appropriate language strings to your UI.

Android supports many different *qualifiers* for your alternative resources. The qualifier is a short string that you include in the name of your resource directories in order to define the device configuration for which those resources should be used. As another example, you should often create different layouts for your activities, depending on the device's screen orientation and size. For example, when the device screen is in portrait orientation (tall), you might want a layout with buttons to be vertical, but when the screen is in landscape orientation (wide), the buttons should be aligned horizontally. To change the layout depending on the orientation, you can define two different layouts and apply the appropriate qualifier to each layout's directory name. Then, the system automatically applies the appropriate layout depending on the current device orientation.

For more about the different kinds of resources you can include in your application and how to create alternative resources for various device configurations, see the [Application Resources](#) developer guide.

# Activities

## Quickview

- An activity provides a user interface for a single screen in your application
- Activities can move into the background and then be resumed with their state restored

## In this document

1. [Creating an Activity](#)
  1. [Implementing a user interface](#)
  2. [Declaring the activity in the manifest](#)
2. [Starting an Activity](#)
  1. [Starting an activity for a result](#)
3. [Shutting Down an Activity](#)
4. [Managing the Activity Lifecycle](#)
  1. [Implementing the lifecycle callbacks](#)
  2. [Saving activity state](#)
  3. [Handling configuration changes](#)
  4. [Coordinating activities](#)

## Key classes

1. [Activity](#)

## See also

1. [Tasks and Back Stack](#)

An [Activity](#) is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. The back stack abides to the basic "last in, first out" stack mechanism, so, when the user is done with the current activity and presses the *Back* button, it is popped from the stack (and destroyed) and the previous activity resumes. (The back stack is discussed more in the [Tasks and Back Stack](#) document.)

When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides you the opportunity to perform specific work that's appropriate to that state change. For instance, when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted. These state transitions are all part of the activity lifecycle.

The rest of this document discusses the basics of how to build and use an activity, including a complete discussion of how the activity lifecycle works, so you can properly manage the transition between various activity states.

## Creating an Activity

To create an activity, you must create a subclass of [Activity](#) (or an existing subclass of it). In your subclass, you need to implement callback methods that the system calls when the activity transitions between various states of its lifecycle, such as when the activity is being created, stopped, resumed, or destroyed. The two most important callback methods are:

### [onCreate\(\)](#)

You must implement this method. The system calls this when creating your activity. Within your implementation, you should initialize the essential components of your activity. Most importantly, this is where you must call [setContentView\(\)](#) to define the layout for the activity's user interface.

### [onPause\(\)](#)

The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

There are several other lifecycle callback methods that you should use in order to provide a fluid user experience between activities and handle unexpected interruptions that cause your activity to be stopped and even destroyed. All of the lifecycle callback methods are discussed later, in the section about [Managing the Activity Lifecycle](#).

## Implementing a user interface

The user interface for an activity is provided by a hierarchy of views—objects derived from the [View](#) class. Each view controls a particular rectangular space within the activity's window and can respond to user interaction. For example, a view might be a button that initiates an action when the user touches it.

Android provides a number of ready-made views that you can use to design and organize your layout. "Widgets" are views that provide a visual (and interactive) elements for the screen, such as a button, text field, checkbox, or just an image. "Layouts" are views derived from [ViewGroup](#) that provide a unique layout model for its child views, such as a linear layout, a grid layout, or relative layout. You can also subclass the [View](#) and [ViewGroup](#) classes (or existing subclasses) to create your own widgets and layouts and apply them to your activity layout.

The most common way to define a layout using views is with an XML layout file saved in your application resources. This way, you can maintain the design of your user interface separately from the source code that defines the activity's behavior. You can set the layout as the UI for your activity with [setContentView\(\)](#), passing the resource ID for the layout. However, you can also create new [Views](#) in your activity code and build a view hierarchy by inserting new [Views](#) into a [ViewGroup](#), then use that layout by passing the root [ViewGroup](#) to [setContentView\(\)](#).

For information about creating a user interface, see the [User Interface](#) documentation.

## Declaring the activity in the manifest

You must declare your activity in the manifest file in order for it to be accessible to the system. To declare your activity, open your manifest file and add an [<activity>](#) element as a child of the [<application>](#) element. For example:

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
...
</manifest >
```

There are several other attributes that you can include in this element, to define properties such as the label for the activity, an icon for the activity, or a theme to style the activity's UI. The [android:name](#) attribute is the only required attribute—it specifies the class name of the activity. Once you publish your application, you should not change this name, because if you do, you might break some functionality, such as application shortcuts (read the blog post, [Things That Cannot Change](#)).

See the [<activity>](#) element reference for more information about declaring your activity in the manifest.

## Using intent filters

An [<activity>](#) element can also specify various intent filters—using the [<intent-filter>](#) element—in order to declare how other application components may activate it.

When you create a new application using the Android SDK tools, the stub activity that's created for you automatically includes an intent filter that declares the activity responds to the "main" action and should be placed in the "launcher" category. The intent filter looks like this:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

The [<action>](#) element specifies that this is the "main" entry point to the application. The [<category>](#) element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).

If you intend for your application to be self-contained and not allow other applications to activate its activities, then you don't need any other intent filters. Only one activity should have the "main" action and "launcher" category, as in the previous example. Activities that you don't want to make available to other applications should have no intent filters and you can start them yourself using explicit intents (as discussed in the following section).

However, if you want your activity to respond to implicit intents that are delivered from other applications (and your own), then you must define additional intent filters for your activity. For each type of intent to which you want to respond, you must include an [<intent-filter>](#) that includes an [<action>](#) element and, optionally, a [<category>](#) element and/or a [<data>](#) element. These elements specify the type of intent to which your activity can respond.

For more information about how your activities can respond to intents, see the [Intents and Intent Filters](#) document.

# Starting an Activity

You can start another activity by calling `startActivity()`, passing it an `Intent` that describes the activity you want to start. The intent specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application). An intent can also carry small amounts of data to be used by the activity that is started.

When working within your own application, you'll often need to simply launch a known activity. You can do so by creating an intent that explicitly defines the activity you want to start, using the class name. For example, here's how one activity starts another activity named `SignInActivity`:

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

However, your application might also want to perform some action, such as send an email, text message, or status update, using data from your activity. In this case, your application might not have its own activities to perform such actions, so you can instead leverage the activities provided by other applications on the device, which can perform the actions for you. This is where intents are really valuable—you can create an intent that describes an action you want to perform and the system launches the appropriate activity from another application. If there are multiple activities that can handle the intent, then the user can select which one to use. For example, if you want to allow the user to send an email message, you can create the following intent:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

The `EXTRA_EMAIL` extra added to the intent is a string array of email addresses to which the email should be sent. When an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form. In this situation, the email application's activity starts and when the user is done, your activity resumes.

## Starting an activity for a result

Sometimes, you might want to receive a result from the activity that you start. In that case, start the activity by calling `startActivityForResult()` (instead of `startActivity()`). To then receive the result from the subsequent activity, implement the `onActivityResult()` callback method. When the subsequent activity is done, it returns a result in an `Intent` to your `onActivityResult()` method.

For example, perhaps you want the user to pick one of their contacts, so your activity can do something with the information in that contact. Here's how you can create such an intent and handle the result:

```
private void pickContact() {
    // Create an intent to "pick" a contact, as defined by the content provider
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode == PICK_CONTACT_REQUEST)
        // Perform a query to the contact's content provider for the contact's
        Cursor cursor = getContentResolver().query(data.getData(),
```

```

        new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not empty
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Do something with the selected contact's name...
        }
    }
}

```

This example shows the basic logic you should use in your [onActivityResult\(\)](#) method in order to handle an activity result. The first condition checks whether the request was successful—if it was, then the `resultCode` will be [RESULT\\_OK](#)—and whether the request to which this result is responding is known—in this case, the `requestCode` matches the second parameter sent with [startActivityForResult\(\)](#). From there, the code handles the activity result by querying the data returned in an [Intent](#) (the `data` parameter).

What happens is, a [ContentResolver](#) performs a query against a content provider, which returns a [Cursor](#) that allows the queried data to be read. For more information, see the [Content Providers](#) document.

For more information about using intents, see the [Intents and Intent Filters](#) document.

## Shutting Down an Activity

You can shut down an activity by calling its [finish\(\)](#) method. You can also shut down a separate activity that you previously started by calling [finishActivity\(\)](#).

**Note:** In most cases, you should not explicitly finish an activity using these methods. As discussed in the following section about the activity lifecycle, the Android system manages the life of an activity for you, so you do not need to finish your own activities. Calling these methods could adversely affect the expected user experience and should only be used when you absolutely do not want the user to return to this instance of the activity.

## Managing the Activity Lifecycle

Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application. The lifecycle of an activity is directly affected by its association with other activities, its task and back stack.

An activity can exist in essentially three states:

### **Resumed**

The activity is in the foreground of the screen and has user focus. (This state is also sometimes referred to as "running".)

### **Paused**

Another activity is in the foreground and has focus, but this one is still visible. That is, another activity is visible on top of this one and that activity is partially transparent or doesn't cover the entire screen. A paused activity is completely alive (the [Activity](#) object is retained in memory, it maintains all state and member information, and remains attached to the window manager), but can be killed by the system in extremely low memory situations.

### **Stopped**

The activity is completely obscured by another activity (the activity is now in the "background"). A stopped activity is also still alive (the [Activity](#) object is retained in memory, it maintains all state and

member information, but is *not* attached to the window manager). However, it is no longer visible to the user and it can be killed by the system when memory is needed elsewhere.

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its [finish\(\)](#) method), or simply killing its process. When the activity is opened again (after being finished or killed), it must be created all over.

## Implementing the lifecycle callbacks

When an activity transitions into and out of the different states described above, it is notified through various callback methods. All of the callback methods are hooks that you can override to do appropriate work when the state of your activity changes. The following skeleton activity includes each of the fundamental lifecycle methods:

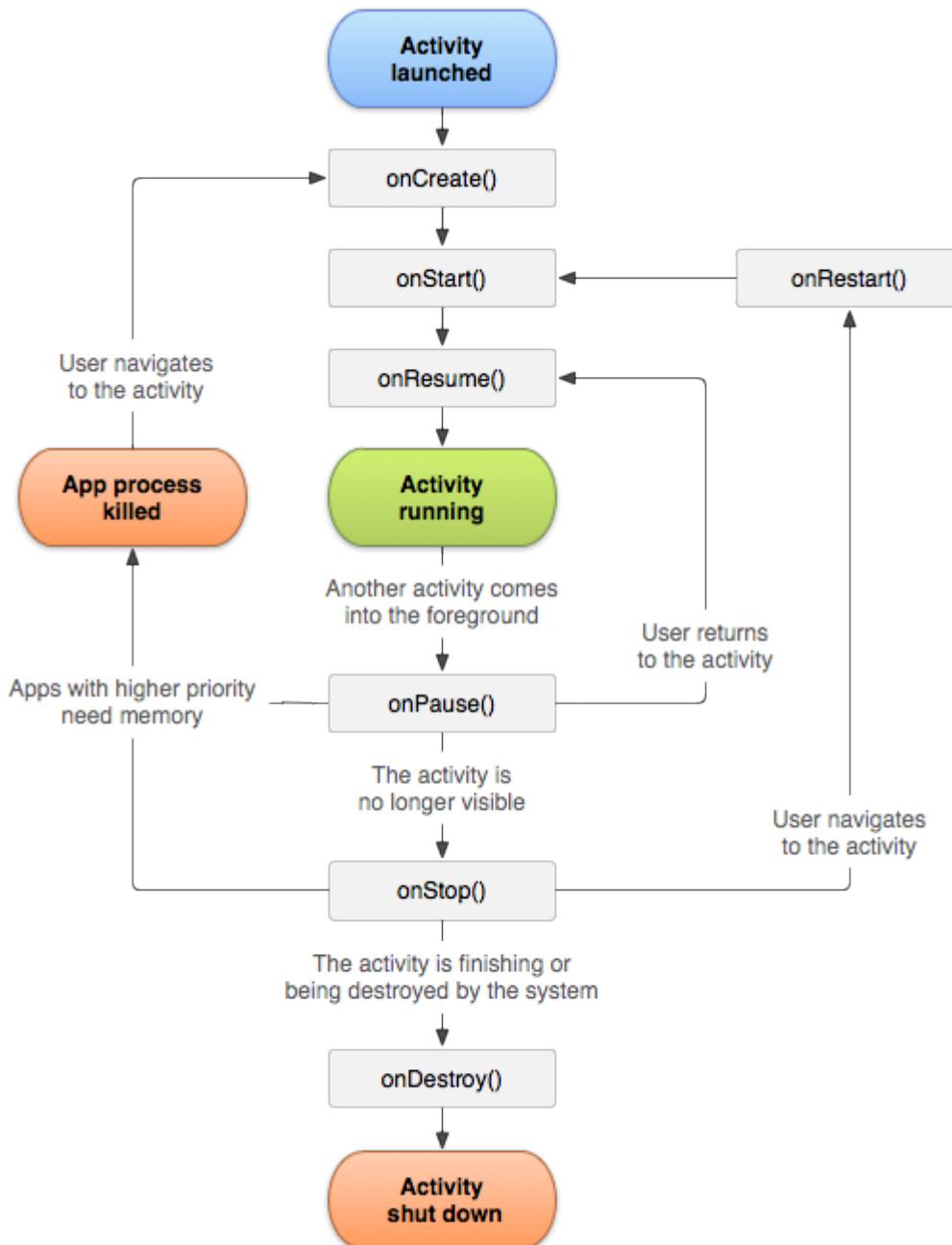
```
public class ExampleActivity extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        // The activity is being created.  
    }  
    @Override  
    protected void onStart\(\) {  
        super.onStart();  
        // The activity is about to become visible.  
    }  
    @Override  
    protected void onResume\(\) {  
        super.onResume();  
        // The activity has become visible (it is now "resumed").  
    }  
    @Override  
    protected void onPause\(\) {  
        super.onPause();  
        // Another activity is taking focus (this activity is about to be "paused").  
    }  
    @Override  
    protected void onStop\(\) {  
        super.onStop();  
        // The activity is no longer visible (it is now "stopped").  
    }  
    @Override  
    protected void onDestroy\(\) {  
        super.onDestroy();  
        // The activity is about to be destroyed.  
    }  
}
```

**Note:** Your implementation of these lifecycle methods must always call the superclass implementation before doing any work, as shown in the examples above.

Taken together, these methods define the entire lifecycle of an activity. By implementing these methods, you can monitor three nested loops in the activity lifecycle:

- The **entire lifetime** of an activity happens between the call to `onCreate()` and the call to `onDestroy()`. Your activity should perform setup of "global" state (such as defining layout) in `onCreate()`, and release all remaining resources in `onDestroy()`. For example, if your activity has a thread running in the background to download data from the network, it might create that thread in `onCreate()` and then stop the thread in `onDestroy()`.
- The **visible lifetime** of an activity happens between the call to `onStart()` and the call to `onStop()`. During this time, the user can see the activity on-screen and interact with it. For example, `onStop()` is called when a new activity starts and this one is no longer visible. Between these two methods, you can maintain resources that are needed to show the activity to the user. For example, you can register a `BroadcastReceiver` in `onStart()` to monitor changes that impact your UI, and unregister it in `onStop()` when the user can no longer see what you are displaying. The system might call `onStart()` and `onStop()` multiple times during the entire lifetime of the activity, as the activity alternates between being visible and hidden to the user.
- The **foreground lifetime** of an activity happens between the call to `onResume()` and the call to `onPause()`. During this time, the activity is in front of all other activities on screen and has user input focus. An activity can frequently transition in and out of the foreground—for example, `onPause()` is called when the device goes to sleep or when a dialog appears. Because this state can transition often, the code in these two methods should be fairly lightweight in order to avoid slow transitions that make the user wait.

Figure 1 illustrates these loops and the paths an activity might take between states. The rectangles represent the callback methods you can implement to perform operations when the activity transitions between states.



**Figure 1.** The activity lifecycle.

The same lifecycle callback methods are listed in table 1, which describes each of the callback methods in more detail and locates each one within the activity's overall lifecycle, including whether the system can kill the activity after the callback method completes.

**Table 1.** A summary of the activity lifecycle's callback methods.

Method	Description	Killable after?	Next
<a href="#">onCreate()</a>	Called when the activity is first created. This is where you should do all of your normal static set up — create views, bind data to lists, and so on. This method is passed a Bundle object containing the activity's previous state, if that state was captured (see <a href="#">Saving Activity State</a> , later).	No	<a href="#">onStart()</a>

Method	Description	Killable after?	Next
	Always followed by <code>onStart()</code> .		
<code>onRestart()</code>	Called after the activity has been stopped, just prior to it being started again.  Always followed by <code>onStart()</code>	No	<code>onStart()</code>
<code>onStart()</code>	Called just before the activity becomes visible to the user.  Followed by <code>onResume()</code> if the activity comes to the foreground, or <code>onStop()</code> if it becomes hidden.	No	<code>onResume()</code> or <code>onStop()</code>
<code>onResume()</code>	Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it.  Always followed by <code>onPause()</code> .	No	<code>onPause()</code>
<code>onPause()</code>	Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.  Followed either by <code>onResume()</code> if the activity returns back to the front, or by <code>onStop()</code> if it becomes invisible to the user.	Yes	<code>onResume()</code> or <code>onStop()</code>
<code>onStop()</code>	Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it.  Followed either by <code>onRestart()</code> if the activity is coming back to interact with the user, or by <code>onDestroy()</code> if this activity is going away.	Yes	<code>onRestart()</code> or <code>onDestroy()</code>
<code>onDestroy()</code>	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing (someone called <code>finish()</code> on	Yes	<i>nothing</i>

Method	Description	Killable after?	Next
	it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the <a href="#">isFinishing()</a> method.		

The column labeled "Killable after?" indicates whether or not the system can kill the process hosting the activity at any time *after the method returns*, without executing another line of the activity's code. Three methods are marked "yes": ([onPause\(\)](#), [onStop\(\)](#), and [onDestroy\(\)](#)). Because [onPause\(\)](#) is the first of the three, once the activity is created, [onPause\(\)](#) is the last method that's guaranteed to be called before the process *can* be killed—if the system must recover memory in an emergency, then [onStop\(\)](#) and [onDestroy\(\)](#) might not be called. Therefore, you should use [onPause\(\)](#) to write crucial persistent data (such as user edits) to storage. However, you should be selective about what information must be retained during [onPause\(\)](#), because any blocking procedures in this method block the transition to the next activity and slow the user experience.

Methods that are marked "No" in the **Killable** column protect the process hosting the activity from being killed from the moment they are called. Thus, an activity is killable from the time [onPause\(\)](#) returns to the time [onResume\(\)](#) is called. It will not again be killable until [onPause\(\)](#) is again called and returns.

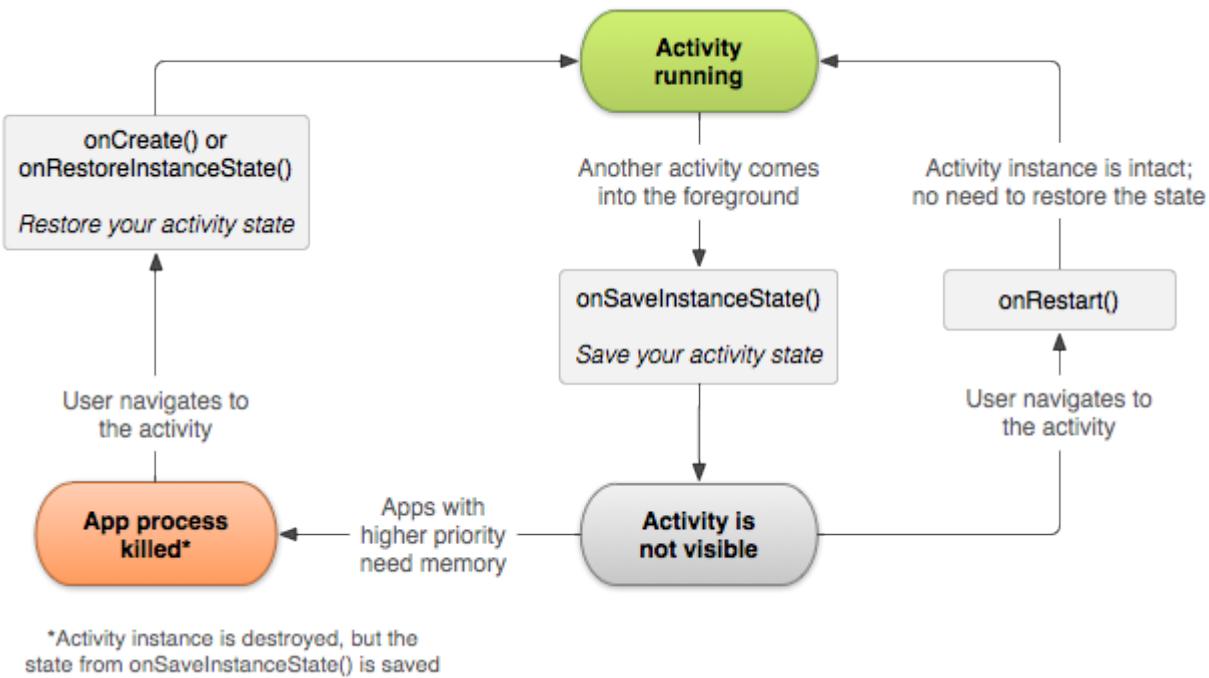
**Note:** An activity that's not technically "killable" by this definition in table 1 might still be killed by the system—but that would happen only in extreme circumstances when there is no other recourse. When an activity might be killed is discussed more in the [Processes and Threading](#) document.

## Saving activity state

The introduction to [Managing the Activity Lifecycle](#) briefly mentions that when an activity is paused or stopped, the state of the activity is retained. This is true because the [Activity](#) object is still held in memory when it is paused or stopped—all information about its members and current state is still alive. Thus, any changes the user made within the activity are retained so that when the activity returns to the foreground (when it "resumes"), those changes are still there.

However, when the system destroys an activity in order to recover memory, the [Activity](#) object is destroyed, so the system cannot simply resume it with its state intact. Instead, the system must recreate the [Activity](#) object if the user navigates back to it. Yet, the user is unaware that the system destroyed the activity and recreated it and, thus, probably expects the activity to be exactly as it was. In this situation, you can ensure that important information about the activity state is preserved by implementing an additional callback method that allows you to save information about the state of your activity: [onSaveInstanceState\(\)](#).

The system calls [onSaveInstanceState\(\)](#) before making the activity vulnerable to destruction. The system passes this method a [Bundle](#) in which you can save state information about the activity as name-value pairs, using methods such as [putString\(\)](#) and [putInt\(\)](#). Then, if the system kills your application process and the user navigates back to your activity, the system recreates the activity and passes the [Bundle](#) to both [onCreate\(\)](#) and [onRestoreInstanceState\(\)](#). Using either of these methods, you can extract your saved state from the [Bundle](#) and restore the activity state. If there is no state information to restore, then the [Bundle](#) passed to you is null (which is the case when the activity is created for the first time).



**Figure 2.** The two ways in which an activity returns to user focus with its state intact: either the activity is destroyed, then recreated and the activity must restore the previously saved state, or the activity is stopped, then resumed and the activity state remains intact.

**Note:** There's no guarantee that `onSaveInstanceState()` will be called before your activity is destroyed, because there are cases in which it won't be necessary to save the state (such as when the user leaves your activity using the *Back* button, because the user is explicitly closing the activity). If the system calls `onSaveInstanceState()`, it does so before `onStop()` and possibly before `onPause()`.

However, even if you do nothing and do not implement `onSaveInstanceState()`, some of the activity state is restored by the `Activity` class's default implementation of `onSaveInstanceState()`. Specifically, the default implementation calls the corresponding `onSaveInstanceState()` method for every `View` in the layout, which allows each view to provide information about itself that should be saved. Almost every widget in the Android framework implements this method as appropriate, such that any visible changes to the UI are automatically saved and restored when your activity is recreated. For example, the `EditText` widget saves any text entered by the user and the `CheckBox` widget saves whether it's checked or not. The only work required by you is to provide a unique ID (with the `android:id` attribute) for each widget you want to save its state. If a widget does not have an ID, then the system cannot save its state.

You can also explicitly stop a view in your layout from saving its state by setting the `android:saveEnabled` attribute to "false" or by calling the `setSaveEnabled()` method. Usually, you should not disable this, but you might if you want to restore the state of the activity UI differently.

Although the default implementation of `onSaveInstanceState()` saves useful information about your activity's UI, you still might need to override it to save additional information. For example, you might need to save member values that changed during the activity's life (which might correlate to values restored in the UI, but the members that hold those UI values are not restored, by default).

Because the default implementation of `onSaveInstanceState()` helps save the state of the UI, if you override the method in order to save additional state information, you should always call the superclass implementation of `onSaveInstanceState()` before doing any work. Likewise, you should also call the superclass implementation of `onRestoreInstanceState()` if you override it, so the default implementation can restore view states.

**Note:** Because `onSaveInstanceState()` is not guaranteed to be called, you should use it only to record the transient state of the activity (the state of the UI)—you should never use it to store persistent data. Instead, you should use `onPause()` to store persistent data (such as data that should be saved to a database) when the user leaves the activity.

A good way to test your application's ability to restore its state is to simply rotate the device so that the screen orientation changes. When the screen orientation changes, the system destroys and recreates the activity in order to apply alternative resources that might be available for the new screen configuration. For this reason alone, it's very important that your activity completely restores its state when it is recreated, because users regularly rotate the screen while using applications.

## Handling configuration changes

Some device configurations can change during runtime (such as screen orientation, keyboard availability, and language). When such a change occurs, Android recreates the running activity (the system calls `onDestroy()`, then immediately calls `onCreate()`). This behavior is designed to help your application adapt to new configurations by automatically reloading your application with alternative resources that you've provided (such as different layouts for different screen orientations and sizes).

If you properly design your activity to handle a restart due to a screen orientation change and restore the activity state as described above, your application will be more resilient to other unexpected events in the activity lifecycle.

The best way to handle such a restart is to save and restore the state of your activity using `onSaveInstanceState()` and `onRestoreInstanceState()` (or `onCreate()`), as discussed in the previous section.

For more information about configuration changes that happen at runtime and how you can handle them, read the guide to [Handling Runtime Changes](#).

## Coordinating activities

When one activity starts another, they both experience lifecycle transitions. The first activity pauses and stops (though, it won't stop if it's still visible in the background), while the other activity is created. In case these activities share data saved to disc or elsewhere, it's important to understand that the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.

The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process and one is starting the other. Here's the order of operations that occur when Activity A starts Activity B:

1. Activity A's `onPause()` method executes.
2. Activity B's `onCreate()`, `onStart()`, and `onResume()` methods execute in sequence. (Activity B now has user focus.)
3. Then, if Activity A is no longer visible on screen, its `onStop()` method executes.

This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another. For example, if you must write to a database when the first activity stops so that the following activity can read it, then you should write to the database during `onPause()` instead of during `onStop()`.

# Fragments

## Quickview

- Fragments decompose application functionality and UI into reusable modules
- Add multiple fragments to a screen to avoid switching activities
- Fragments have their own lifecycle, state, and back stack
- Fragments require API Level 11 or greater

## In this document

1. [Design Philosophy](#)
2. [Creating a Fragment](#)
  1. [Adding a user interface](#)
  2. [Adding a fragment to an activity](#)
3. [Managing Fragments](#)
4. [Performing Fragment Transactions](#)
5. [Communicating with the Activity](#)
  1. [Creating event callbacks to the activity](#)
  2. [Adding items to the Action Bar](#)
6. [Handling the Fragment Lifecycle](#)
  1. [Coordinating with the activity lifecycle](#)
7. [Example](#)

## Key classes

1. [Fragment](#)
2. [FragmentManager](#)
3. [FragmentTransaction](#)

## See also

1. [Building a Dynamic UI with Fragments](#)
2. [Supporting Tablets and Handsets](#)

A [Fragment](#) represents a behavior or a portion of user interface in an [Activity](#). You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities).

A fragment must always be embedded in an activity and the fragment's lifecycle is directly affected by the host activity's lifecycle. For example, when the activity is paused, so are all fragments in it, and when the activity is destroyed, so are all fragments. However, while an activity is running (it is in the *resumed* [lifecycle state](#)), you can manipulate each fragment independently, such as add or remove them. When you perform such a fragment transaction, you can also add it to a back stack that's managed by the activity—each back stack entry in the activity is a record of the fragment transaction that occurred. The back stack allows the user to reverse a fragment transaction (navigate backwards), by pressing the *Back* button.

When you add a fragment as a part of your activity layout, it lives in a [ViewGroup](#) inside the activity's view hierarchy and the fragment defines its own view layout. You can insert a fragment into your activity layout by

declaring the fragment in the activity's layout file, as a `<fragment>` element, or from your application code by adding it to an existing [ViewGroup](#). However, a fragment is not required to be a part of the activity layout; you may also use a fragment without its own UI as an invisible worker for the activity.

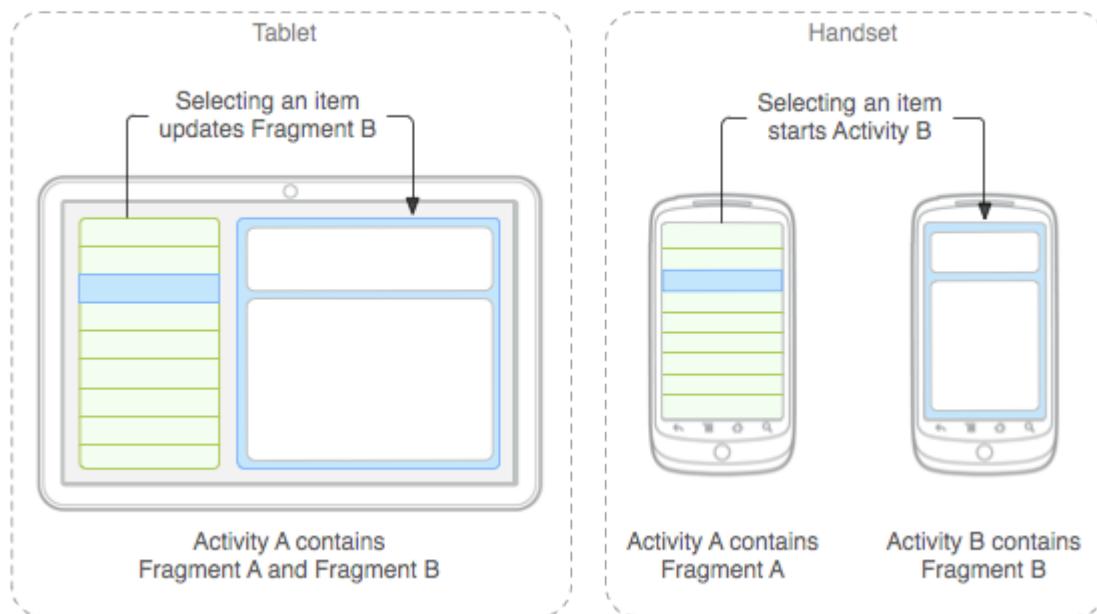
This document describes how to build your application to use fragments, including how fragments can maintain their state when added to the activity's back stack, share events with the activity and other fragments in the activity, contribute to the activity's action bar, and more.

## Design Philosophy

Android introduced fragments in Android 3.0 (API level 11), primarily to support more dynamic and flexible UI designs on large screens, such as tablets. Because a tablet's screen is much larger than that of a handset, there's more room to combine and interchange UI components. Fragments allow such designs without the need for you to manage complex changes to the view hierarchy. By dividing the layout of an activity into fragments, you become able to modify the activity's appearance at runtime and preserve those changes in a back stack that's managed by the activity.

For example, a news application can use one fragment to show a list of articles on the left and another fragment to display an article on the right—both fragments appear in one activity, side by side, and each fragment has its own set of lifecycle callback methods and handle their own user input events. Thus, instead of using one activity to select an article and another activity to read the article, the user can select an article and read it all within the same activity, as illustrated in the tablet layout in figure 1.

You should design each fragment as a modular and reusable activity component. That is, because each fragment defines its own layout and its own behavior with its own lifecycle callbacks, you can include one fragment in multiple activities, so you should design for reuse and avoid directly manipulating one fragment from another fragment. This is especially important because a modular fragment allows you to change your fragment combinations for different screen sizes. When designing your application to support both tablets and handsets, you can reuse your fragments in different layout configurations to optimize the user experience based on the available screen space. For example, on a handset, it might be necessary to separate fragments to provide a single-pane UI when more than one cannot fit within the same activity.

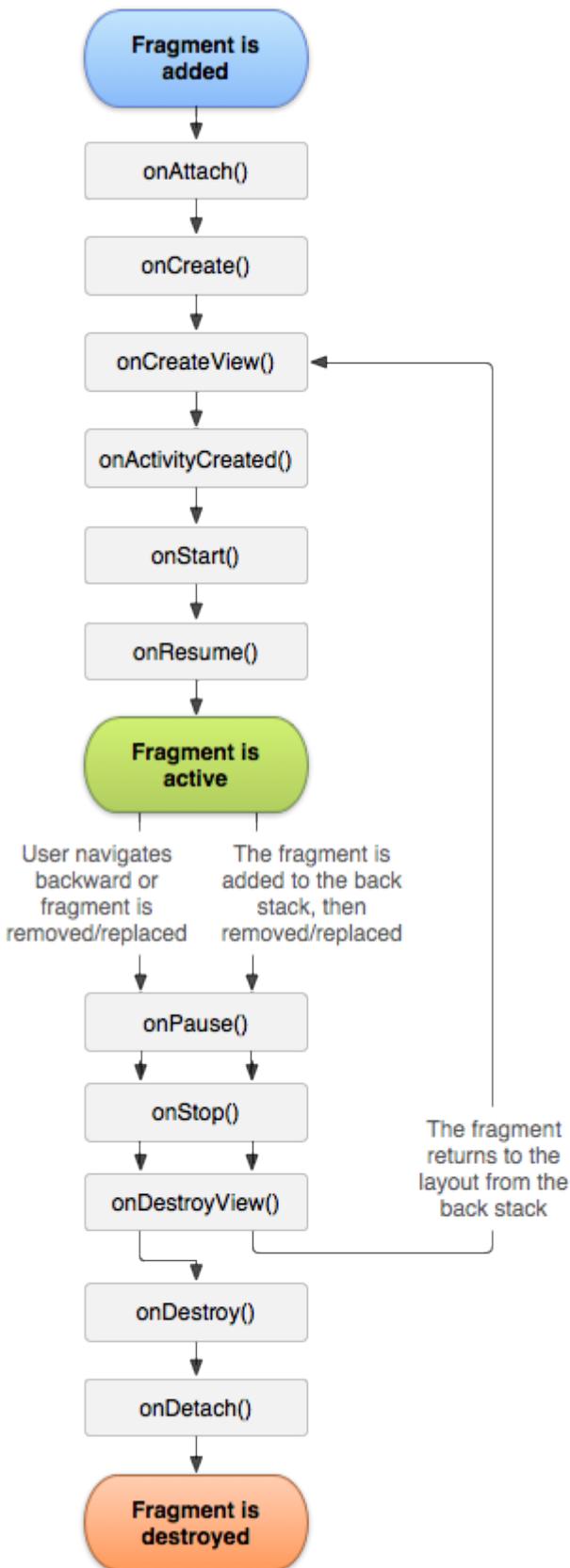


**Figure 1.** An example of how two UI modules defined by fragments can be combined into one activity for a tablet design, but separated for a handset design.

For example—to continue with the news application example—the application can embed two fragments in *Activity A*, when running on a tablet-sized device. However, on a handset-sized screen, there's not enough room for both fragments, so *Activity A* includes only the fragment for the list of articles, and when the user selects an article, it starts *Activity B*, which includes the second fragment to read the article. Thus, the application supports both tablets and handsets by reusing fragments in different combinations, as illustrated in figure 1.

For more information about designing your application with different fragment combinations for different screen configurations, see the guide to [Supporting Tablets and Handsets](#).

# Creating a Fragment



**Figure 2.** The lifecycle of a fragment (while its activity is running).

To create a fragment, you must create a subclass of `Fragment` (or an existing subclass of it). The `Fragment` class has code that looks a lot like an `Activity`. It contains callback methods similar to an activity, such as `onCreate()`, `onStart()`, `onPause()`, and `onStop()`. In fact, if you're converting an existing Android

application to use fragments, you might simply move code from your activity's callback methods into the respective callback methods of your fragment.

Usually, you should implement at least the following lifecycle methods:

#### [onCreate\(\)](#)

The system calls this when creating the fragment. Within your implementation, you should initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.

#### [onCreateView\(\)](#)

The system calls this when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a [View](#) from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.

#### [onPause\(\)](#)

The system calls this method as the first indication that the user is leaving the fragment (though it does not always mean the fragment is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

Most applications should implement at least these three methods for every fragment, but there are several other callback methods you should also use to handle various stages of the fragment lifecycle. All the lifecycle callback methods are discussed in more detail in the section about [Handling the Fragment Lifecycle](#).

There are also a few subclasses that you might want to extend, instead of the base [Fragment](#) class:

#### [DialogFragment](#)

Displays a floating dialog. Using this class to create a dialog is a good alternative to using the dialog helper methods in the [Activity](#) class, because you can incorporate a fragment dialog into the back stack of fragments managed by the activity, allowing the user to return to a dismissed fragment.

#### [ListFragment](#)

Displays a list of items that are managed by an adapter (such as a [SimpleCursorAdapter](#)), similar to [ListActivity](#). It provides several methods for managing a list view, such as the [onListItemClick\(\)](#) callback to handle click events.

#### [PreferenceFragment](#)

Displays a hierarchy of [Preference](#) objects as a list, similar to [PreferenceActivity](#). This is useful when creating a "settings" activity for your application.

## **Adding a user interface**

A fragment is usually used as part of an activity's user interface and contributes its own layout to the activity.

To provide a layout for a fragment, you must implement the [onCreateView\(\)](#) callback method, which the Android system calls when it's time for the fragment to draw its layout. Your implementation of this method must return a [View](#) that is the root of your fragment's layout.

**Note:** If your fragment is a subclass of [ListFragment](#), the default implementation returns a [ListView](#) from [onCreateView\(\)](#), so you don't need to implement it.

To return a layout from [onCreateView\(\)](#), you can inflate it from a [layout resource](#) defined in XML. To help you do so, [onCreateView\(\)](#) provides a [LayoutInflater](#) object.

For example, here's a subclass of [Fragment](#) that loads a layout from the `example_fragment.xml` file:

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

## Creating a layout

In the sample above, `R.layout.example_fragment` is a reference to a layout resource named `example_fragment.xml` saved in the application resources. For information about how to create a layout in XML, see the [User Interface](#) documentation.

The `container` parameter passed to [onCreateView\(\)](#) is the parent [ViewGroup](#) (from the activity's layout) in which your fragment layout will be inserted. The `savedInstanceState` parameter is a [Bundle](#) that provides data about the previous instance of the fragment, if the fragment is being resumed (restoring state is discussed more in the section about [Handling the Fragment Lifecycle](#)).

The [inflate\(\)](#) method takes three arguments:

- The resource ID of the layout you want to inflate.
- The [ViewGroup](#) to be the parent of the inflated layout. Passing the `container` is important in order for the system to apply layout parameters to the root view of the inflated layout, specified by the parent view in which it's going.
- A boolean indicating whether the inflated layout should be attached to the [ViewGroup](#) (the second parameter) during inflation. (In this case, this is false because the system is already inserting the inflated layout into the `container`—passing true would create a redundant view group in the final layout.)

Now you've seen how to create a fragment that provides a layout. Next, you need to add the fragment to your activity.

## Adding a fragment to an activity

Usually, a fragment contributes a portion of UI to the host activity, which is embedded as a part of the activity's overall view hierarchy. There are two ways you can add a fragment to the activity layout:

- **Declare the fragment inside the activity's layout file.**

In this case, you can specify layout properties for the fragment as if it were a view. For example, here's the layout file for an activity with two fragments:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.example.news.ArticleListFragment"
        android:id="@+id/list"
        android:layout_weight="1"
        android:layout_width="0dp"
```

```
        android:layout_height="match_parent" />
<fragment android:name="com.example.news.ArticleReaderFragment"
          android:id="@+id/viewer"
          android:layout_weight="2"
          android:layout_width="0dp"
          android:layout_height="match_parent" />
</LinearLayout>
```

The `android:name` attribute in the `<fragment>` specifies the [Fragment](#) class to instantiate in the layout.

When the system creates this activity layout, it instantiates each fragment specified in the layout and calls the [onCreateView\(\)](#) method for each one, to retrieve each fragment's layout. The system inserts the [View](#) returned by the fragment directly in place of the `<fragment>` element.

**Note:** Each fragment requires a unique identifier that the system can use to restore the fragment if the activity is restarted (and which you can use to capture the fragment to perform transactions, such as remove it). There are three ways to provide an ID for a fragment:

- Supply the `android:id` attribute with a unique ID.
- Supply the `android:tag` attribute with a unique string.
- If you provide neither of the previous two, the system uses the ID of the container view.

- **Or, programmatically add the fragment to an existing [ViewGroup](#).**

At any time while your activity is running, you can add fragments to your activity layout. You simply need to specify a [ViewGroup](#) in which to place the fragment.

To make fragment transactions in your activity (such as add, remove, or replace a fragment), you must use APIs from [FragmentTransaction](#). You can get an instance of [FragmentTransaction](#) from your [Activity](#) like this:

```
FragmentManager fragmentManager = getFragmentManager\(\)
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction\(\)
```

You can then add a fragment using the [add\(\)](#) method, specifying the fragment to add and the view in which to insert it. For example:

```
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

The first argument passed to [add\(\)](#) is the [ViewGroup](#) in which the fragment should be placed, specified by resource ID, and the second parameter is the fragment to add.

Once you've made your changes with [FragmentTransaction](#), you must call [commit\(\)](#) for the changes to take effect.

## Adding a fragment without a UI

The examples above show how to add a fragment to your activity in order to provide a UI. However, you can also use a fragment to provide a background behavior for the activity without presenting additional UI.

To add a fragment without a UI, add the fragment from the activity using [add\(Fragment, String\)](#) (supplying a unique string "tag" for the fragment, rather than a view ID). This adds the fragment, but, because it's

not associated with a view in the activity layout, it does not receive a call to `onCreateView()`. So you don't need to implement that method.

Supplying a string tag for the fragment isn't strictly for non-UI fragments—you can also supply string tags to fragments that do have a UI—but if the fragment does not have a UI, then the string tag is the only way to identify it. If you want to get the fragment from the activity later, you need to use `findFragmentByTag()`.

For an example activity that uses a fragment as a background worker, without a UI, see the [FragmentRetainInstance.java](#) sample.

## Managing Fragments

To manage the fragments in your activity, you need to use `FragmentManager`. To get it, call `getFragmentManager()` from your activity.

Some things that you can do with `FragmentManager` include:

- Get fragments that exist in the activity, with `findFragmentById()` (for fragments that provide a UI in the activity layout) or `findFragmentByTag()` (for fragments that do or don't provide a UI).
- Pop fragments off the back stack, with `popBackStack()` (simulating a *Back* command by the user).
- Register a listener for changes to the back stack, with `addOnBackStackChangedListener()`.

For more information about these methods and others, refer to the [FragmentManager](#) class documentation.

As demonstrated in the previous section, you can also use `FragmentManager` to open a `FragmentTransaction`, which allows you to perform transactions, such as add and remove fragments.

## Performing Fragment Transactions

A great feature about using fragments in your activity is the ability to add, remove, replace, and perform other actions with them, in response to user interaction. Each set of changes that you commit to the activity is called a transaction and you can perform one using APIs in `FragmentTransaction`. You can also save each transaction to a back stack managed by the activity, allowing the user to navigate backward through the fragment changes (similar to navigating backward through activities).

You can acquire an instance of `FragmentTransaction` from the `FragmentManager` like this:

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

Each transaction is a set of changes that you want to perform at the same time. You can set up all the changes you want to perform for a given transaction using methods such as `add()`, `remove()`, and `replace()`. Then, to apply the transaction to the activity, you must call `commit()`.

Before you call `commit()`, however, you might want to call `addToBackStack()`, in order to add the transaction to a back stack of fragment transactions. This back stack is managed by the activity and allows the user to return to the previous fragment state, by pressing the *Back* button.

For example, here's how you can replace one fragment with another, and preserve the previous state in the back stack:

```
// Create new fragment and transaction  
Fragment newFragment = new ExampleFragment();
```

```
FragmentTransaction transaction = getFragmentManager().beginTransaction();  
  
// Replace whatever is in the fragment_container view with this fragment,  
// and add the transaction to the back stack  
transaction.replace(R.id.fragment_container, newFragment);  
transaction.addToBackStack(null);  
  
// Commit the transaction  
transaction.commit();
```

In this example, `newFragment` replaces whatever fragment (if any) is currently in the layout container identified by the `R.id.fragment_container` ID. By calling [addToBackStack\(\)](#), the replace transaction is saved to the back stack so the user can reverse the transaction and bring back the previous fragment by pressing the *Back* button.

If you add multiple changes to the transaction (such as another [add\(\)](#) or [remove\(\)](#)) and call [addToBackStack\(\)](#), then all changes applied before you call [commit\(\)](#) are added to the back stack as a single transaction and the *Back* button will reverse them all together.

The order in which you add changes to a [FragmentTransaction](#) doesn't matter, except:

- You must call [commit\(\)](#) last
- If you're adding multiple fragments to the same container, then the order in which you add them determines the order they appear in the view hierarchy

If you do not call [addToBackStack\(\)](#) when you perform a transaction that removes a fragment, then that fragment is destroyed when the transaction is committed and the user cannot navigate back to it. Whereas, if you do call [addToBackStack\(\)](#) when removing a fragment, then the fragment is *stopped* and will be resumed if the user navigates back.

**Tip:** For each fragment transaction, you can apply a transition animation, by calling [setTransition\(\)](#) before you commit.

Calling [commit\(\)](#) does not perform the transaction immediately. Rather, it schedules it to run on the activity's UI thread (the "main" thread) as soon as the thread is able to do so. If necessary, however, you may call [executePendingTransactions\(\)](#) from your UI thread to immediately execute transactions submitted by [commit\(\)](#). Doing so is usually not necessary unless the transaction is a dependency for jobs in other threads.

**Caution:** You can commit a transaction using [commit\(\)](#) only prior to the activity [saving its state](#) (when the user leaves the activity). If you attempt to commit after that point, an exception will be thrown. This is because the state after the commit can be lost if the activity needs to be restored. For situations in which it's okay that you lose the commit, use [commitAllowingStateLoss\(\)](#).

## Communicating with the Activity

Although a [Fragment](#) is implemented as an object that's independent from an [Activity](#) and can be used inside multiple activities, a given instance of a fragment is directly tied to the activity that contains it.

Specifically, the fragment can access the [Activity](#) instance with [getActivity\(\)](#) and easily perform tasks such as find a view in the activity layout:

```
View listView = getActivity().findViewById(R.id.list);
```

Likewise, your activity can call methods in the fragment by acquiring a reference to the [Fragment](#) from [FragmentManager](#), using [findFragmentById\(\)](#) or [findFragmentByTag\(\)](#). For example:

```
ExampleFragment fragment = (ExampleFragment) getSupportFragmentManager().findFragmentByTag("tag");
```

## Creating event callbacks to the activity

In some cases, you might need a fragment to share events with the activity. A good way to do that is to define a callback interface inside the fragment and require that the host activity implement it. When the activity receives a callback through the interface, it can share the information with other fragments in the layout as necessary.

For example, if a news application has two fragments in an activity—one to show a list of articles (fragment A) and another to display an article (fragment B)—then fragment A must tell the activity when a list item is selected so that it can tell fragment B to display the article. In this case, the `OnArticleSelectedListener` interface is declared inside fragment A:

```
public static class FragmentA extends ListFragment {  
    ...  
    // Container Activity must implement this interface  
    public interface OnArticleSelectedListener {  
        public void onArticleSelected(Uri articleUri);  
    }  
    ...  
}
```

Then the activity that hosts the fragment implements the `OnArticleSelectedListener` interface and overrides `onArticleSelected()` to notify fragment B of the event from fragment A. To ensure that the host activity implements this interface, fragment A's [onAttach\(\)](#) callback method (which the system calls when adding the fragment to the activity) instantiates an instance of `OnArticleSelectedListener` by casting the `Activity` that is passed into [onAttach\(\)](#):

```
public static class FragmentA extends ListFragment {  
    OnArticleSelectedListener mListener;  
    ...  
    @Override  
    public void onAttach(Activity activity) {  
        super.onAttach(activity);  
        try {  
            mListener = (OnArticleSelectedListener) activity;  
        } catch (ClassCastException e) {  
            throw new ClassCastException(activity.toString() + " must implement  
        }  
    }  
    ...  
}
```

If the activity has not implemented the interface, then the fragment throws a [ClassCastException](#). On success, the `mListener` member holds a reference to activity's implementation of `OnArticleSelectedListener`, so that fragment A can share events with the activity by calling methods defined by the `OnArticleSelectedListener` interface. For example, if fragment A is an extension of [ListFragment](#), each time the user clicks a list item, the system calls [onListItemClick\(\)](#) in the fragment, which then calls `onArticleSelected()` to share the event with the activity:

```
public static class FragmentA extends ListFragment {  
    OnArticleSelectedListener mListener;  
    ...  
    @Override  
    public void onListItemClick(ListView l, View v, int position, long id) {  
        // Append the clicked item's row ID with the content provider Uri  
        Uri noteUri = ContentUris.withAppendedId(ArticleColumns.CONTENT_URI, id);  
        // Send the event and Uri to the host activity  
        mListener.onArticleSelected(noteUri);  
    }  
    ...  
}
```

The `id` parameter passed to [onListItemClick\(\)](#) is the row ID of the clicked item, which the activity (or other fragment) uses to fetch the article from the application's [Content Provider](#).

More information about using a content provider is available in the [Content Providers](#) document.

## Adding items to the Action Bar

Your fragments can contribute menu items to the activity's [Options Menu](#) (and, consequently, the [Action Bar](#)) by implementing [onCreateOptionsMenu\(\)](#). In order for this method to receive calls, however, you must call [setHasOptionsMenu\(\)](#) during [onCreate\(\)](#), to indicate that the fragment would like to add items to the Options Menu (otherwise, the fragment will not receive a call to [onCreateOptionsMenu\(\)](#)).

Any items that you then add to the Options Menu from the fragment are appended to the existing menu items. The fragment also receives callbacks to [onOptionsItemSelected\(\)](#) when a menu item is selected.

You can also register a view in your fragment layout to provide a context menu by calling [registerForContextMenu\(\)](#). When the user opens the context menu, the fragment receives a call to [onCreateContextMenu\(\)](#). When the user selects an item, the fragment receives a call to [onContextItemSelected\(\)](#).

**Note:** Although your fragment receives an on-item-selected callback for each menu item it adds, the activity is first to receive the respective callback when the user selects a menu item. If the activity's implementation of the on-item-selected callback does not handle the selected item, then the event is passed to the fragment's callback. This is true for the Options Menu and context menus.

For more information about menus, see the [Menus](#) and [Action Bar](#) developer guides.

# Handling the Fragment Lifecycle



**Figure 3.** The effect of the activity lifecycle on the fragment lifecycle.

Managing the lifecycle of a fragment is a lot like managing the lifecycle of an activity. Like an activity, a fragment can exist in three states:

## ***Resumed***

The fragment is visible in the running activity.

## ***Paused***

Another activity is in the foreground and has focus, but the activity in which this fragment lives is still visible (the foreground activity is partially transparent or doesn't cover the entire screen).

## ***Stopped***

The fragment is not visible. Either the host activity has been stopped or the fragment has been removed from the activity but added to the back stack. A stopped fragment is still alive (all state and member infor-

mation is retained by the system). However, it is no longer visible to the user and will be killed if the activity is killed.

Also like an activity, you can retain the state of a fragment using a [Bundle](#), in case the activity's process is killed and you need to restore the fragment state when the activity is recreated. You can save the state during the fragment's [onSaveInstanceState\(\)](#) callback and restore it during either [onCreate\(\)](#), [onCreateView\(\)](#), or [onActivityCreated\(\)](#). For more information about saving state, see the [Activities](#) document.

The most significant difference in lifecycle between an activity and a fragment is how one is stored in its respective back stack. An activity is placed into a back stack of activities that's managed by the system when it's stopped, by default (so that the user can navigate back to it with the *Back* button, as discussed in [Tasks and Back Stack](#)). However, a fragment is placed into a back stack managed by the host activity only when you explicitly request that the instance be saved by calling [addToBackStack\(\)](#) during a transaction that removes the fragment.

Otherwise, managing the fragment lifecycle is very similar to managing the activity lifecycle. So, the same practices for [managing the activity lifecycle](#) also apply to fragments. What you also need to understand, though, is how the life of the activity affects the life of the fragment.

**Caution:** If you need a [Context](#) object within your [Fragment](#), you can call [getActivity\(\)](#). However, be careful to call [getActivity\(\)](#) only when the fragment is attached to an activity. When the fragment is not yet attached, or was detached during the end of its lifecycle, [getActivity\(\)](#) will return null.

## Coordinating with the activity lifecycle

The lifecycle of the activity in which the fragment lives directly affects the lifecycle of the fragment, such that each lifecycle callback for the activity results in a similar callback for each fragment. For example, when the activity receives [onPause\(\)](#), each fragment in the activity receives [onPause\(\)](#).

Fragments have a few extra lifecycle callbacks, however, that handle unique interaction with the activity in order to perform actions such as build and destroy the fragment's UI. These additional callback methods are:

### [onAttach\(\)](#)

Called when the fragment has been associated with the activity (the [Activity](#) is passed in here).

### [onCreateView\(\)](#)

Called to create the view hierarchy associated with the fragment.

### [onActivityCreated\(\)](#)

Called when the activity's [onCreate\(\)](#) method has returned.

### [onDestroyView\(\)](#)

Called when the view hierarchy associated with the fragment is being removed.

### [onDetach\(\)](#)

Called when the fragment is being disassociated from the activity.

The flow of a fragment's lifecycle, as it is affected by its host activity, is illustrated by figure 3. In this figure, you can see how each successive state of the activity determines which callback methods a fragment may receive. For example, when the activity has received its [onCreate\(\)](#) callback, a fragment in the activity receives no more than the [onActivityCreated\(\)](#) callback.

Once the activity reaches the resumed state, you can freely add and remove fragments to the activity. Thus, only while the activity is in the resumed state can the lifecycle of a fragment change independently.

However, when the activity leaves the resumed state, the fragment again is pushed through its lifecycle by the activity.

## Example

To bring everything discussed in this document together, here's an example of an activity using two fragments to create a two-pane layout. The activity below includes one fragment to show a list of Shakespeare play titles and another to show a summary of the play when selected from the list. It also demonstrates how to provide different configurations of the fragments, based on the screen configuration.

**Note:** The complete source code for this activity is available in [FragmentLayout.java](#).

The main activity applies a layout in the usual way, during [onCreate\(\)](#):

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

The layout applied is `fragment_layout.xml`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="horizontal"  
    android:layout_width="match_parent" android:layout_height="match_parent">  
  
    <fragment class="com.example.android.apis.app.FragmentLayout$TitlesFragment"  
        android:id="@+id/titles" android:layout_weight="1"  
        android:layout_width="0px" android:layout_height="match_parent" />  
  
    <FrameLayout android:id="@+id/details" android:layout_weight="1"  
        android:layout_width="0px" android:layout_height="match_parent"  
        android:background="?android:attr/detailsElementBackground" />  
  
</LinearLayout>
```

Using this layout, the system instantiates the `TitlesFragment` (which lists the play titles) as soon as the activity loads the layout, while the `FrameLayout` (where the fragment for showing the play summary will go) consumes space on the right side of the screen, but remains empty at first. As you'll see below, it's not until the user selects an item from the list that a fragment is placed into the `FrameLayout`.

However, not all screen configurations are wide enough to show both the list of plays and the summary, side by side. So, the layout above is used only for the landscape screen configuration, by saving it at `res/layout-land/fragment_layout.xml`.

Thus, when the screen is in portrait orientation, the system applies the following layout, which is saved at `res/layout/fragment_layout.xml`:

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

```
<fragment class="com.example.android.apis.app.FragmentLayout$TitlesFragment"
    android:id="@+id/titles"
    android:layout_width="match_parent" android:layout_height="match_parent"
</FrameLayout>
```

This layout includes only `TitlesFragment`. This means that, when the device is in portrait orientation, only the list of play titles is visible. So, when the user clicks a list item in this configuration, the application will start a new activity to show the summary, instead of loading a second fragment.

Next, you can see how this is accomplished in the fragment classes. First is `TitlesFragment`, which shows the list of Shakespeare play titles. This fragment extends [ListFragment](#) and relies on it to handle most of the list view work.

As you inspect this code, notice that there are two possible behaviors when the user clicks a list item: depending on which of the two layouts is active, it can either create and display a new fragment to show the details in the same activity (adding the fragment to the [FrameLayout](#)), or start a new activity (where the fragment can be shown).

```
public static class TitlesFragment extends ListFragment {
    boolean mDualPane;
    int mCurCheckPosition = 0;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        // Populate list with our static array of titles.
        setListAdapter(new ArrayAdapter<String>(getActivity(),
            android.R.layout.simple_list_item_activated_1, Shakespeare.TITLES));

        // Check to see if we have a frame in which to embed the details
        // fragment directly in the containing UI.
        View detailsFrame = getActivity().findViewById(R.id.details);
        mDualPane = detailsFrame != null && detailsFrame.getVisibility() == View.VISIBLE;

        if (savedInstanceState != null) {
            // Restore last state for checked position.
            mCurCheckPosition = savedInstanceState.getInt("curChoice", 0);
        }

        if (mDualPane) {
            // In dual-pane mode, the list view highlights the selected item.
            getListView().setSelectionMode(ListView.CHOICE_MODE_SINGLE);
            // Make sure our UI is in the correct state.
            showDetails(mCurCheckPosition);
        }
    }

    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putInt("curChoice", mCurCheckPosition);
    }
}
```

```

@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    showDetails(position);
}

/**
 * Helper function to show the details of a selected item, either by
 * displaying a fragment in-place in the current UI, or starting a
 * whole new activity in which it is displayed.
 */
void showDetails(int index) {
    mCurCheckPosition = index;

    if (mDualPane) {
        // We can display everything in-place with fragments, so update
        // the list to highlight the selected item and show the data.
        getListView().setItemChecked(index, true);

        // Check what fragment is currently shown, replace if needed.
        DetailsFragment details = (DetailsFragment)
            getSupportFragmentManager().findFragmentById(R.id.details);
        if (details == null || details.getShownIndex() != index) {
            // Make new fragment to show this selection.
            details = DetailsFragment.newInstance(index);

            // Execute a transaction, replacing any existing fragment
            // with this one inside the frame.
            FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
            if (index == 0) {
                ft.replace(R.id.details, details);
            } else {
                ft.replace(R.id.a_item, details);
            }
            ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
            ft.commit();
        }
    } else {
        // Otherwise we need to launch a new activity to display
        // the dialog fragment with selected text.
        Intent intent = new Intent();
        intent.setClass(getActivity(), DetailsActivity.class);
        intent.putExtra("index", index);
        startActivity(intent);
    }
}
}

```

The second fragment, DetailsFragment shows the play summary for the item selected from the list from TitlesFragment:

```

public static class DetailsFragment extends Fragment {
    /**
     * Create a new instance of DetailsFragment, initialized to

```

```

        * show the text at 'index'.
    */
public static DetailsFragment newInstance(int index) {
    DetailsFragment f = new DetailsFragment();

    // Supply index input as an argument.
    Bundle args = new Bundle();
    args.putInt("index", index);
    f.setArguments(args);

    return f;
}

public int getShownIndex() {
    return getArguments().getInt("index", 0);
}

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    if (container == null) {
        // We have different layouts, and in one of them this
        // fragment's containing frame doesn't exist. The fragment
        // may still be created from its saved state, but there is
        // no reason to try to create its view hierarchy because it
        // won't be displayed. Note this is not needed -- we could
        // just run the code below, where we would create and return
        // the view hierarchy; it would just never be used.
        return null;
    }

    ScrollView scroller = new ScrollView(getActivity());
    TextView text = new TextView(getActivity());
    int padding = (int)TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP
        4, getActivity().getResources().getDisplayMetrics());
    text.setPadding(padding, padding, padding, padding);
    scroller.addView(text);
    text.setText(Shakespeare.DIALOGUE[getShownIndex()]);
    return scroller;
}
}

```

Recall from the `TitlesFragment` class, that, if the user clicks a list item and the current layout does *not* include the `R.id.details` view (which is where the `DetailsFragment` belongs), then the application starts the `DetailsActivity` activity to display the content of the item.

Here is the `DetailsActivity`, which simply embeds the `DetailsFragment` to display the selected play summary when the screen is in portrait orientation:

```

public static class DetailsActivity extends Activity {

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

```

```
if (getResources().getConfiguration().orientation
    == Configuration.ORIENTATION_LANDSCAPE) {
    // If the screen is now in landscape mode, we can show the
    // dialog in-line with the list so we don't need this activity.
    finish();
    return;
}

if (savedInstanceState == null) {
    // During initial setup, plug in the details fragment.
    DetailsFragment details = new DetailsFragment();
    details.setArguments(getIntent().getExtras());
    getFragmentManager().beginTransaction().add(android.R.id.content,
}
}
}
```

Notice that this activity finishes itself if the configuration is landscape, so that the main activity can take over and display the `DetailsFragment` alongside the `TitlesFragment`. This can happen if the user begins the `DetailsActivity` while in portrait orientation, but then rotates to landscape (which restarts the current activity).

For more samples using fragments (and complete source files for this example), see the API Demos sample app available in [ApiDemos](#) (available for download from the [Samples SDK component](#)).

# Loaders

## In this document

1. [Loader API Summary](#)
2. [Using Loaders in an Application](#)
  - 1.
  2. [Starting a Loader](#)
  3. [Restarting a Loader](#)
  4. [Using the LoaderManager Callbacks](#)
3. [Example](#)
  1. [More Examples](#)

## Key classes

1. [LoaderManager](#)
2. [Loader](#)

## Related samples

1. [LoaderCursor](#)
2. [LoaderThrottle](#)

Introduced in Android 3.0, loaders make it easy to asynchronously load data in an activity or fragment. Loaders have these characteristics:

- They are available to every [Activity](#) and [Fragment](#).
- They provide asynchronous loading of data.
- They monitor the source of their data and deliver new results when the content changes.
- They automatically reconnect to the last loader's cursor when being recreated after a configuration change. Thus, they don't need to re-query their data.

## Loader API Summary

There are multiple classes and interfaces that may be involved in using loaders in an application. They are summarized in this table:

Class/Interface	Description
<a href="#">LoaderManager</a>	An abstract class associated with an <a href="#">Activity</a> or <a href="#">Fragment</a> for managing one or more <a href="#">Loader</a> instances. This helps an application manage longer-running operations in conjunction with the <a href="#">Activity</a> or <a href="#">Fragment</a> lifecycle; the most common use of this is with a <a href="#">CursorLoader</a> , however applications are free to write their own loaders for loading other types of data.
<a href="#">LoaderCallbacks</a>	There is only one <a href="#">LoaderManager</a> per activity or fragment. But a <a href="#">LoaderManager</a> can have multiple loaders. A callback interface for a client to interact with the <a href="#">LoaderManager</a> . For example, you use the <a href="#">onCreateLoader()</a> callback method to create a new loader.

## [Loader](#)

An abstract class that performs asynchronous loading of data. This is the base class for a loader. You would typically use [CursorLoader](#), but you can implement your own subclass. While loaders are active they should monitor the source of their data and deliver new results when the contents change.

## [AsyncTaskLoader](#)

Abstract loader that provides an [AsyncTask](#) to do the work.

## [CursorLoader](#)

A subclass of [AsyncTaskLoader](#) that queries the [ContentResolver](#) and returns a [Cursor](#). This class implements the [Loader](#) protocol in a standard way for querying cursors, building on [AsyncTaskLoader](#) to perform the cursor query on a background thread so that it does not block the application's UI. Using this loader is the best way to asynchronously load data from a [ContentProvider](#), instead of performing a managed query through the fragment or activity's APIs.

The classes and interfaces in the above table are the essential components you'll use to implement a loader in your application. You won't need all of them for each loader you create, but you'll always need a reference to the [LoaderManager](#) in order to initialize a loader and an implementation of a [Loader](#) class such as [CursorLoader](#). The following sections show you how to use these classes and interfaces in an application.

# Using Loaders in an Application

This section describes how to use loaders in an Android application. An application that uses loaders typically includes the following:

- An [Activity](#) or [Fragment](#).
- An instance of the [LoaderManager](#).
- A [CursorLoader](#) to load data backed by a [ContentProvider](#). Alternatively, you can implement your own subclass of [Loader](#) or [AsyncTaskLoader](#) to load data from some other source.
- An implementation for [LoaderManager.LoaderCallbacks](#). This is where you create new loaders and manage your references to existing loaders.
- A way of displaying the loader's data, such as a [SimpleCursorAdapter](#).
- A data source, such as a [ContentProvider](#), when using a [CursorLoader](#).

## Starting a Loader

The [LoaderManager](#) manages one or more [Loader](#) instances within an [Activity](#) or [Fragment](#). There is only one [LoaderManager](#) per activity or fragment.

You typically initialize a [Loader](#) within the activity's [onCreate\(\)](#) method, or within the fragment's [onActivityCreated\(\)](#) method. You do this as follows:

```
// Prepare the loader. Either re-connect with an existing one,  
// or start a new one.  
getLoaderManager().initLoader(0, null, this);
```

The [initLoader\(\)](#) method takes the following parameters:

- A unique ID that identifies the loader. In this example, the ID is 0.
- Optional arguments to supply to the loader at construction (null in this example).

- A `LoaderManager.LoaderCallbacks` implementation, which the `LoaderManager` calls to report loader events. In this example, the local class implements the `LoaderManager.LoaderCallbacks` interface, so it passes a reference to itself, `this`.

The `initLoader()` call ensures that a loader is initialized and active. It has two possible outcomes:

- If the loader specified by the ID already exists, the last created loader is reused.
- If the loader specified by the ID does *not* exist, `initLoader()` triggers the `LoaderManager.LoaderCallbacks` method `onCreateLoader()`. This is where you implement the code to instantiate and return a new loader. For more discussion, see the section [onCreateLoader](#).

In either case, the given `LoaderManager.LoaderCallbacks` implementation is associated with the loader, and will be called when the loader state changes. If at the point of this call the caller is in its started state, and the requested loader already exists and has generated its data, then the system calls `onLoadFinished()` immediately (during `initLoader()`), so you must be prepared for this to happen. See [onLoadFinished](#) for more discussion of this callback

Note that the `initLoader()` method returns the `Loader` that is created, but you don't need to capture a reference to it. The `LoaderManager` manages the life of the loader automatically. The `LoaderManager` starts and stops loading when necessary, and maintains the state of the loader and its associated content. As this implies, you rarely interact with loaders directly (though for an example of using loader methods to fine-tune a loader's behavior, see the [LoaderThrottle](#) sample). You most commonly use the `LoaderManager.LoaderCallbacks` methods to intervene in the loading process when particular events occur. For more discussion of this topic, see [Using the LoaderManager Callbacks](#).

## Restarting a Loader

When you use `initLoader()`, as shown above, it uses an existing loader with the specified ID if there is one. If there isn't, it creates one. But sometimes you want to discard your old data and start over.

To discard your old data, you use `restartLoader()`. For example, this implementation of `SearchView.OnQueryTextListener` restarts the loader when the user's query changes. The loader needs to be restarted so that it can use the revised search filter to do a new query:

```
public boolean onQueryTextChanged(String newText) {
    // Called when the action bar search text has changed. Update
    // the search filter, and restart the loader to do a new query
    // with this filter.
    mCurFilter = !TextUtils.isEmpty(newText) ? newText : null;
    getLoaderManager().restartLoader(0, null, this);
    return true;
}
```

## Using the LoaderManager Callbacks

`LoaderManager.LoaderCallbacks` is a callback interface that lets a client interact with the `LoaderManager`.

Loaders, in particular `CursorLoader`, are expected to retain their data after being stopped. This allows applications to keep their data across the activity or fragment's `onStop()` and `onStart()` methods, so that when users return to an application, they don't have to wait for the data to reload. You use the `LoaderManager.LoaderCallbacks` methods when to know when to create a new loader, and to tell the application when it is time to stop using a loader's data.

`LoaderManager.LoaderCallbacks` includes these methods:

- `onCreateLoader()` — Instantiate and return a new `Loader` for the given ID.
  - `onLoadFinished()` — Called when a previously created loader has finished its load.
  - `onLoaderReset()` — Called when a previously created loader is being reset, thus making its data unavailable.

These methods are described in more detail in the following sections.

### **onCreateLoader**

When you attempt to access a loader (for example, through `initLoader()`), it checks to see whether the loader specified by the ID exists. If it doesn't, it triggers the `LoaderManager.LoaderCallbacks` method `onCreateLoader()`. This is where you create a new loader. Typically this will be a `CursorLoader`, but you can implement your own `Loader` subclass.

In this example, the `onCreateLoader()` callback method creates a `CursorLoader`. You must build the `CursorLoader` using its constructor method, which requires the complete set of information needed to perform a query to the `ContentProvider`. Specifically, it needs:

- *uri* — The URI for the content to retrieve.
  - *projection* — A list of which columns to return. Passing `null` will return all columns, which is inefficient.
  - *selection* — A filter declaring which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). Passing `null` will return all rows for the given URI.
  - *selectionArgs* — You may include ?s in the selection, which will be replaced by the values from *selectionArgs*, in the order that they appear in the selection. The values will be bound as Strings.
  - *sortOrder* — How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing `null` will use the default sort order, which may be unordered.

For example:

```
// If non-null, this is the current filter the user has provided.  
String mCurFilter;  
...  
public Loader<Cursor> onCreateLoader(int id, Bundle args) {  
    // This is called when a new Loader needs to be created.  This  
    // sample only has one Loader, so we don't care about the ID.  
    // First, pick the base URI to use depending on whether we are  
    // currently filtering.  
    Uri baseUri;  
    if (mCurFilter != null) {  
        baseUri = Uri.withAppendedPath(Contacts.CONTENT_FILTER_URI,  
            Uri.encode(mCurFilter));  
    } else {  
        baseUri = Contacts.CONTENT_URI;  
    }  
  
    // Now create and return a CursorLoader that will take care of  
    // creating a Cursor for the data being displayed.  
    String select = "(" + Contacts.DISPLAY_NAME + " NOTNULL) AND ("  
        + Contacts.HAS_PHONE_NUMBER + "=1) AND ("  
        + Contacts.DISPLAY_NAME + " != '' ))";
```

```
        return new CursorLoader(getActivity(), baseUri,
            CONTACTS_SUMMARY_PROJECTION, select, null,
            Contacts.DISPLAY_NAME + " COLLATE LOCALIZED ASC");
    }
```

## onLoadFinished

This method is called when a previously created loader has finished its load. This method is guaranteed to be called prior to the release of the last data that was supplied for this loader. At this point you should remove all use of the old data (since it will be released soon), but should not do your own release of the data since its loader owns it and will take care of that.

The loader will release the data once it knows the application is no longer using it. For example, if the data is a cursor from a [CursorLoader](#), you should not call [close\(\)](#) on it yourself. If the cursor is being placed in a [CursorAdapter](#), you should use the [swapCursor\(\)](#) method so that the old [Cursor](#) is not closed. For example:

```
// This is the Adapter being used to display the list's data.
SimpleCursorAdapter mAdapter;
...

public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    // Swap the new cursor in.  (The framework will take care of closing the
    // old cursor once we return.)
    mAdapter.swapCursor(data);
}
```

## onLoaderReset

This method is called when a previously created loader is being reset, thus making its data unavailable. This callback lets you find out when the data is about to be released so you can remove your reference to it.

This implementation calls [swapCursor\(\)](#) with a value of `null`:

```
// This is the Adapter being used to display the list's data.
SimpleCursorAdapter mAdapter;
...

public void onLoaderReset(Loader<Cursor> loader) {
    // This is called when the last Cursor provided to onLoadFinished()
    // above is about to be closed.  We need to make sure we are no
    // longer using it.
    mAdapter.swapCursor(null);
}
```

## Example

As an example, here is the full implementation of a [Fragment](#) that displays a [ListView](#) containing the results of a query against the contacts content provider. It uses a [CursorLoader](#) to manage the query on the provider.

For an application to access a user's contacts, as shown in this example, its manifest must include the permission [READ\\_CONTACTS](#).

```
public static class CursorLoaderListFragment extends ListFragment
    implements OnQueryTextListener, LoaderManager.LoaderCallbacks<Cursor> {

    // This is the Adapter being used to display the list's data.
    SimpleCursorAdapter mAdapter;

    // If non-null, this is the current filter the user has provided.
    String mCurFilter;

    @Override public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        // Give some text to display if there is no data. In a real
        // application this would come from a resource.
        setEmptyText("No phone numbers");

        // We have a menu item to show in action bar.
        setHasOptionsMenu(true);

        // Create an empty adapter we will use to display the loaded data.
        mAdapter = new SimpleCursorAdapter(getActivity(),
            android.R.layout.simple_list_item_2, null,
            new String[] { Contacts.DISPLAY_NAME, Contacts.CONTACT_STATUS },
            new int[] { android.R.id.text1, android.R.id.text2 }, 0);
        setListAdapter(mAdapter);

        // Prepare the loader. Either re-connect with an existing one,
        // or start a new one.
        getLoaderManager().initLoader(0, null, this);
    }

    @Override public void onCreateOptionsMenu(Menu menu, MenuInflater inflater)
        // Place an action bar item for searching.
        MenuItem item = menu.add("Search");
        item.setIcon(android.R.drawable.ic_menu_search);
        item.setShowAsAction(MenuItem.SHOW_AS_ACTION_IF_ROOM);
        SearchView sv = new SearchView(getActivity());
        sv.setOnQueryTextListener(this);
        item.setActionView(sv);
    }

    public boolean onQueryTextChange(String newText) {
        // Called when the action bar search text has changed. Update
        // the search filter, and restart the loader to do a new query
        // with this filter.
        mCurFilter = !TextUtils.isEmpty(newText) ? newText : null;
        getLoaderManager().restartLoader(0, null, this);
        return true;
    }

    @Override public boolean onQueryTextSubmit(String query) {
        // Don't care about this.
        return true;
    }
}
```

```
@Override public void onListItemClick(ListView l, View v, int position, long id) {
    // Insert desired behavior here.
    Log.i("FragmentComplexList", "Item clicked: " + id);
}

// These are the Contacts rows that we will retrieve.
static final String[] CONTACTS_SUMMARY_PROJECTION = new String[] {
    Contacts._ID,
    Contacts.DISPLAY_NAME,
    Contacts.CONTACT_STATUS,
    Contacts.CONTACT_PRESENCE,
    Contacts.PHOTO_ID,
    Contacts.LOOKUP_KEY,
};
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    // This is called when a new Loader needs to be created.  This
    // sample only has one Loader, so we don't care about the ID.
    // First, pick the base URI to use depending on whether we are
    // currently filtering.
    Uri baseUri;
    if (mCurFilter != null) {
        baseUri = Uri.withAppendedPath(Contacts.CONTENT_FILTER_URI,
            Uri.encode(mCurFilter));
    } else {
        baseUri = Contacts.CONTENT_URI;
    }

    // Now create and return a CursorLoader that will take care of
    // creating a Cursor for the data being displayed.
    String select = "(((" + Contacts.DISPLAY_NAME + " NOTNULL) AND (
        + Contacts.HAS_PHONE_NUMBER + "=1) AND (
        + Contacts.DISPLAY_NAME + " != ' ' ))";
    return new CursorLoader(getActivity(), baseUri,
        CONTACTS_SUMMARY_PROJECTION, select, null,
        Contacts.DISPLAY_NAME + " COLLATE LOCALIZED ASC");
}

public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    // Swap the new cursor in.  (The framework will take care of closing the
    // old cursor once we return.)
    mAdapter.swapCursor(data);
}

public void onLoaderReset(Loader<Cursor> loader) {
    // This is called when the last Cursor provided to onLoadFinished()
    // above is about to be closed.  We need to make sure we are no
    // longer using it.
    mAdapter.swapCursor(null);
}
```

## More Examples

There are a few different samples in **ApiDemos** that illustrate how to use loaders:

- [LoaderCursor](#) — A complete version of the snippet shown above.
- [LoaderThrottle](#) — An example of how to use throttling to reduce the number of queries a content provider does when its data changes.

For information on downloading and installing the SDK samples, see [Getting the Samples](#).

# Tasks and Back Stack

## Quickview

- All activities belong to a task
- A task contains a collection of activities in the order in which the user interacts with them
- Tasks can move to the background and retain the state of each activity in order for users to perform other tasks without losing their work

## In this document

1. [Saving Activity State](#)
2. [Managing Tasks](#)
  1. [Defining launch modes](#)
  2. [Handling affinities](#)
  3. [Clearing the back stack](#)
  4. [Starting a task](#)

## Articles

1. [Multitasking the Android Way](#)

## See also

1. [Android Design: Navigation](#)
2. [<activity> manifest element](#)

An application usually contains multiple [activities](#). Each activity should be designed around a specific kind of action the user can perform and can start other activities. For example, an email application might have one activity to show a list of new email. When the user selects an email, a new activity opens to view that email.

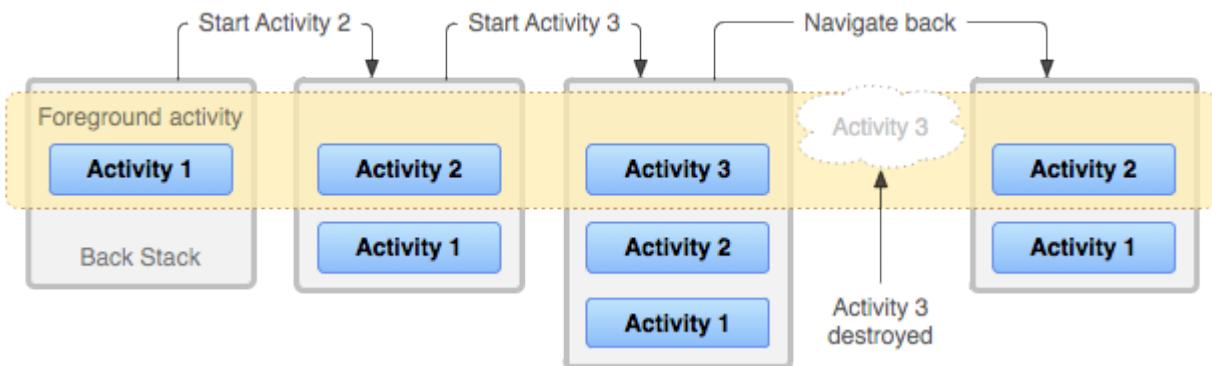
An activity can even start activities that exist in other applications on the device. For example, if your application wants to send an email, you can define an intent to perform a "send" action and include some data, such as an email address and a message. An activity from another application that declares itself to handle this kind of intent then opens. In this case, the intent is to send an email, so an email application's "compose" activity starts (if multiple activities support the same intent, then the system lets the user select which one to use). When the email is sent, your activity resumes and it seems as if the email activity was part of your application. Even though the activities may be from different applications, Android maintains this seamless user experience by keeping both activities in the same *task*.

A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack (the "back stack"), in the order in which each activity is opened.

The device Home screen is the starting place for most tasks. When the user touches an icon in the application launcher (or a shortcut on the Home screen), that application's task comes to the foreground. If no task exists for the application (the application has not been used recently), then a new task is created and the "main" activity for that application opens as the root activity in the stack.

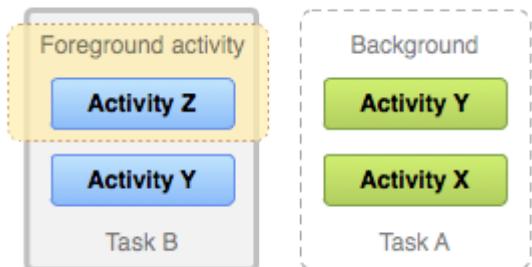
When the current activity starts another, the new activity is pushed on the top of the stack and takes focus. The previous activity remains in the stack, but is stopped. When an activity stops, the system retains the current state of its user interface. When the user presses the *Back* button, the current activity is popped from the top of

the stack (the activity is destroyed) and the previous activity resumes (the previous state of its UI is restored). Activities in the stack are never rearranged, only pushed and popped from the stack—pushed onto the stack when started by the current activity and popped off when the user leaves it using the *Back* button. As such, the back stack operates as a "last in, first out" object structure. Figure 1 visualizes this behavior with a timeline showing the progress between activities along with the current back stack at each point in time.



**Figure 1.** A representation of how each new activity in a task adds an item to the back stack. When the user presses the *Back* button, the current activity is destroyed and the previous activity resumes.

If the user continues to press *Back*, then each activity in the stack is popped off to reveal the previous one, until the user returns to the Home screen (or to whichever activity was running when the task began). When all activities are removed from the stack, the task no longer exists.



**Figure 2.** Two tasks: Task B receives user interaction in the foreground, while Task A is in the background, waiting to be resumed.



**Figure 3.** A single activity is instantiated multiple times.

A task is a cohesive unit that can move to the "background" when users begin a new task or go to the Home screen, via the *Home* button. While in the background, all the activities in the task are stopped, but the back stack for the task remains intact—the task has simply lost focus while another task takes place, as shown in figure 2. A task can then return to the "foreground" so users can pick up where they left off. Suppose, for example, that the current task (Task A) has three activities in its stack—two under the current activity. The user presses

the *Home* button, then starts a new application from the application launcher. When the Home screen appears, Task A goes into the background. When the new application starts, the system starts a task for that application (Task B) with its own stack of activities. After interacting with that application, the user returns Home again and selects the application that originally started Task A. Now, Task A comes to the foreground—all three activities in its stack are intact and the activity at the top of the stack resumes. At this point, the user can also switch back to Task B by going Home and selecting the application icon that started that task (or by selecting the app's task from the *recent apps* screen). This is an example of multitasking on Android.

**Note:** Multiple tasks can be held in the background at once. However, if the user is running many background tasks at the same time, the system might begin destroying background activities in order to recover memory, causing the activity states to be lost. See the following section about [Activity state](#).

Because the activities in the back stack are never rearranged, if your application allows users to start a particular activity from more than one activity, a new instance of that activity is created and pushed onto the stack (rather than bringing any previous instance of the activity to the top). As such, one activity in your application might be instantiated multiple times (even from different tasks), as shown in figure 3. As such, if the user navigates backward using the *Back* button, each instance of the activity is revealed in the order they were opened (each with their own UI state). However, you can modify this behavior if you do not want an activity to be instantiated more than once. How to do so is discussed in the later section about [Managing Tasks](#).

To summarize the default behavior for activities and tasks:

- When Activity A starts Activity B, Activity A is stopped, but the system retains its state (such as scroll position and text entered into forms). If the user presses the *Back* button while in Activity B, Activity A resumes with its state restored.
- When the user leaves a task by pressing the *Home* button, the current activity is stopped and its task goes into the background. The system retains the state of every activity in the task. If the user later resumes the task by selecting the launcher icon that began the task, the task comes to the foreground and resumes the activity at the top of the stack.
- If the user presses the *Back* button, the current activity is popped from the stack and destroyed. The previous activity in the stack is resumed. When an activity is destroyed, the system *does not* retain the activity's state.
- Activities can be instantiated multiple times, even from other tasks.

## Navigation Design

For more about how app navigation works on Android, read Android Design's [Navigation](#) guide.

## Saving Activity State

As discussed above, the system's default behavior preserves the state of an activity when it is stopped. This way, when users navigate back to a previous activity, its user interface appears the way they left it. However, you can—and **should**—proactively retain the state of your activities using callback methods, in case the activity is destroyed and must be recreated.

When the system stops one of your activities (such as when a new activity starts or the task moves to the background), the system might destroy that activity completely if it needs to recover system memory. When this happens, information about the activity state is lost. If this happens, the system still knows that the activity has a place in the back stack, but when the activity is brought to the top of the stack the system must recreate it (rather than resume it). In order to avoid losing the user's work, you should proactively retain it by implementing the [onSaveInstanceState\(\)](#) callback methods in your activity.

For more information about how to save your activity state, see the [Activities](#) document.

# Managing Tasks

The way Android manages tasks and the back stack, as described above—by placing all activities started in succession in the same task and in a "last in, first out" stack—works great for most applications and you shouldn't have to worry about how your activities are associated with tasks or how they exist in the back stack. However, you might decide that you want to interrupt the normal behavior. Perhaps you want an activity in your application to begin a new task when it is started (instead of being placed within the current task); or, when you start an activity, you want to bring forward an existing instance of it (instead of creating a new instance on top of the back stack); or, you want your back stack to be cleared of all activities except for the root activity when the user leaves the task.

You can do these things and more, with attributes in the [`<activity>`](#) manifest element and with flags in the intent that you pass to [`startActivity\(\)`](#).

In this regard, the principal [`<activity>`](#) attributes you can use are:

- [taskAffinity](#)
- [launchMode](#)
- [allowTaskReparenting](#)
- [clearTaskOnLaunch](#)
- [alwaysRetainTaskState](#)
- [finishOnTaskLaunch](#)

And the principal intent flags you can use are:

- [FLAG\\_ACTIVITY\\_NEW\\_TASK](#)
- [FLAG\\_ACTIVITY\\_CLEAR\\_TOP](#)
- [FLAG\\_ACTIVITY\\_SINGLE\\_TOP](#)

In the following sections, you'll see how you can use these manifest attributes and intent flags to define how activities are associated with tasks and how they behave in the back stack.

**Caution:** Most applications should not interrupt the default behavior for activities and tasks. If you determine that it's necessary for your activity to modify the default behaviors, use caution and be sure to test the usability of the activity during launch and when navigating back to it from other activities and tasks with the *Back* button. Be sure to test for navigation behaviors that might conflict with the user's expected behavior.

## Defining launch modes

Launch modes allow you to define how a new instance of an activity is associated with the current task. You can define different launch modes in two ways:

- [Using the manifest file](#)

When you declare an activity in your manifest file, you can specify how the activity should associate with tasks when it starts.

- [Using Intent flags](#)

When you call [`startActivity\(\)`](#), you can include a flag in the [`Intent`](#) that declares how (or whether) the new activity should associate with the current task.

As such, if Activity A starts Activity B, Activity B can define in its manifest how it should associate with the current task (if at all) and Activity A can also request how Activity B should associate with current task. If both activities define how Activity B should associate with a task, then Activity A's request (as defined in the intent) is honored over Activity B's request (as defined in its manifest).

**Note:** Some launch modes available for the manifest file are not available as flags for an intent and, likewise, some launch modes available as flags for an intent cannot be defined in the manifest.

## Using the manifest file

When declaring an activity in your manifest file, you can specify how the activity should associate with a task using the [`<activity>`](#) element's [`launchMode`](#) attribute.

The [`launchMode`](#) attribute specifies an instruction on how the activity should be launched into a task. There are four different launch modes you can assign to the [`launchMode`](#) attribute:

### "standard" (the default mode)

Default. The system creates a new instance of the activity in the task from which it was started and routes the intent to it. The activity can be instantiated multiple times, each instance can belong to different tasks, and one task can have multiple instances.

### "singleTop"

If an instance of the activity already exists at the top of the current task, the system routes the intent to that instance through a call to its [`onNewIntent\(\)`](#) method, rather than creating a new instance of the activity. The activity can be instantiated multiple times, each instance can belong to different tasks, and one task can have multiple instances (but only if the activity at the top of the back stack is *not* an existing instance of the activity).

For example, suppose a task's back stack consists of root activity A with activities B, C, and D on top (the stack is A-B-C-D; D is on top). An intent arrives for an activity of type D. If D has the default "standard" launch mode, a new instance of the class is launched and the stack becomes A-B-C-D-D. However, if D's launch mode is "singleTop", the existing instance of D receives the intent through [`onNewIntent\(\)`](#), because it's at the top of the stack—the stack remains A-B-C-D. However, if an intent arrives for an activity of type B, then a new instance of B is added to the stack, even if its launch mode is "singleTop".

**Note:** When a new instance of an activity is created, the user can press the *Back* button to return to the previous activity. But when an existing instance of an activity handles a new intent, the user cannot press the *Back* button to return to the state of the activity before the new intent arrived in [`onNewIntent\(\)`](#).

### "singleTask"

The system creates a new task and instantiates the activity at the root of the new task. However, if an instance of the activity already exists in a separate task, the system routes the intent to the existing instance through a call to its [`onNewIntent\(\)`](#) method, rather than creating a new instance. Only one instance of the activity can exist at a time.

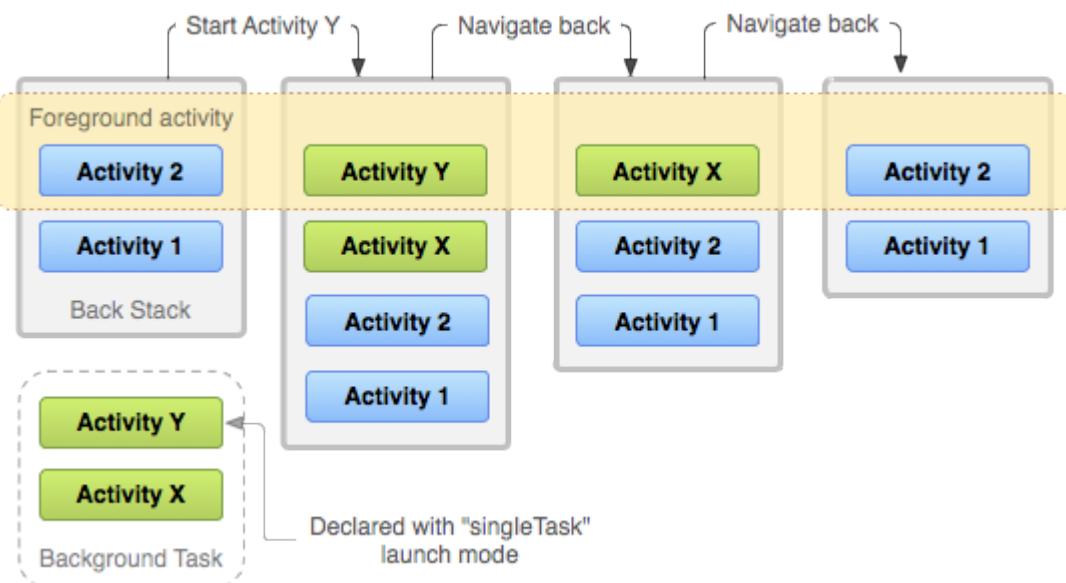
**Note:** Although the activity starts in a new task, the *Back* button still returns the user to the previous activity.

### "singleInstance".

Same as "singleTask", except that the system doesn't launch any other activities into the task holding the instance. The activity is always the single and only member of its task; any activities started by this one open in a separate task.

As another example, the Android Browser application declares that the web browser activity should always open in its own task—by specifying the `singleTask` launch mode in the [`<activity>`](#) element. This means that if your application issues an intent to open the Android Browser, its activity is *not* placed in the same task as your application. Instead, either a new task starts for the Browser or, if the Browser already has a task running in the background, that task is brought forward to handle the new intent.

Regardless of whether an activity starts in a new task or in the same task as the activity that started it, the *Back* button always takes the user to the previous activity. However, if you start an activity that specifies the `singleTask` launch mode, then if an instance of that activity exists in a background task, that whole task is brought to the foreground. At this point, the back stack now includes all activities from the task brought forward, at the top of the stack. Figure 4 illustrates this type of scenario.



**Figure 4.** A representation of how an activity with launch mode "singleTask" is added to the back stack. If the activity is already a part of a background task with its own back stack, then the entire back stack also comes forward, on top of the current task.

For more information about using launch modes in the manifest file, see the [`<activity>`](#) element documentation, where the `launchMode` attribute and the accepted values are discussed more.

**Note:** The behaviors that you specify for your activity with the `launchMode` attribute can be overridden by flags included with the intent that start your activity, as discussed in the next section.

## Using Intent flags

When starting an activity, you can modify the default association of an activity to its task by including flags in the intent that you deliver to [`startActivity\(\)`](#). The flags you can use to modify the default behavior are:

### [FLAG ACTIVITY NEW TASK](#)

Start the activity in a new task. If a task is already running for the activity you are now starting, that task is brought to the foreground with its last state restored and the activity receives the new intent in [`onNewIntent\(\)`](#).

This produces the same behavior as the "singleTask" `launchMode` value, discussed in the previous section.

## FLAG ACTIVITY SINGLE TOP

If the activity being started is the current activity (at the top of the back stack), then the existing instance receives a call to [onNewIntent \(\)](#), instead of creating a new instance of the activity.

This produces the same behavior as the "singleTop" [launchMode](#) value, discussed in the previous section.

## FLAG ACTIVITY CLEAR TOP

If the activity being started is already running in the current task, then instead of launching a new instance of that activity, all of the other activities on top of it are destroyed and this intent is delivered to the resumed instance of the activity (now on top), through [onNewIntent \(\)](#).

There is no value for the [launchMode](#) attribute that produces this behavior.

`FLAG_ACTIVITY_CLEAR_TOP` is most often used in conjunction with `FLAG_ACTIVITY_NEW_TASK`. When used together, these flags are a way of locating an existing activity in another task and putting it in a position where it can respond to the intent.

**Note:** If the launch mode of the designated activity is "standard", it too is removed from the stack and a new instance is launched in its place to handle the incoming intent. That's because a new instance is always created for a new intent when the launch mode is "standard".

## Handling affinities

The *affinity* indicates which task an activity prefers to belong to. By default, all the activities from the same application have an affinity for each other. So, by default, all activities in the same application prefer to be in the same task. However, you can modify the default affinity for an activity. Activities defined in different applications can share an affinity, or activities defined in the same application can be assigned different task affinities.

You can modify the affinity for any given activity with the [taskAffinity](#) attribute of the [`<activity>`](#) element.

The [taskAffinity](#) attribute takes a string value, which must be unique from the default package name declared in the [`<manifest>`](#) element, because the system uses that name to identify the default task affinity for the application.

The affinity comes into play in two circumstances:

- When the intent that launches an activity contains the [FLAG\\_ACTIVITY\\_NEW\\_TASK](#) flag.

A new activity is, by default, launched into the task of the activity that called [startActivity\(\)](#). It's pushed onto the same back stack as the caller. However, if the intent passed to [startActivity\(\)](#) contains the [FLAG\\_ACTIVITY\\_NEW\\_TASK](#) flag, the system looks for a different task to house the new activity. Often, it's a new task. However, it doesn't have to be. If there's already an existing task with the same affinity as the new activity, the activity is launched into that task. If not, it begins a new task.

If this flag causes an activity to begin a new task and the user presses the *Home* button to leave it, there must be some way for the user to navigate back to the task. Some entities (such as the notification manager) always start activities in an external task, never as part of their own, so they always put `FLAG_ACTIVITY_NEW_TASK` in the intents they pass to [startActivity\(\)](#). If you have an activity that can be invoked by an external entity that might use this flag, take care that the user has an independent way to get back to the task that's started, such as with a launcher icon (the root activity of the task has a [CATEGORY\\_LAUNCHER](#) intent filter; see the [Starting a task](#) section below).

- When an activity has its [allowTaskReparenting](#) attribute set to "true".

In this case, the activity can move from the task it starts to the task it has an affinity for, when that task comes to the foreground.

For example, suppose that an activity that reports weather conditions in selected cities is defined as part of a travel application. It has the same affinity as other activities in the same application (the default application affinity) and it allows re-parenting with this attribute. When one of your activities starts the weather reporter activity, it initially belongs to the same task as your activity. However, when the travel application's task comes to the foreground, the weather reporter activity is reassigned to that task and displayed within it.

**Tip:** If an .apk file contains more than one "application" from the user's point of view, you probably want to use the [taskAffinity](#) attribute to assign different affinities to the activities associated with each "application".

## Clearing the back stack

If the user leaves a task for a long time, the system clears the task of all activities except the root activity. When the user returns to the task again, only the root activity is restored. The system behaves this way, because, after an extended amount of time, users likely have abandoned what they were doing before and are returning to the task to begin something new.

There are some activity attributes that you can use to modify this behavior:

### [alwaysRetainTaskState](#)

If this attribute is set to "true" in the root activity of a task, the default behavior just described does not happen. The task retains all activities in its stack even after a long period.

### [clearTaskOnLaunch](#)

If this attribute is set to "true" in the root activity of a task, the stack is cleared down to the root activity whenever the user leaves the task and returns to it. In other words, it's the opposite of [alwaysRetainTaskState](#). The user always returns to the task in its initial state, even after leaving the task for only a moment.

### [finishOnTaskLaunch](#)

This attribute is like [clearTaskOnLaunch](#), but it operates on a single activity, not an entire task. It can also cause any activity to go away, including the root activity. When it's set to "true", the activity remains part of the task only for the current session. If the user leaves and then returns to the task, it is no longer present.

## Starting a task

You can set up an activity as the entry point for a task by giving it an intent filter with "android.intent.action.MAIN" as the specified action and "android.intent.category.LAUNCHER" as the specified category. For example:

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

An intent filter of this kind causes an icon and label for the activity to be displayed in the application launcher, giving users a way to launch the activity and to return to the task that it creates any time after it has been launched.

This second ability is important: Users must be able to leave a task and then come back to it later using this activity launcher. For this reason, the two [launch modes](#) that mark activities as always initiating a task, "singleTask" and ""singleInstance", should be used only when the activity has an [ACTION\\_MAIN](#) and a [CATEGORY\\_LAUNCHER](#) filter. Imagine, for example, what could happen if the filter is missing: An intent launches a "singleTask" activity, initiating a new task, and the user spends some time working in that task. The user then presses the *Home* button. The task is now sent to the background and is not visible. Now the user has no way to return to the task, because it is not represented in the application launcher.

For those cases where you don't want the user to be able to return to an activity, set the [`<activity>`](#) element's [`finishOnTaskLaunch`](#) to "true" (see [Clearing the stack](#)).

# Services

## Quickview

- A service can run in the background to perform work even while the user is in a different application
- A service can allow other components to bind to it, in order to interact with it and perform interprocess communication
- A service runs in the main thread of the application that hosts it, by default

## In this document

1. [The Basics](#)
  1. [Declaring a service in the manifest](#)
2. [Creating a Started Service](#)
  1. [Extending the IntentService class](#)
  2. [Extending the Service class](#)
  3. [Starting a service](#)
  4. [Stopping a service](#)
3. [Creating a Bound Service](#)
4. [Sending Notifications to the User](#)
5. [Running a Service in the Foreground](#)
6. [Managing the Lifecycle of a Service](#)
  1. [Implementing the lifecycle callbacks](#)

## Key classes

1. [Service](#)
2. [IntentService](#)

## Samples

1. [ServiceStartArguments](#)
2. [LocalService](#)

## See also

1. [Bound Services](#)

A [Service](#) is an application component that can perform long-running operations in the background and does not provide a user interface. Another application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

A service can essentially take two forms:

### Started

A service is "started" when an application component (such as an activity) starts it by calling [startService\(\)](#). Once started, a service can run in the background indefinitely, even if the component that started

it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should stop itself.

## Bound

A service is "bound" when an application component binds to it by calling [bindService\(\)](#). A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Although this documentation generally discusses these two types of services separately, your service can work both ways—it can be started (to run indefinitely) and also allow binding. It's simply a matter of whether you implement a couple callback methods: [onStartCommand\(\)](#) to allow components to start it and [onBind\(\)](#) to allow binding.

Regardless of whether your application is started, bound, or both, any application component can use the service (even from a separate application), in the same way that any component can use an activity—by starting it with an [Intent](#). However, you can declare the service as private, in the manifest file, and block access from other applications. This is discussed more in the section about [Declaring the service in the manifest](#).

**Caution:** A service runs in the main thread of its hosting process—the service does **not** create its own thread and does **not** run in a separate process (unless you specify otherwise). This means that, if your service is going to do any CPU intensive work or blocking operations (such as MP3 playback or networking), you should create a new thread within the service to do that work. By using a separate thread, you will reduce the risk of Application Not Responding (ANR) errors and the application's main thread can remain dedicated to user interaction with your activities.

# The Basics

## Should you use a service or a thread?

A service is simply a component that can run in the background even when the user is not interacting with your application. Thus, you should create a service only if that is what you need.

If you need to perform work outside your main thread, but only while the user is interacting with your application, then you should probably instead create a new thread and not a service. For example, if you want to play some music, but only while your activity is running, you might create a thread in [onCreate\(\)](#), start running it in [onStart\(\)](#), then stop it in [onStop\(\)](#). Also consider using [AsyncTask](#) or [HandlerThread](#), instead of the traditional [Thread](#) class. See the [Processes and Threading](#) document for more information about threads.

Remember that if you do use a service, it still runs in your application's main thread by default, so you should still create a new thread within the service if it performs intensive or blocking operations.

To create a service, you must create a subclass of [Service](#) (or one of its existing subclasses). In your implementation, you need to override some callback methods that handle key aspects of the service lifecycle and provide a mechanism for components to bind to the service, if appropriate. The most important callback methods you should override are:

### [onStartCommand\(\)](#)

The system calls this method when another component, such as an activity, requests that the service be started, by calling [startService\(\)](#). Once this method executes, the service is started and can run in

the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is done, by calling [stopSelf\(\)](#) or [stopService\(\)](#). (If you only want to provide binding, you don't need to implement this method.)

#### [onBind\(\)](#)

The system calls this method when another component wants to bind with the service (such as to perform RPC), by calling [bindService\(\)](#). In your implementation of this method, you must provide an interface that clients use to communicate with the service, by returning an [IBinder](#). You must always implement this method, but if you don't want to allow binding, then you should return null.

#### [onCreate\(\)](#)

The system calls this method when the service is first created, to perform one-time setup procedures (before it calls either [onStartCommand\(\)](#) or [onBind\(\)](#)). If the service is already running, this method is not called.

#### [onDestroy\(\)](#)

The system calls this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc. This is the last call the service receives.

If a component starts the service by calling [startService\(\)](#) (which results in a call to [onStartCommand\(\)](#)), then the service remains running until it stops itself with [stopSelf\(\)](#) or another component stops it by calling [stopService\(\)](#).

If a component calls [bindService\(\)](#) to create the service (and [onStartCommand\(\)](#) is *not* called), then the service runs only as long as the component is bound to it. Once the service is unbound from all clients, the system destroys it.

The Android system will force-stop a service only when memory is low and it must recover system resources for the activity that has user focus. If the service is bound to an activity that has user focus, then it's less likely to be killed, and if the service is declared to [run in the foreground](#) (discussed later), then it will almost never be killed. Otherwise, if the service was started and is long-running, then the system will lower its position in the list of background tasks over time and the service will become highly susceptible to killing—if your service is started, then you must design it to gracefully handle restarts by the system. If the system kills your service, it restarts it as soon as resources become available again (though this also depends on the value you return from [onStartCommand\(\)](#), as discussed later). For more information about when the system might destroy a service, see the [Processes and Threading](#) document.

In the following sections, you'll see how you can create each type of service and how to use it from other application components.

## Declaring a service in the manifest

Like activities (and other components), you must declare all services in your application's manifest file.

To declare your service, add a [`<service>`](#) element as a child of the [`<application>`](#) element. For example:

```
<manifest ... >
  ...
  <application ... >
    <service android:name=".ExampleService" />
  ...

```

```
</application>  
</manifest>
```

There are other attributes you can include in the [`<service>`](#) element to define properties such as permissions required to start the service and the process in which the service should run. The [`android:name`](#) attribute is the only required attribute—it specifies the class name of the service. Once you publish your application, you should not change this name, because if you do, you might break some functionality where explicit intents are used to reference your service (read the blog post, [Things That Cannot Change](#)).

See the [`<service>`](#) element reference for more information about declaring your service in the manifest.

Just like an activity, a service can define intent filters that allow other components to invoke the service using implicit intents. By declaring intent filters, components from any application installed on the user's device can potentially start your service if your service declares an intent filter that matches the intent another application passes to [`startService\(\)`](#).

If you plan on using your service only locally (other applications do not use it), then you don't need to (and should not) supply any intent filters. Without any intent filters, you must start the service using an intent that explicitly names the service class. More information about [starting a service](#) is discussed below.

Additionally, you can ensure that your service is private to your application only if you include the [`an-  
droid:exported`](#) attribute and set it to "false". This is effective even if your service supplies intent filters.

For more information about creating intent filters for your service, see the [Intents and Intent Filters](#) document.

## Creating a Started Service

A started service is one that another component starts by calling [`startService\(\)`](#), resulting in a call to the service's [`onStartCommand\(\)`](#) method.

When a service is started, it has a lifecycle that's independent of the component that started it and the service can run in the background indefinitely, even if the component that started it is destroyed. As such, the service should stop itself when its job is done by calling [`stopSelf\(\)`](#), or another component can stop it by calling [`stopService\(\)`](#).

An application component such as an activity can start the service by calling [`startService\(\)`](#) and passing an [`Intent`](#) that specifies the service and includes any data for the service to use. The service receives this [`Intent`](#) in the [`onStartCommand\(\)`](#) method.

For instance, suppose an activity needs to save some data to an online database. The activity can start a companion service and deliver it the data to save by passing an intent to [`startService\(\)`](#). The service receives the intent in [`onStartCommand\(\)`](#), connects to the Internet and performs the database transaction. When the transaction is done, the service stops itself and it is destroyed.

**Caution:** A service runs in the same process as the application in which it is declared and in the main thread of that application, by default. So, if your service performs intensive or blocking operations while the user interacts with an activity from the same application, the service will slow down activity performance. To avoid impacting application performance, you should start a new thread inside the service.

Traditionally, there are two classes you can extend to create a started service:

## Service

This is the base class for all services. When you extend this class, it's important that you create a new thread in which to do all the service's work, because the service uses your application's main thread, by default, which could slow the performance of any activity your application is running.

## IntentService

This is a subclass of [Service](#) that uses a worker thread to handle all start requests, one at a time. This is the best option if you don't require that your service handle multiple requests simultaneously. All you need to do is implement [onHandleIntent\(\)](#), which receives the intent for each start request so you can do the background work.

The following sections describe how you can implement your service using either one for these classes.

## Extending the IntentService class

Because most started services don't need to handle multiple requests simultaneously (which can actually be a dangerous multi-threading scenario), it's probably best if you implement your service using the [IntentService](#) class.

The [IntentService](#) does the following:

- Creates a default worker thread that executes all intents delivered to [onStartCommand\(\)](#) separate from your application's main thread.
- Creates a work queue that passes one intent at a time to your [onHandleIntent\(\)](#) implementation, so you never have to worry about multi-threading.
- Stops the service after all start requests have been handled, so you never have to call [stopSelf\(\)](#).
- Provides default implementation of  [onBind\(\)](#) that returns null.
- Provides a default implementation of [onStartCommand\(\)](#) that sends the intent to the work queue and then to your [onHandleIntent\(\)](#) implementation.

All this adds up to the fact that all you need to do is implement [onHandleIntent\(\)](#) to do the work provided by the client. (Though, you also need to provide a small constructor for the service.)

Here's an example implementation of [IntentService](#):

```
public class HelloIntentService extends IntentService {  
  
    /**  
     * A constructor is required, and must call the super IntentService\(String\)  
     * constructor with a name for the worker thread.  
     */  
    public HelloIntentService() {  
        super("HelloIntentService");  
    }  
  
    /**  
     * The IntentService calls this method from the default worker thread with  
     * the intent that started the service. When this method returns, IntentServic  
     * stops the service, as appropriate.  
     */  
    @Override  
    protected void onHandleIntent(Intent intent) {  
        // Normally we would do some work here, like download a file.  
        // For our sample, we just sleep for 5 seconds.  
    }  
}
```

```

        long endTime = System.currentTimeMillis() + 5*1000;
        while (System.currentTimeMillis() < endTime) {
            synchronized (this) {
                try {
                    wait(endTime - System.currentTimeMillis());
                } catch (Exception e) {
                }
            }
        }
    }
}

```

That's all you need: a constructor and an implementation of [onHandleIntent\(\)](#).

If you decide to also override other callback methods, such as [onCreate\(\)](#), [onStartCommand\(\)](#), or [onDestroy\(\)](#), be sure to call the super implementation, so that the [IntentService](#) can properly handle the life of the worker thread.

For example, [onStartCommand\(\)](#) must return the default implementation (which is how the intent gets delivered to [onHandleIntent\(\)](#)):

```

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();
    return super.onStartCommand(intent, flags, startId);
}

```

Besides [onHandleIntent\(\)](#), the only method from which you don't need to call the super class is [onBind\(\)](#) (but you only need to implement that if your service allows binding).

In the next section, you'll see how the same kind of service is implemented when extending the base [Service](#) class, which is a lot more code, but which might be appropriate if you need to handle simultaneous start requests.

## Extending the Service class

As you saw in the previous section, using [IntentService](#) makes your implementation of a started service very simple. If, however, you require your service to perform multi-threading (instead of processing start requests through a work queue), then you can extend the [Service](#) class to handle each intent.

For comparison, the following example code is an implementation of the [Service](#) class that performs the exact same work as the example above using [IntentService](#). That is, for each start request, it uses a worker thread to perform the job and processes only one request at a time.

```

public class HelloService extends Service {
    private Looper mServiceLooper;
    private ServiceHandler mServiceHandler;

    // Handler that receives messages from the thread
    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
    }
    @Override

```

```
public void handleMessage(Message msg) {
    // Normally we would do some work here, like download a file.
    // For our sample, we just sleep for 5 seconds.
    long endTime = System.currentTimeMillis() + 5*1000;
    while (System.currentTimeMillis() < endTime) {
        synchronized (this) {
            try {
                wait(endTime - System.currentTimeMillis());
            } catch (Exception e) {
            }
        }
    }
    // Stop the service using the startId, so that we don't stop
    // the service in the middle of handling another job
    stopSelf(msg.arg1);
}

@Override
public void onCreate() {
    // Start up the thread running the service. Note that we create a
    // separate thread because the service normally runs in the process's
    // main thread, which we don't want to block. We also make it
    // background priority so CPU-intensive work will not disrupt our UI.
    HandlerThread thread = new HandlerThread("ServiceStartArguments",
        Process.THREAD_PRIORITY_BACKGROUND);
    thread.start();

    // Get the HandlerThread's Looper and use it for our Handler
    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

    // For each start request, send a message to start a job and deliver the
    // start ID so we know which request we're stopping when we finish the job
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    mServiceHandler.sendMessage(msg);

    // If we get killed, after returning from here, restart
    return START_STICKY;
}

@Override
public IBinder onBind(Intent intent) {
    // We don't provide binding, so return null
    return null;
}

@Override
```

```
public void onDestroy() {
    Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
}
}
```

As you can see, it's a lot more work than using [IntentService](#).

However, because you handle each call to [onStartCommand\(\)](#) yourself, you can perform multiple requests simultaneously. That's not what this example does, but if that's what you want, then you can create a new thread for each request and run them right away (instead of waiting for the previous request to finish).

Notice that the [onStartCommand\(\)](#) method must return an integer. The integer is a value that describes how the system should continue the service in the event that the system kills it (as discussed above, the default implementation for [IntentService](#) handles this for you, though you are able to modify it). The return value from [onStartCommand\(\)](#) must be one of the following constants:

#### **START NOT STICKY**

If the system kills the service after [onStartCommand\(\)](#) returns, *do not* recreate the service, unless there are pending intents to deliver. This is the safest option to avoid running your service when not necessary and when your application can simply restart any unfinished jobs.

#### **START STICKY**

If the system kills the service after [onStartCommand\(\)](#) returns, recreate the service and call [onStartCommand\(\)](#), but *do not* redeliver the last intent. Instead, the system calls [onStartCommand\(\)](#) with a null intent, unless there were pending intents to start the service, in which case, those intents are delivered. This is suitable for media players (or similar services) that are not executing commands, but running indefinitely and waiting for a job.

#### **START REDELIVER INTENT**

If the system kills the service after [onStartCommand\(\)](#) returns, recreate the service and call [onStartCommand\(\)](#) with the last intent that was delivered to the service. Any pending intents are delivered in turn. This is suitable for services that are actively performing a job that should be immediately resumed, such as downloading a file.

For more details about these return values, see the linked reference documentation for each constant.

## Starting a Service

You can start a service from an activity or other application component by passing an [Intent](#) (specifying the service to start) to [startService\(\)](#). The Android system calls the service's [onStartCommand\(\)](#) method and passes it the [Intent](#). (You should never call [onStartCommand\(\)](#) directly.)

For example, an activity can start the example service in the previous section (`HelloService`) using an explicit intent with [startService\(\)](#):

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

The [startService\(\)](#) method returns immediately and the Android system calls the service's [onStartCommand\(\)](#) method. If the service is not already running, the system first calls [onCreate\(\)](#), then calls [onStartCommand\(\)](#).

If the service does not also provide binding, the intent delivered with [startService\(\)](#) is the only mode of communication between the application component and the service. However, if you want the service to send a

result back, then the client that starts the service can create a [PendingIntent](#) for a broadcast (with [getBroadcast\(\)](#)) and deliver it to the service in the [Intent](#) that starts the service. The service can then use the broadcast to deliver a result.

Multiple requests to start the service result in multiple corresponding calls to the service's [onStartCommand\(\)](#). However, only one request to stop the service (with [stopSelf\(\)](#) or [stopService\(\)](#)) is required to stop it.

## Stopping a service

A started service must manage its own lifecycle. That is, the system does not stop or destroy the service unless it must recover system memory and the service continues to run after [onStartCommand\(\)](#) returns. So, the service must stop itself by calling [stopSelf\(\)](#) or another component can stop it by calling [stopService\(\)](#).

Once requested to stop with [stopSelf\(\)](#) or [stopService\(\)](#), the system destroys the service as soon as possible.

However, if your service handles multiple requests to [onStartCommand\(\)](#) concurrently, then you shouldn't stop the service when you're done processing a start request, because you might have since received a new start request (stopping at the end of the first request would terminate the second one). To avoid this problem, you can use [stopSelf\(int\)](#) to ensure that your request to stop the service is always based on the most recent start request. That is, when you call [stopSelf\(int\)](#), you pass the ID of the start request (the `startId` delivered to [onStartCommand\(\)](#)) to which your stop request corresponds. Then if the service received a new start request before you were able to call [stopSelf\(int\)](#), then the ID will not match and the service will not stop.

**Caution:** It's important that your application stops its services when it's done working, to avoid wasting system resources and consuming battery power. If necessary, other components can stop the service by calling [stopService\(\)](#). Even if you enable binding for the service, you must always stop the service yourself if it ever received a call to [onStartCommand\(\)](#).

For more information about the lifecycle of a service, see the section below about [Managing the Lifecycle of a Service](#).

## Creating a Bound Service

A bound service is one that allows application components to bind to it by calling [bindService\(\)](#) in order to create a long-standing connection (and generally does not allow components to *start* it by calling [startService\(\)](#)).

You should create a bound service when you want to interact with the service from activities and other components in your application or to expose some of your application's functionality to other applications, through interprocess communication (IPC).

To create a bound service, you must implement the [onBind\(\)](#) callback method to return an [IBinder](#) that defines the interface for communication with the service. Other application components can then call [bindService\(\)](#) to retrieve the interface and begin calling methods on the service. The service lives only to serve the application component that is bound to it, so when there are no components bound to the service, the system destroys it (you do *not* need to stop a bound service in the way you must when the service is started through [onStartCommand\(\)](#)).

To create a bound service, the first thing you must do is define the interface that specifies how a client can communicate with the service. This interface between the service and a client must be an implementation of [IBinder](#) and is what your service must return from the [onBind\(\)](#) callback method. Once the client receives the [IBinder](#), it can begin interacting with the service through that interface.

Multiple clients can bind to the service at once. When a client is done interacting with the service, it calls [unbindService\(\)](#) to unbind. Once there are no clients bound to the service, the system destroys the service.

There are multiple ways to implement a bound service and the implementation is more complicated than a started service, so the bound service discussion appears in a separate document about [Bound Services](#).

## Sending Notifications to the User

Once running, a service can notify the user of events using [Toast Notifications](#) or [Status Bar Notifications](#).

A toast notification is a message that appears on the surface of the current window for a moment then disappears, while a status bar notification provides an icon in the status bar with a message, which the user can select in order to take an action (such as start an activity).

Usually, a status bar notification is the best technique when some background work has completed (such as a file completed downloading) and the user can now act on it. When the user selects the notification from the expanded view, the notification can start an activity (such as to view the downloaded file).

See the [Toast Notifications](#) or [Status Bar Notifications](#) developer guides for more information.

## Running a Service in the Foreground

A foreground service is a service that's considered to be something the user is actively aware of and thus not a candidate for the system to kill when low on memory. A foreground service must provide a notification for the status bar, which is placed under the "Ongoing" heading, which means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground.

For example, a music player that plays music from a service should be set to run in the foreground, because the user is explicitly aware of its operation. The notification in the status bar might indicate the current song and allow the user to launch an activity to interact with the music player.

To request that your service run in the foreground, call [startForeground\(\)](#). This method takes two parameters: an integer that uniquely identifies the notification and the [Notification](#) for the status bar. For example:

```
Notification notification = new Notification(R.drawable.icon, getText(R.string.
    System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIn
notification.setLatestEventInfo(this, getText(R.string.notification_title),
    getText(R.string.notification_message), pendingIntent);
startForeground(ONGOING_NOTIFICATION_ID, notification);
```

**Caution:** The integer ID you give to [startForeground\(\)](#) must not be 0.

To remove the service from the foreground, call [stopForeground\(\)](#). This method takes a boolean, indicating whether to remove the status bar notification as well. This method does *not* stop the service. However, if you stop the service while it's still running in the foreground, then the notification is also removed.

For more information about notifications, see [Creating Status Bar Notifications](#).

## Managing the Lifecycle of a Service

The lifecycle of a service is much simpler than that of an activity. However, it's even more important that you pay close attention to how your service is created and destroyed, because a service can run in the background without the user being aware.

The service lifecycle—from when it's created to when it's destroyed—can follow two different paths:

- A started service

The service is created when another component calls [startService\(\)](#). The service then runs indefinitely and must stop itself by calling [stopSelf\(\)](#). Another component can also stop the service by calling [stopService\(\)](#). When the service is stopped, the system destroys it..

- A bound service

The service is created when another component (a client) calls [bindService\(\)](#). The client then communicates with the service through an [IBinder](#) interface. The client can close the connection by calling [unbindService\(\)](#). Multiple clients can bind to the same service and when all of them unbind, the system destroys the service. (The service does *not* need to stop itself.)

These two paths are not entirely separate. That is, you can bind to a service that was already started with [startService\(\)](#). For example, a background music service could be started by calling [startService\(\)](#) with an [Intent](#) that identifies the music to play. Later, possibly when the user wants to exercise some control over the player or get information about the current song, an activity can bind to the service by calling [bindService\(\)](#). In cases like this, [stopService\(\)](#) or [stopSelf\(\)](#) does not actually stop the service until all clients unbind.

## Implementing the lifecycle callbacks

Like an activity, a service has lifecycle callback methods that you can implement to monitor changes in the service's state and perform work at the appropriate times. The following skeleton service demonstrates each of the lifecycle methods:

```
public class ExampleService extends Service {
    int mStartMode;          // indicates how to behave if the service is killed
    IBinder mBinder;         // interface for clients that bind
    boolean mAllowRebind;    // indicates whether onRebind should be used

    @Override
    public void onCreate() {
        // The service is being created
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // The service is starting, due to a call to startService()
        return mStartMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService()
    }
}
```

```

        return mBinder;
    }

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

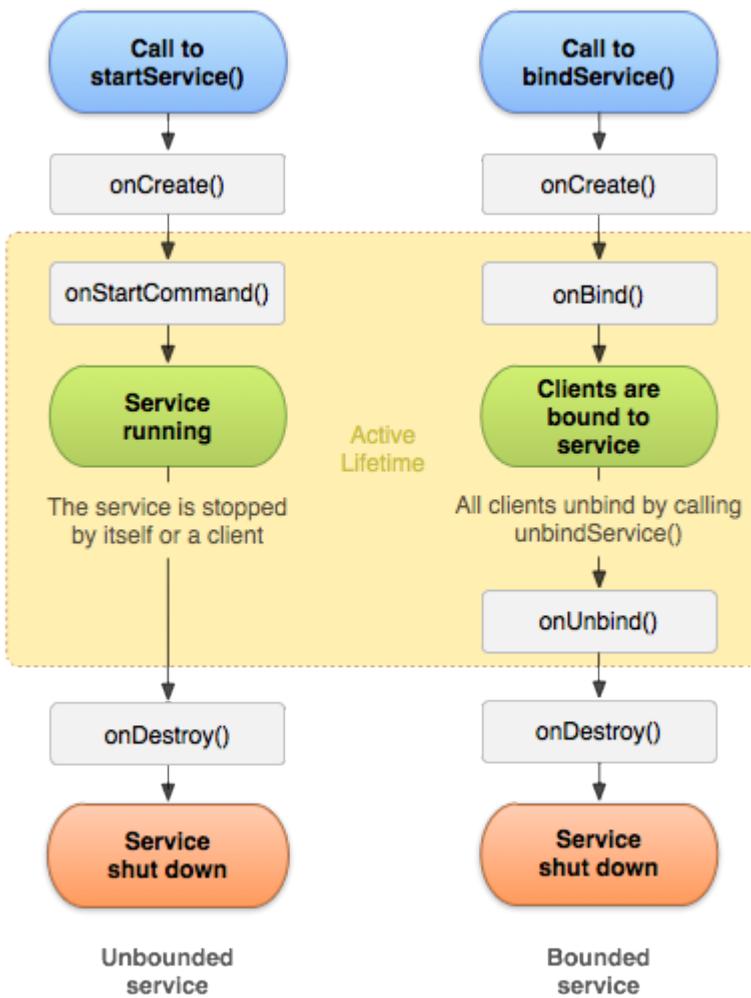
@Override
public void onRebind(Intent intent) {
    // A client is binding to the service with bindService(),
    // after onUnbind() has already been called
}

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

}

```

**Note:** Unlike the activity lifecycle callback methods, you are *not* required to call the superclass implementation of these callback methods.



**Figure 2.** The service lifecycle. The diagram on the left shows the lifecycle when the service is created with `startService()` and the diagram on the right shows the lifecycle when the service is created with `bindService()`.

By implementing these methods, you can monitor two nested loops of the service's lifecycle:

- The **entire lifetime** of a service happens between the time `onCreate()` is called and the time `onDestroy()` returns. Like an activity, a service does its initial setup in `onCreate()` and releases all remaining resources in `onDestroy()`. For example, a music playback service could create the thread where the music will be played in `onCreate()`, then stop the thread in `onDestroy()`.

The `onCreate()` and `onDestroy()` methods are called for all services, whether they're created by `startService()` or `bindService()`.

- The **active lifetime** of a service begins with a call to either `onStartCommand()` or `onBind()`. Each method is handed the `Intent` that was passed to either `startService()` or `bindService()`, respectively.

If the service is started, the active lifetime ends the same time that the entire lifetime ends (the service is still active even after `onStartCommand()` returns). If the service is bound, the active lifetime ends when `onUnbind()` returns.

**Note:** Although a started service is stopped by a call to either `stopSelf()` or `stopService()`, there is not a respective callback for the service (there's no `onStop()` callback). So, unless the service is bound to a client, the system destroys it when the service is stopped—`onDestroy()` is the only callback received.

Figure 2 illustrates the typical callback methods for a service. Although the figure separates services that are created by `startService()` from those created by `bindService()`, keep in mind that any service, no matter how it's started, can potentially allow clients to bind to it. So, a service that was initially started with `onStartCommand()` (by a client calling `startService()`) can still receive a call to `onBind()` (when a client calls `bindService()`).

For more information about creating a service that provides binding, see the [Bound Services](#) document, which includes more information about the `onRebind()` callback method in the section about [Managing the Lifecycle of a Bound Service](#).

# Bound Services

## Quickview

- A bound service allows other components to bind to it, in order to interact with it and perform interprocess communication
- A bound service is destroyed once all clients unbind, unless the service was also started

## In this document

1. [The Basics](#)
2. [Creating a Bound Service](#)
  1. [Extending the Binder class](#)
  2. [Using a Messenger](#)
3. [Binding to a Service](#)
4. [Managing the Lifecycle of a Bound Service](#)

## Key classes

1. [Service](#)
2. [ServiceConnection](#)
3. [IBinder](#)

## Samples

1. [RemoteService](#)
2. [LocalService](#)

## See also

1. [Services](#)

A bound service is the server in a client-server interface. A bound service allows components (such as activities) to bind to the service, send requests, receive responses, and even perform interprocess communication (IPC). A bound service typically lives only while it serves another application component and does not run in the background indefinitely.

This document shows you how to create a bound service, including how to bind to the service from other application components. However, you should also refer to the [Services](#) document for additional information about services in general, such as how to deliver notifications from a service, set the service to run in the foreground, and more.

## The Basics

A bound service is an implementation of the [Service](#) class that allows other applications to bind to it and interact with it. To provide binding for a service, you must implement the [onBind\(\)](#) callback method. This method returns an [IBinder](#) object that defines the programming interface that clients can use to interact with the service.

## Binding to a Started Service

As discussed in the [Services](#) document, you can create a service that is both started and bound. That is, the service can be started by calling [`startService\(\)`](#), which allows the service to run indefinitely, and also allow a client to bind to the service by calling [`bindService\(\)`](#).

If you do allow your service to be started and bound, then when the service has been started, the system does *not* destroy the service when all clients unbind. Instead, you must explicitly stop the service, by calling [`stopSelf\(\)`](#) or [`stopService\(\)`](#).

Although you should usually implement either [`onBind\(\)`](#) or [`onStartCommand\(\)`](#), it's sometimes necessary to implement both. For example, a music player might find it useful to allow its service to run indefinitely and also provide binding. This way, an activity can start the service to play some music and the music continues to play even if the user leaves the application. Then, when the user returns to the application, the activity can bind to the service to regain control of playback.

Be sure to read the section about [Managing the Lifecycle of a Bound Service](#), for more information about the service lifecycle when adding binding to a started service.

A client can bind to the service by calling [`bindService\(\)`](#). When it does, it must provide an implementation of [`ServiceConnection`](#), which monitors the connection with the service. The [`bindService\(\)`](#) method returns immediately without a value, but when the Android system creates the connection between the client and service, it calls [`onServiceConnected\(\)`](#) on the [`ServiceConnection`](#), to deliver the [`IBinder`](#) that the client can use to communicate with the service.

Multiple clients can connect to the service at once. However, the system calls your service's [`onBind\(\)`](#) method to retrieve the [`IBinder`](#) only when the first client binds. The system then delivers the same [`IBinder`](#) to any additional clients that bind, without calling [`onBind\(\)`](#) again.

When the last client unbinds from the service, the system destroys the service (unless the service was also started by [`startService\(\)`](#)).

When you implement your bound service, the most important part is defining the interface that your [`onBind\(\)`](#) callback method returns. There are a few different ways you can define your service's [`IBinder`](#) interface and the following section discusses each technique.

## Creating a Bound Service

When creating a service that provides binding, you must provide an [`IBinder`](#) that provides the programming interface that clients can use to interact with the service. There are three ways you can define the interface:

### [Extending the Binder class](#)

If your service is private to your own application and runs in the same process as the client (which is common), you should create your interface by extending the [`Binder`](#) class and returning an instance of it from [`onBind\(\)`](#). The client receives the [`Binder`](#) and can use it to directly access public methods available in either the [`Binder`](#) implementation or even the [`Service`](#).

This is the preferred technique when your service is merely a background worker for your own application. The only reason you would not create your interface this way is because your service is used by other applications or across separate processes.

## Using a Messenger

If you need your interface to work across different processes, you can create an interface for the service with a [Messenger](#). In this manner, the service defines a [Handler](#) that responds to different types of [Message](#) objects. This [Handler](#) is the basis for a [Messenger](#) that can then share an [IBinder](#) with the client, allowing the client to send commands to the service using [Message](#) objects. Additionally, the client can define a [Messenger](#) of its own so the service can send messages back.

This is the simplest way to perform interprocess communication (IPC), because the [Messenger](#) queues all requests into a single thread so that you don't have to design your service to be thread-safe.

## Using AIDL

AIDL (Android Interface Definition Language) performs all the work to decompose objects into primitives that the operating system can understand and marshall them across processes to perform IPC. The previous technique, using a [Messenger](#), is actually based on AIDL as its underlying structure. As mentioned above, the [Messenger](#) creates a queue of all the client requests in a single thread, so the service receives requests one at a time. If, however, you want your service to handle multiple requests simultaneously, then you can use AIDL directly. In this case, your service must be capable of multi-threading and be built thread-safe.

To use AIDL directly, you must create an `.aidl` file that defines the programming interface. The Android SDK tools use this file to generate an abstract class that implements the interface and handles IPC, which you can then extend within your service.

**Note:** Most applications **should not** use AIDL to create a bound service, because it may require multithreading capabilities and can result in a more complicated implementation. As such, AIDL is not suitable for most applications and this document does not discuss how to use it for your service. If you're certain that you need to use AIDL directly, see the [AIDL](#) document.

## Extending the Binder class

If your service is used only by the local application and does not need to work across processes, then you can implement your own [Binder](#) class that provides your client direct access to public methods in the service.

**Note:** This works only if the client and service are in the same application and process, which is most common. For example, this would work well for a music application that needs to bind an activity to its own service that's playing music in the background.

Here's how to set it up:

1. In your service, create an instance of [Binder](#) that either:
  - contains public methods that the client can call
  - returns the current [Service](#) instance, which has public methods the client can call
  - or, returns an instance of another class hosted by the service with public methods the client can call
2. Return this instance of [Binder](#) from the [onBind\(\)](#) callback method.
3. In the client, receive the [Binder](#) from the [onServiceConnected\(\)](#) callback method and make calls to the bound service using the methods provided.

**Note:** The reason the service and client must be in the same application is so the client can cast the returned object and properly call its APIs. The service and client must also be in the same process, because this technique does not perform any marshalling across processes.

For example, here's a service that provides clients access to methods in the service through a [Binder](#) implementation:

```

public class LocalService extends Service {
    // Binder given to clients
    private final IBinder mBinder = new LocalBinder();
    // Random number generator
    private final Random mGenerator = new Random();

    /**
     * Class used for the client Binder. Because we know this service always
     * runs in the same process as its clients, we don't need to deal with IPC.
     */
    public class LocalBinder extends Binder {
        LocalService getService() {
            // Return this instance of LocalService so clients can call public
            return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    /** method for clients */
    public int getRandomNumber() {
        return mGenerator.nextInt(100);
    }
}

```

The `LocalBinder` provides the `getService()` method for clients to retrieve the current instance of `LocalService`. This allows clients to call public methods in the service. For example, clients can call `getRandomNumber()` from the service.

Here's an activity that binds to `LocalService` and calls `getRandomNumber()` when a button is clicked:

```

public class BindingActivity extends Activity {
    LocalService mService;
    boolean mBound = false;

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

    @Override
    protected void onStart() {
        super.onStart();
        // Bind to LocalService
        Intent intent = new Intent(this, LocalService.class);
        bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
    }

    @Override
    protected void onStop() {

```

```

super.onStop();
// Unbind from the service
if (mBound) {
    unbindService(mConnection);
    mBound = false;
}
}

/** Called when a button is clicked (the button in the layout file attaches
 * this method with the android:onClick attribute) */
public void onButtonClick(View v) {
    if (mBound) {
        // Call a method from the LocalService.
        // However, if this call were something that might hang, then this
        // occur in a separate thread to avoid slowing down the activity per-
        int num = mService.getRandomNumber();
        Toast.makeText(this, "number: " + num, Toast.LENGTH_SHORT).show();
    }
}

/** Defines callbacks for service binding, passed to bindService() */
private ServiceConnection mConnection = new ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName className,
                                   IBinder service) {
        // We've bound to LocalService, cast the IBinder and get LocalServ
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;
    }

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        mBound = false;
    }
};

}
}

```

The above sample shows how the client binds to the service using an implementation of [ServiceConnection](#) and the [onServiceConnected\(\)](#) callback. The next section provides more information about this process of binding to the service.

**Note:** The example above doesn't explicitly unbind from the service, but all clients should unbind at an appropriate time (such as when the activity pauses).

For more sample code, see the [LocalService.java](#) class and the [LocalServiceActivities.java](#) class in [ApiDemos](#).

# Using a Messenger

## Compared to AIDL

When you need to perform IPC, using a [Messenger](#) for your interface is simpler than implementing it with AIDL, because [Messenger](#) queues all calls to the service, whereas, a pure AIDL interface sends simultaneous requests to the service, which must then handle multi-threading.

For most applications, the service doesn't need to perform multi-threading, so using a [Messenger](#) allows the service to handle one call at a time. If it's important that your service be multi-threaded, then you should use [AIDL](#) to define your interface.

If you need your service to communicate with remote processes, then you can use a [Messenger](#) to provide the interface for your service. This technique allows you to perform interprocess communication (IPC) without the need to use AIDL.

Here's a summary of how to use a [Messenger](#):

- The service implements a [Handler](#) that receives a callback for each call from a client.
- The [Handler](#) is used to create a [Messenger](#) object (which is a reference to the [Handler](#)).
- The [Messenger](#) creates an [IBinder](#) that the service returns to clients from [onBind\(\)](#).
- Clients use the [IBinder](#) to instantiate the [Messenger](#) (that references the service's [Handler](#)), which the client uses to send [Message](#) objects to the service.
- The service receives each [Message](#) in its [Handler](#)—specifically, in the [handleMessage\(\)](#) method.

In this way, there are no "methods" for the client to call on the service. Instead, the client delivers "messages" ([Message](#) objects) that the service receives in its [Handler](#).

Here's a simple example service that uses a [Messenger](#) interface:

```
public class MessengerService extends Service {  
    /** Command to the service to display a message */  
    static final int MSG_SAY_HELLO = 1;  
  
    /**  
     * Handler of incoming messages from clients.  
     */  
    class IncomingHandler extends Handler {  
        @Override  
        public void handleMessage(Message msg) {  
            switch (msg.what) {  
                case MSG_SAY_HELLO:  
                    Toast.makeText(getApplicationContext(), "hello!", Toast.LENGTH_SHORT).show();  
                    break;  
                default:  
                    super.handleMessage(msg);  
            }  
        }  
    }  
  
    /**  
     * Target we publish for clients to send messages to IncomingHandler.  
     */
```

```

*/
final Messenger mMessenger = new Messenger(new IncomingHandler());

/**
 * When binding to the service, we return an interface to our messenger
 * for sending messages to the service.
 */
@Override
public IBinder onBind(Intent intent) {
    Toast.makeText(getApplicationContext(), "binding", Toast.LENGTH_SHORT).show();
    return mMessenger.getBinder();
}
}

```

Notice that the [handleMessage\(\)](#) method in the [Handler](#) is where the service receives the incoming [Message](#) and decides what to do, based on the [what](#) member.

All that a client needs to do is create a [Messenger](#) based on the [IBinder](#) returned by the service and send a message using [send\(\)](#). For example, here's a simple activity that binds to the service and delivers the `MSG_SAY_HELLO` message to the service:

```

public class ActivityMessenger extends Activity {
    /** Messenger for communicating with the service. */
    Messenger mService = null;

    /** Flag indicating whether we have called bind on the service. */
    boolean mBound;

    /**
     * Class for interacting with the main interface of the service.
     */
    private ServiceConnection mConnection = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {
            // This is called when the connection with the service has been
            // established, giving us the object we can use to
            // interact with the service. We are communicating with the
            // service using a Messenger, so here we get a client-side
            // representation of that from the raw IBinder object.
            mService = new Messenger(service);
            mBound = true;
        }
    }

    public void onServiceDisconnected(ComponentName className) {
        // This is called when the connection with the service has been
        // unexpectedly disconnected -- that is, its process crashed.
        mService = null;
        mBound = false;
    }
};

public void sayHello(View v) {
    if (!mBound) return;
    // Create and send a message to the service, using a supported 'what' value
    Message msg = Message.obtain(null, MessengerService.MSG_SAY_HELLO, 0, 0);
}

```

```

        try {
            mService.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

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

@Override
protected void onStart() {
    super.onStart();
    // Bind to the service
    bindService(new Intent(this, MessengerService.class), mConnection,
               Context.BIND_AUTO_CREATE);
}

@Override
protected void onStop() {
    super.onStop();
    // Unbind from the service
    if (mBound) {
        unbindService(mConnection);
        mBound = false;
    }
}
}

```

Notice that this example does not show how the service can respond to the client. If you want the service to respond, then you need to also create a [Messenger](#) in the client. Then when the client receives the [onServiceConnected\(\)](#) callback, it sends a [Message](#) to the service that includes the client's [Messenger](#) in the [replyTo](#) parameter of the [send\(\)](#) method.

You can see an example of how to provide two-way messaging in the [MessengerService.java](#) (service) and [MessengerServiceActivities.java](#) (client) samples.

## Binding to a Service

Application components (clients) can bind to a service by calling [bindService\(\)](#). The Android system then calls the service's [onBind\(\)](#) method, which returns an [IBinder](#) for interacting with the service.

The binding is asynchronous. [bindService\(\)](#) returns immediately and does *not* return the [IBinder](#) to the client. To receive the [IBinder](#), the client must create an instance of [ServiceConnection](#) and pass it to [bindService\(\)](#). The [ServiceConnection](#) includes a callback method that the system calls to deliver the [IBinder](#).

**Note:** Only activities, services, and content providers can bind to a service—you **cannot** bind to a service from a broadcast receiver.

So, to bind to a service from your client, you must:

1. Implement [ServiceConnection](#).

Your implementation must override two callback methods:

#### [onServiceConnected\(\)](#)

The system calls this to deliver the [IBinder](#) returned by the service's [onBind\(\)](#) method.

#### [onServiceDisconnected\(\)](#)

The Android system calls this when the connection to the service is unexpectedly lost, such as when the service has crashed or has been killed. This is *not* called when the client unbinds.

2. Call [bindService\(\)](#), passing the [ServiceConnection](#) implementation.
3. When the system calls your [onServiceConnected\(\)](#) callback method, you can begin making calls to the service, using the methods defined by the interface.
4. To disconnect from the service, call [unbindService\(\)](#).

When your client is destroyed, it will unbind from the service, but you should always unbind when you're done interacting with the service or when your activity pauses so that the service can shutdown while its not being used. (Appropriate times to bind and unbind is discussed more below.)

For example, the following snippet connects the client to the service created above by [extending the Binder class](#), so all it must do is cast the returned [IBinder](#) to the `LocalService` class and request the `LocalService` instance:

```
LocalService mService;
private ServiceConnection mConnection = new ServiceConnection() {
    // Called when the connection with the service is established
    public void onServiceConnected(ComponentName className, IBinder service) {
        // Because we have bound to an explicit
        // service that is running in our own process, we can
        // cast its IBinder to a concrete class and directly access it.
        LocalBinder binder = (LocalBinder) service;
        mService = binder.getService();
        mBound = true;
    }

    // Called when the connection with the service disconnects unexpectedly
    public void onServiceDisconnected(ComponentName className) {
        Log.e(TAG, "onServiceDisconnected");
        mBound = false;
    }
};
```

With this [ServiceConnection](#), the client can bind to a service by passing it to [bindService\(\)](#). For example:

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

- The first parameter of [bindService\(\)](#) is an [Intent](#) that explicitly names the service to bind (thought the intent could be implicit).
- The second parameter is the [ServiceConnection](#) object.

- The third parameter is a flag indicating options for the binding. It should usually be `BIND_AUTO_CREATE` in order to create the service if its not already alive. Other possible values are `BIND_DEBUG_UNBIND` and `BIND_NOT_FOREGROUND`, or 0 for none.

## Additional notes

Here are some important notes about binding to a service:

- You should always trap `DeadObjectException` exceptions, which are thrown when the connection has broken. This is the only exception thrown by remote methods.
- Objects are reference counted across processes.
- You should usually pair the binding and unbinding during matching bring-up and tear-down moments of the client's lifecycle. For example:
  - If you only need to interact with the service while your activity is visible, you should bind during `onStart()` and unbind during `onStop()`.
  - If you want your activity to receive responses even while it is stopped in the background, then you can bind during `onCreate()` and unbind during `onDestroy()`. Beware that this implies that your activity needs to use the service the entire time it's running (even in the background), so if the service is in another process, then you increase the weight of the process and it becomes more likely that the system will kill it.

**Note:** You should usually **not** bind and unbind during your activity's `onResume()` and `onPause()`, because these callbacks occur at every lifecycle transition and you should keep the processing that occurs at these transitions to a minimum. Also, if multiple activities in your application bind to the same service and there is a transition between two of those activities, the service may be destroyed and recreated as the current activity unbinds (during pause) before the next one binds (during resume). (This activity transition for how activities coordinate their lifecycles is described in the [Activities](#) document.)

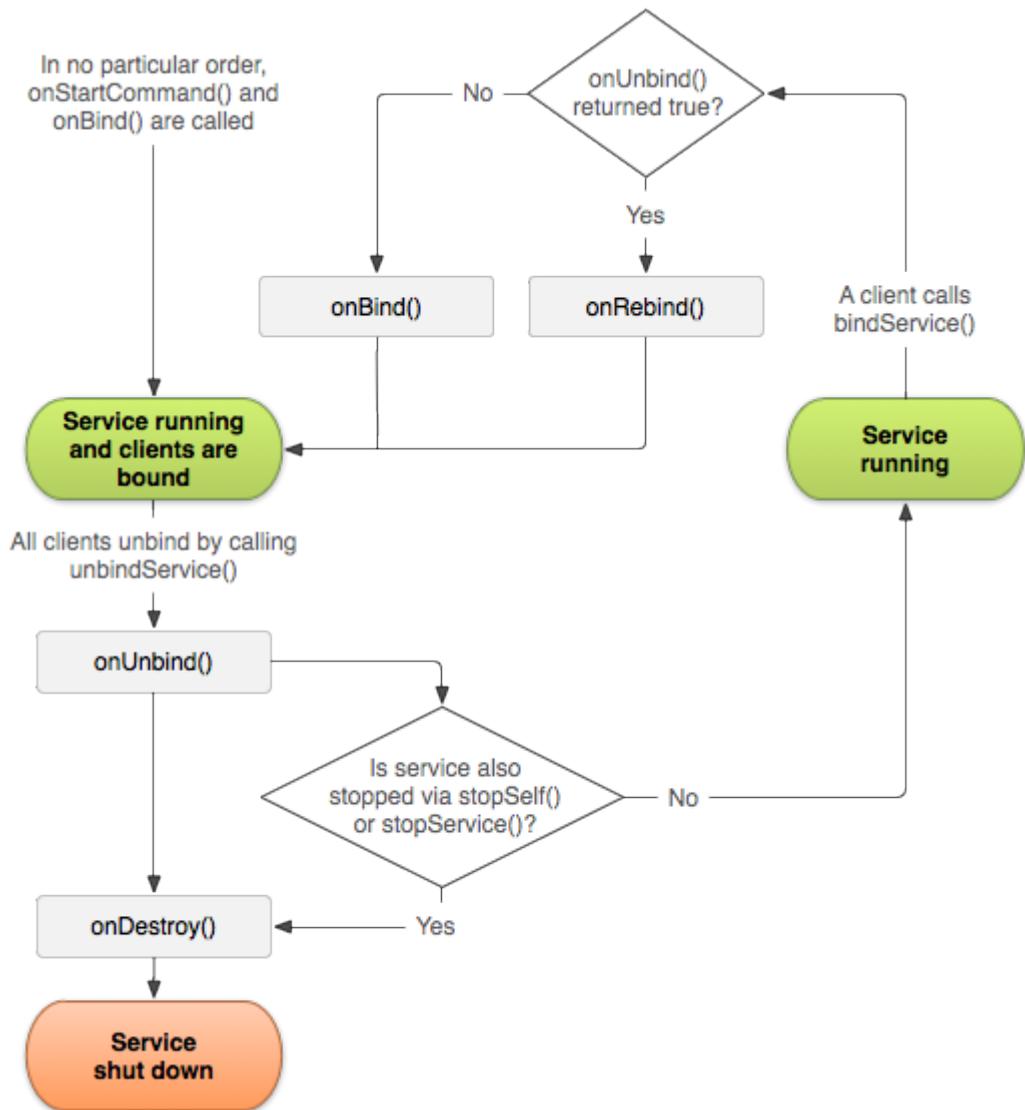
For more sample code, showing how to bind to a service, see the `RemoteService.java` class in [Api-Demos](#).

## Managing the Lifecycle of a Bound Service

When a service is unbound from all clients, the Android system destroys it (unless it was also started with `onStartCommand()`). As such, you don't have to manage the lifecycle of your service if it's purely a bound service—the Android system manages it for you based on whether it is bound to any clients.

However, if you choose to implement the `onStartCommand()` callback method, then you must explicitly stop the service, because the service is now considered to be *started*. In this case, the service runs until the service stops itself with `stopSelf()` or another component calls `stopService()`, regardless of whether it is bound to any clients.

Additionally, if your service is started and accepts binding, then when the system calls your `onUnbind()` method, you can optionally return `true` if you would like to receive a call to `onRebind()` the next time a client binds to the service (instead of receiving a call to `onBind()`). `onRebind()` returns void, but the client still receives the `IBinder` in its `onServiceConnected()` callback. Below, figure 1 illustrates the logic for this kind of lifecycle.



**Figure 1.** The lifecycle for a service that is started and also allows binding.

For more information about the lifecycle of a started service, see the [Services](#) document.

# Android Interface Definition Language (AIDL)

## In this document

1. [Defining an AIDL Interface](#)
  1. [Create the .aidl file](#)
  2. [Implement the interface](#)
  3. [Expose the interface to clients](#)
2. [Passing Objects over IPC](#)
3. [Calling an IPC Method](#)

## See also

1. [Bound Services](#)

AIDL (Android Interface Definition Language) is similar to other IDLs you might have worked with. It allows you to define the programming interface that both the client and service agree upon in order to communicate with each other using interprocess communication (IPC). On Android, one process cannot normally access the memory of another process. So to talk, they need to decompose their objects into primitives that the operating system can understand, and marshall the objects across that boundary for you. The code to do that marshalling is tedious to write, so Android handles it for you with AIDL.

**Note:** Using AIDL is necessary only if you allow clients from different applications to access your service for IPC and want to handle multithreading in your service. If you do not need to perform concurrent IPC across different applications, you should create your interface by [implementing a Binder](#) or, if you want to perform IPC, but do *not* need to handle multithreading, implement your interface [using a Messenger](#). Regardless, be sure that you understand [Bound Services](#) before implementing an AIDL.

Before you begin designing your AIDL interface, be aware that calls to an AIDL interface are direct function calls. You should not make assumptions about the thread in which the call occurs. What happens is different depending on whether the call is from a thread in the local process or a remote process. Specifically:

- Calls made from the local process are executed in the same thread that is making the call. If this is your main UI thread, that thread continues to execute in the AIDL interface. If it is another thread, that is the one that executes your code in the service. Thus, if only local threads are accessing the service, you can completely control which threads are executing in it (but if that is the case, then you shouldn't be using AIDL at all, but should instead create the interface by [implementing a Binder](#)).
- Calls from a remote process are dispatched from a thread pool the platform maintains inside of your own process. You must be prepared for incoming calls from unknown threads, with multiple calls happening at the same time. In other words, an implementation of an AIDL interface must be completely thread-safe.
- The `oneway` keyword modifies the behavior of remote calls. When used, a remote call does not block; it simply sends the transaction data and immediately returns. The implementation of the interface eventually receives this as a regular call from the [Binder](#) thread pool as a normal remote call. If `oneway` is used with a local call, there is no impact and the call is still synchronous.

## Defining an AIDL Interface

You must define your AIDL interface in an `.aidl` file using the Java programming language syntax, then save it in the source code (in the `src/` directory) of both the application hosting the service and any other application that binds to the service.

When you build each application that contains the `.aidl` file, the Android SDK tools generate an [IBinder](#) interface based on the `.aidl` file and save it in the project's `gen/` directory. The service must implement the [IBinder](#) interface as appropriate. The client applications can then bind to the service and call methods from the [IBinder](#) to perform IPC.

To create a bounded service using AIDL, follow these steps:

1. [Create the `.aidl` file](#)

This file defines the programming interface with method signatures.

2. [Implement the interface](#)

The Android SDK tools generate an interface in the Java programming language, based on your `.aidl` file. This interface has an inner abstract class named `Stub` that extends [Binder](#) and implements methods from your AIDL interface. You must extend the `Stub` class and implement the methods.

3. [Expose the interface to clients](#)

Implement a [Service](#) and override [onBind\(\)](#) to return your implementation of the `Stub` class.

**Caution:** Any changes that you make to your AIDL interface after your first release must remain backward compatible in order to avoid breaking other applications that use your service. That is, because your `.aidl` file must be copied to other applications in order for them to access your service's interface, you must maintain support for the original interface.

## 1. Create the `.aidl` file

AIDL uses a simple syntax that lets you declare an interface with one or more methods that can take parameters and return values. The parameters and return values can be of any type, even other AIDL-generated interfaces.

You must construct the `.aidl` file using the Java programming language. Each `.aidl` file must define a single interface and requires only the interface declaration and method signatures.

By default, AIDL supports the following data types:

- All primitive types in the Java programming language (such as `int`, `long`, `char`, `boolean`, and so on)
- [String](#)
- [CharSequence](#)
- [List](#)

All elements in the [List](#) must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you've declared. A [List](#) may optionally be used as a "generic" class (for example, `List<String>`). The actual concrete class that the other side receives is always an [ArrayList](#), although the method is generated to use the [List](#) interface.

- [Map](#)

All elements in the [Map](#) must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you've declared. Generic maps, (such as those of the form `Map<String, Integer>`) are not supported. The actual concrete class that the other side receives is always a [HashMap](#), although the method is generated to use the [Map](#) interface.

You must include an `import` statement for each additional type not listed above, even if they are defined in the same package as your interface.

When defining your service interface, be aware that:

- Methods can take zero or more parameters, and return a value or void.
- All non-primitive parameters require a directional tag indicating which way the data goes. Either `in`, `out`, or `inout` (see the example below).

Primitives are `in` by default, and cannot be otherwise.

**Caution:** You should limit the direction to what is truly needed, because marshalling parameters is expensive.

- All code comments included in the `.aidl` file are included in the generated [IBinder](#) interface (except for comments before the `import` and `package` statements).
- Only methods are supported; you cannot expose static fields in AIDL.

Here is an example `.aidl` file:

```
// IRemoteService.aidl
package com.example.android;

// Declare any non-default types here with import statements

/** Example service interface */
interface IRemoteService {
    /** Request the process ID of this service, to do evil things with it. */
    int getPid();

    /** Demonstrates some basic types that you can use as parameters
     * and return values in AIDL.
    */
    void basicTypes(int anInt, long aLong, boolean aBoolean, float aFloat,
                   double aDouble, String aString);
}
```

Simply save your `.aidl` file in your project's `src/` directory and when you build your application, the SDK tools generate the [IBinder](#) interface file in your project's `gen/` directory. The generated file name matches the `.aidl` file name, but with a `.java` extension (for example, `IRemoteService.aidl` results in `IRemoteService.java`).

If you use Eclipse, the incremental build generates the binder class almost immediately. If you do not use Eclipse, then the Ant tool generates the binder class next time you build your application—you should build your project with `ant debug` (or `ant release`) as soon as you're finished writing the `.aidl` file, so that your code can link against the generated class.

## 2. Implement the interface

When you build your application, the Android SDK tools generate a `.java` interface file named after your `.aidl` file. The generated interface includes a subclass named `Stub` that is an abstract implementation of its parent interface (for example, `YourInterface.Stub`) and declares all the methods from the `.aidl` file.

**Note:** Stub also defines a few helper methods, most notably `asInterface()`, which takes an `IBinder` (usually the one passed to a client's `onServiceConnected()` callback method) and returns an instance of the stub interface. See the section [Calling an IPC Method](#) for more details on how to make this cast.

To implement the interface generated from the `.aidl`, extend the generated `Binder` interface (for example, `YourInterface.Stub`) and implement the methods inherited from the `.aidl` file.

Here is an example implementation of an interface called `IRemoteService` (defined by the `IRemoteService.aidl` example, above) using an anonymous instance:

```
private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
    public int getPid(){
        return Process.myPid();
    }
    public void basicTypes(int anInt, long aLong, boolean aBoolean,
        float aFloat, double aDouble, String aString) {
        // Does nothing
    }
};
```

Now the `mBinder` is an instance of the `Stub` class (a `Binder`), which defines the RPC interface for the service. In the next step, this instance is exposed to clients so they can interact with the service.

There are a few rules you should be aware of when implementing your AIDL interface:

- Incoming calls are not guaranteed to be executed on the main thread, so you need to think about multi-threading from the start and properly build your service to be thread-safe.
- By default, RPC calls are synchronous. If you know that the service takes more than a few milliseconds to complete a request, you should not call it from the activity's main thread, because it might hang the application (Android might display an "Application is Not Responding" dialog)—you should usually call them from a separate thread in the client.
- No exceptions that you throw are sent back to the caller.

### 3. Expose the interface to clients

Once you've implemented the interface for your service, you need to expose it to clients so they can bind to it. To expose the interface for your service, extend `Service` and implement `onBind()` to return an instance of your class that implements the generated `Stub` (as discussed in the previous section). Here's an example service that exposes the `IRemoteService` example interface to clients.

```
public class RemoteService extends Service {
    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public IBinder onBind(Intent intent) {
        // Return the interface
        return mBinder;
    }

    private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
        public int getPid(){
```

```

        return Process.myPid();
    }
    public void basicTypes(int anInt, long aLong, boolean aBoolean,
        float aFloat, double aDouble, String aString) {
        // Does nothing
    }
}
}

```

Now, when a client (such as an activity) calls [bindService\(\)](#) to connect to this service, the client's [onServiceConnected\(\)](#) callback receives the `mBinder` instance returned by the service's [onBind\(\)](#) method.

The client must also have access to the interface class, so if the client and service are in separate applications, then the client's application must have a copy of the `.aidl` file in its `src/` directory (which generates the `android.os.Binder` interface—providing the client access to the AIDL methods).

When the client receives the `IBinder` in the [onServiceConnected\(\)](#) callback, it must call `YourServiceInterface.Stub.asInterface(service)` to cast the returned parameter to `YourServiceInterface` type. For example:

```

IRemoteService mIRemoteService;
private ServiceConnection mConnection = new ServiceConnection() {
    // Called when the connection with the service is established
    public void onServiceConnected(ComponentName className, IBinder service) {
        // Following the example above for an AIDL interface,
        // this gets an instance of the IRemoteInterface, which we can use to ...
        mIRemoteService = IRemoteService.Stub.asInterface(service);
    }

    // Called when the connection with the service disconnects unexpectedly
    public void onServiceDisconnected(ComponentName className) {
        Log.e(TAG, "Service has unexpectedly disconnected");
        mIRemoteService = null;
    }
};

```

For more sample code, see the [RemoteService.java](#) class in [ApiDemos](#).

## Passing Objects over IPC

If you have a class that you would like to send from one process to another through an IPC interface, you can do that. However, you must ensure that the code for your class is available to the other side of the IPC channel and your class must support the [Parcelable](#) interface. Supporting the [Parcelable](#) interface is important because it allows the Android system to decompose objects into primitives that can be marshalled across processes.

To create a class that supports the [Parcelable](#) protocol, you must do the following:

1. Make your class implement the [Parcelable](#) interface.
2. Implement [writeToParcel](#), which takes the current state of the object and writes it to a [Parcel](#).
3. Add a static field called `CREATOR` to your class which is an object implementing the [Parcelable.Creator](#) interface.

- Finally, create an `.aidl` file that declares your parcelable class (as shown for the `Rect.aidl` file, below).

If you are using a custom build process, do *not* add the `.aidl` file to your build. Similar to a header file in the C language, this `.aidl` file isn't compiled.

AIDL uses these methods and fields in the code it generates to marshall and unmarshall your objects.

For example, here is a `Rect.aidl` file to create a `Rect` class that's parcelable:

```
package android.graphics;

// Declare Rect so AIDL can find it and knows that it implements
// the parcelable protocol.
parcelable Rect;
```

And here is an example of how the [Rect](#) class implements the [Parcelable](#) protocol.

```
import android.os.Parcel;
import android.os.Parcelable;

public final class Rect implements Parcelable {
    public int left;
    public int top;
    public int right;
    public int bottom;

    public static final Parcelable.Creator<Rect> CREATOR = new
Parcelable.Creator<Rect>() {
        public Rect createFromParcel(Parcel in) {
            return new Rect(in);
        }

        public Rect[] newArray(int size) {
            return new Rect[size];
        }
    };

    public Rect() {
    }

    private Rect(Parcel in) {
        readFromParcel(in);
    }

    public void writeToParcel(Parcel out) {
        out.writeInt(left);
        out.writeInt(top);
        out.writeInt(right);
        out.writeInt(bottom);
    }

    public void readFromParcel(Parcel in) {
        left = in.readInt();
```

```

        top = in.readInt();
        right = in.readInt();
        bottom = in.readInt();
    }
}

```

The marshalling in the `Rect` class is pretty simple. Take a look at the other methods on [Parcel](#) to see the other kinds of values you can write to a `Parcel`.

**Warning:** Don't forget the security implications of receiving data from other processes. In this case, the `Rect` reads four numbers from the [Parcel](#), but it is up to you to ensure that these are within the acceptable range of values for whatever the caller is trying to do. See [Security and Permissions](#) for more information about how to keep your application secure from malware.

## Calling an IPC Method

Here are the steps a calling class must take to call a remote interface defined with AIDL:

1. Include the `.aidl` file in the project `src/` directory.
2. Declare an instance of the [IBinder](#) interface (generated based on the AIDL).
3. Implement [ServiceConnection](#).
4. Call [Context.bindService\(\)](#), passing in your [ServiceConnection](#) implementation.
5. In your implementation of [onServiceConnected\(\)](#), you will receive an [IBinder](#) instance (called `service`). Call `YourInterfaceName.Stub.asInterface((IBinder) service)` to cast the returned parameter to `YourInterface` type.
6. Call the methods that you defined on your interface. You should always trap [DeadObjectException](#) exceptions, which are thrown when the connection has broken; this will be the only exception thrown by remote methods.
7. To disconnect, call [Context.unbindService\(\)](#) with the instance of your interface.

A few comments on calling an IPC service:

- Objects are reference counted across processes.
- You can send anonymous objects as method arguments.

For more information about binding to a service, read the [Bound Services](#) document.

Here is some sample code demonstrating calling an AIDL-created service, taken from the Remote Service sample in the ApiDemos project.

```

public static class Binding extends Activity {
    /** The primary interface we will be calling on the service. */
    IRemoteService mService = null;
    /** Another interface we use on the service. */
    ISecondary mSecondaryService = null;

    Button mKillButton;
    TextView mCallbackText;

    private boolean mIsBound;

    /**
     * Standard initialization of this activity. Set up the UI, then wait

```

```
* for the user to poke it before doing anything.  
*/  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    setContentView(R.layout.remote_service_binding);  
  
    // Watch for button clicks.  
    Button button = (Button) findViewById(R.id.bind);  
    button.setOnClickListener(mBindListener);  
    button = (Button) findViewById(R.id.unbind);  
    button.setOnClickListener(mUnbindListener);  
    mKillButton = (Button) findViewById(R.id.kill);  
    mKillButton.setOnClickListener(mKillListener);  
    mKillButton.setEnabled(false);  
  
    mCallbackText = (TextView) findViewById(R.id.callback);  
    mCallbackText.setText("Not attached.");  
}  
  
/**  
 * Class for interacting with the main interface of the service.  
 */  
private ServiceConnection mConnection = new ServiceConnection() {  
    public void onServiceConnected(ComponentName className,  
        IBinder service) {  
        // This is called when the connection with the service has been  
        // established, giving us the service object we can use to  
        // interact with the service. We are communicating with our  
        // service through an IDL interface, so get a client-side  
        // representation of that from the raw service object.  
        mService = IRemoteService.Stub.asInterface(service);  
        mKillButton.setEnabled(true);  
        mCallbackText.setText("Attached.");  
  
        // We want to monitor the service for as long as we are  
        // connected to it.  
        try {  
            mService.registerCallback(mCallback);  
        } catch (RemoteException e) {  
            // In this case the service has crashed before we could even  
            // do anything with it; we can count on soon being  
            // disconnected (and then reconnected if it can be restarted)  
            // so there is no need to do anything here.  
        }  
  
        // As part of the sample, tell the user what happened.  
        Toast.makeText(Binding.this, R.string.remote_service_connected,  
            Toast.LENGTH_SHORT).show();  
    }  
  
    public void onServiceDisconnected(ComponentName className) {  
        // This is called when the connection with the service has been
```

```
// unexpectedly disconnected -- that is, its process crashed.
mService = null;
mKillButton.setEnabled(false);
mCallbackText.setText("Disconnected.");

// As part of the sample, tell the user what happened.
Toast.makeText(Binding.this, R.string.remote_service_disconnected,
    Toast.LENGTH_SHORT).show();
}

};

/***
 * Class for interacting with the secondary interface of the service.
 */
private ServiceConnection mSecondaryConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
        IBinder service) {
        // Connecting to a secondary interface is the same as any
        // other interface.
        mSecondaryService = ISecondary.Stub.asInterface(service);
        mKillButton.setEnabled(true);
    }

    public void onServiceDisconnected(ComponentName className) {
        mSecondaryService = null;
        mKillButton.setEnabled(false);
    }
};

private OnClickListener mBindListener = new OnClickListener() {
    public void onClick(View v) {
        // Establish a couple connections with the service, binding
        // by interface names. This allows other applications to be
        // installed that replace the remote service by implementing
        // the same interface.
        bindService(new Intent(IRemoteService.class.getName()),
            mConnection, Context.BIND_AUTO_CREATE);
        bindService(new Intent(ISecondary.class.getName()),
            mSecondaryConnection, Context.BIND_AUTO_CREATE);
        mIsBound = true;
        mCallbackText.setText("Binding.");
    }
};

private OnClickListener mUnbindListener = new OnClickListener() {
    public void onClick(View v) {
        if (mIsBound) {
            // If we have received the service, and hence registered with
            // it, then now is the time to unregister.
            if (mService != null) {
                try {
                    mService.unregisterCallback(mCallback);
                } catch (RemoteException e) {
                    // There is nothing special we need to do if the service
                }
            }
        }
    }
};
```

```

        // has crashed.
    }

}

// Detach our existing connection.
unbindService(mConnection);
unbindService(mSecondaryConnection);
mKillButton.setEnabled(false);
mIsBound = false;
mCallbackText.setText("Unbinding.");
}
}

};

private OnClickListener mKillListener = new OnClickListener() {
    public void onClick(View v) {
        // To kill the process hosting our service, we need to know its
        // PID. Conveniently our service has a call that will return
        // to us that information.
        if (mSecondaryService != null) {
            try {
                int pid = mSecondaryService.getPid();
                // Note that, though this API allows us to request to
                // kill any process based on its PID, the kernel will
                // still impose standard restrictions on which PIDs you
                // are actually able to kill. Typically this means only
                // the process running your application and any additional
                // processes created by that app as shown here; packages
                // sharing a common UID will also be able to kill each
                // other's processes.
                Process.killProcess(pid);
                mCallbackText.setText("Killed service process.");
            } catch (RemoteException ex) {
                // Recover gracefully from the process hosting the
                // server dying.
                // Just for purposes of the sample, put up a notification.
                Toast.makeText(Binding.this,
                    R.string.remote_call_failed,
                    Toast.LENGTH_SHORT).show();
            }
        }
    }
};

// -----
// Code showing how to deal with callbacks.
// -----


/**
 * This implementation is used to receive callbacks from the remote
 * service.
 */
private IRemoteServiceCallback mCallback = new IRemoteServiceCallback.Stub()
{

```

```
* This is called by the remote service regularly to tell us about
* new values. Note that IPC calls are dispatched through a thread
* pool running in each process, so the code executing here will
* NOT be running in our main thread like most other things -- so,
* to update the UI, we need to use a Handler to hop over there.
*/
public void valueChanged(int value) {
    mHandler.sendMessage(mHandler.obtainMessage(BUMP_MSG, value, 0));
}
};

private static final int BUMP_MSG = 1;

private Handler mHandler = new Handler() {
    @Override public void handleMessage(Message msg) {
        switch (msg.what) {
            case BUMP_MSG:
                mCallbackText.setText("Received from service: " + msg.arg1);
                break;
            default:
                super.handleMessage(msg);
        }
    }
};

}
```

# Content Providers

## Topics

1. [Content Provider Basics](#)
2. [Creating a Content Provider](#)
3. [Calendar Provider](#)
4. [Contacts Provider](#)

## Related Samples

1. [Contact Manager](#) application
2. ["Cursor \(People\)"](#)
3. ["Cursor \(Phones\)"](#)
4. [Sample Sync Adapter](#)

Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.

When you want to access data in a content provider, you use the [ContentResolver](#) object in your application's [Context](#) to communicate with the provider as a client. The [ContentResolver](#) object communicates with the provider object, an instance of a class that implements [ContentProvider](#). The provider object receives data requests from clients, performs the requested action, and returns the results.

You don't need to develop your own provider if you don't intend to share your data with other applications. However, you do need your own provider to provide custom search suggestions in your own application. You also need your own provider if you want to copy and paste complex data or files from your application to other applications.

Android itself includes content providers that manage data such as audio, video, images, and personal contact information. You can see some of them listed in the reference documentation for the [android.provider](#) package. With some restrictions, these providers are accessible to any Android application.

The following topics describe content providers in more detail:

### [Content Provider Basics](#)

How to access data in a content provider when the data is organized in tables.

### [Creating a Content Provider](#)

How to create your own content provider.

### [Calendar Provider](#)

How to access the Calendar Provider that is part of the Android platform.

### [Contacts Provider](#)

How to access the Contacts Provider that is part of the Android platform.

# Content Provider Basics

## In this document

1. [Overview](#)
  1. [Accessing a provider](#)
  2. [Content URIs](#)
2. [Retrieving Data from the Provider](#)
  1. [Requesting read access permission](#)
  2. [Constructing the query](#)
  3. [Displaying query results](#)
  4. [Getting data from query results](#)
3. [Content Provider Permissions](#)
4. [Inserting, Updating, and Deleting Data](#)
  1. [Inserting data](#)
  2. [Updating data](#)
  3. [Deleting data](#)
5. [Provider Data Types](#)
6. [Alternative Forms of Provider Access](#)
  1. [Batch access](#)
  2. [Data access via intents](#)
7. [Contract Classes](#)
8. [MIME Type Reference](#)

## Key classes

1. [ContentProvider](#)
2. [ContentResolver](#)
3. [Cursor](#)
4. [Uri](#)

## Related Samples

1. [Cursor \(People\)](#)
2. [Cursor \(Phones\)](#)

## See also

1. [Creating a Content Provider](#)
2. [Calendar Provider](#)

A content provider manages access to a central repository of data. A provider is part of an Android application, which often provides its own UI for working with the data. However, content providers are primarily intended to be used by other applications, which access the provider using a provider client object. Together, providers and provider clients offer a consistent, standard interface to data that also handles inter-process communication and secure data access.

This topic describes the basics of the following:

- How content providers work.
- The API you use retrieve data from a content provider.

- The API you use to insert, update, or delete data in a content provider.
- Other API features that facilitate working with providers.

## Overview

A content provider presents data to external applications as one or more tables that are similar to the tables found in a relational database. A row represents an instance of some type of data the provider collects, and each column in the row represents an individual piece of data collected for an instance.

For example, one of the built-in providers in the Android platform is the user dictionary, which stores the spellings of non-standard words that the user wants to keep. Table 1 illustrates what the data might look like in this provider's table:

**Table 1:** Sample user dictionary table.

word	app id	frequency	locale _ID	
mapreduce	user1	100	en_US	1
precompiler	user14	200	fr_FR	2
applet	user2	225	fr_CA	3
const	user1	255	pt_BR	4
int	user5	100	en_UK	5

In table 1, each row represents an instance of a word that might not be found in a standard dictionary. Each column represents some data for that word, such as the locale in which it was first encountered. The column headers are column names that are stored in the provider. To refer to a row's locale, you refer to its `locale` column. For this provider, the `_ID` column serves as a "primary key" column that the provider automatically maintains.

**Note:** A provider isn't required to have a primary key, and it isn't required to use `_ID` as the column name of a primary key if one is present. However, if you want to bind data from a provider to a [ListView](#), one of the column names has to be `_ID`. This requirement is explained in more detail in the section [Displaying query results](#).

## Accessing a provider

An application accesses the data from a content provider with a [ContentResolver](#) client object. This object has methods that call identically-named methods in the provider object, an instance of one of the concrete subclasses of [ContentProvider](#). The [ContentResolver](#) methods provide the basic "CRUD" (create, retrieve, update, and delete) functions of persistent storage.

The [ContentResolver](#) object in the client application's process and the [ContentProvider](#) object in the application that owns the provider automatically handle inter-process communication. [ContentProvider](#) also acts as an abstraction layer between its repository of data and the external appearance of data as tables.

**Note:** To access a provider, your application usually has to request specific permissions in its manifest file. This is described in more detail in the section [Content Provider Permissions](#)

For example, to get a list of the words and their locales from the User Dictionary Provider, you call `ContentResolver.query()`. The `query()` method calls the `ContentProvider.query()` method defined by the User Dictionary Provider. The following lines of code show a `ContentResolver.query()` call:

```

// Queries the user dictionary and returns results
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,           // The content URI of the words table
    mProjection,                                // The columns to return for each row
    mSelectionClause,                           // Selection criteria
    mSelectionArgs,                             // Selection criteria
    mSortOrder);                               // The sort order for the returned rows

```

Table 2 shows how the arguments to `query(Uri,projection,selection,selectionArgs,sortOrder)` match an SQL SELECT statement:

**Table 2:** Query() compared to SQL query.

query() argument	SELECT keyword/parameter	Notes
Uri	FROM <i>table_name</i>	<i>Uri</i> maps to the table in the provider named <i>table_name</i> .
projection	<i>col,col,col,...</i>	<i>projection</i> is an array of columns that should be included for each row retrieved.
selection	WHERE <i>col</i> = <i>value</i>	<i>selection</i> specifies the criteria for selecting rows.
selectionArgs	(No exact equivalent. Selection arguments replace ? placeholders in the selection clause.)	
sortOrder	ORDER BY <i>col,col,...</i>	<i>sortOrder</i> specifies the order in which rows appear in the returned <a href="#">Cursor</a> .

## Content URIs

A **content URI** is a URI that identifies data in a provider. Content URIs include the symbolic name of the entire provider (its **authority**) and a name that points to a table (a **path**). When you call a client method to access a table in a provider, the content URI for the table is one of the arguments.

In the preceding lines of code, the constant `CONTENT_URI` contains the content URI of the user dictionary's "words" table. The `ContentResolver` object parses out the URI's authority, and uses it to "resolve" the provider by comparing the authority to a system table of known providers. The `ContentResolver` can then dispatch the query arguments to the correct provider.

The `ContentProvider` uses the path part of the content URI to choose the table to access. A provider usually has a **path** for each table it exposes.

In the previous lines of code, the full URI for the "words" table is:

```
content://user_dictionary/words
```

where the `user_dictionary` string is the provider's authority, and `words` string is the table's path. The string `content://` (the **scheme**) is always present, and identifies this as a content URI.

Many providers allow you to access a single row in a table by appending an ID value to the end of the URI. For example, to retrieve a row whose `_ID` is 4 from user dictionary, you can use this content URI:

```
Uri singleUri = ContentUris.withAppendedId(UserDictionary.Words.CONTENT_URI, 4);
```

You often use id values when you've retrieved a set of rows and then want to update or delete one of them.

**Note:** The [Uri](#) and [Uri.Builder](#) classes contain convenience methods for constructing well-formed Uri objects from strings. The [ContentUris](#) contains convenience methods for appending id values to a URI. The previous snippet uses [withAppendedId\(\)](#) to append an id to the UserDictionary content URI.

## Retrieving Data from the Provider

This section describes how to retrieve data from a provider, using the User Dictionary Provider as an example.

For the sake of clarity, the code snippets in this section call [ContentResolver.query\(\)](#) on the "UI thread"". In actual code, however, you should do queries asynchronously on a separate thread. One way to do this is to use the [CursorLoader](#) class, which is described in more detail in the [Loaders](#) guide. Also, the lines of code are snippets only; they don't show a complete application.

To retrieve data from a provider, follow these basic steps:

1. Request the read access permission for the provider.
2. Define the code that sends a query to the provider.

### Requesting read access permission

To retrieve data from a provider, your application needs "read access permission" for the provider. You can't request this permission at run-time; instead, you have to specify that you need this permission in your manifest, using the [<uses-permission>](#) element and the exact permission name defined by the provider. When you specify this element in your manifest, you are in effect "requesting" this permission for your application. When users install your application, they implicitly grant this request.

To find the exact name of the read access permission for the provider you're using, as well as the names for other access permissions used by the provider, look in the provider's documentation.

The role of permissions in accessing providers is described in more detail in the section [Content Provider Permissions](#).

The User Dictionary Provider defines the permission `android.permission.READ_USER_DICTIONARY` in its manifest file, so an application that wants to read from the provider must request this permission.

### Constructing the query

The next step in retrieving data a provider is to construct a query. This first snippet defines some variables for accessing the User Dictionary Provider:

```
// A "projection" defines the columns that will be returned for each row
String[] mProjection =
{
    UserDictionary.Words._ID,      // Contract class constant for the _ID column
    UserDictionary.Words.WORD,     // Contract class constant for the word column
    UserDictionary.Words.LOCALE   // Contract class constant for the locale column
};

// Defines a string to contain the selection clause
String mSelectionClause = null;

// Initializes an array to contain selection arguments
```

```
String[] mSelectionArgs = {"");
```

The next snippet shows how to use [ContentResolver.query\(\)](#), using the User Dictionary Provider as an example. A provider client query is similar to an SQL query, and it contains a set of columns to return, a set of selection criteria, and a sort order.

The set of columns that the query should return is called a **projection** (the variable `mProjection`).

The expression that specifies the rows to retrieve is split into a selection clause and selection arguments. The selection clause is a combination of logical and Boolean expressions, column names, and values (the variable `mSelectionClause`). If you specify the replaceable parameter `?` instead of a value, the query method retrieves the value from the selection arguments array (the variable `mSelectionArgs`).

In the next snippet, if the user doesn't enter a word, the selection clause is set to `null`, and the query returns all the words in the provider. If the user enters a word, the selection clause is set to `UserDictionary.Words.WORD + " = ?"` and the first element of selection arguments array is set to the word the user enters.

```
/*
 * This defines a one-element String array to contain the selection argument.
 */
String[] mSelectionArgs = {"};

// Gets a word from the UI
mSearchString = mSearchWord.getText().toString();

// Remember to insert code here to check for invalid or malicious input.

// If the word is the empty string, gets everything
if (TextUtils.isEmpty(mSearchString)) {
    // Setting the selection clause to null will return all words
    mSelectionClause = null;
    mSelectionArgs[0] = "";

} else {
    // Constructs a selection clause that matches the word that the user entered
    mSelectionClause = UserDictionary.Words.WORD + " = ?";

    // Moves the user's input string to the selection arguments.
    mSelectionArgs[0] = mSearchString;

}

// Does a query against the table and returns a Cursor object
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,           // The content URI of the words table
    mProjection,                             // The columns to return for each row
    mSelectionClause,                        // Either null, or the word the user entered
    mSelectionArgs,                           // Either empty, or the string the user entered
    mSortOrder);                            // The sort order for the returned rows

// Some providers return null if an error occurs, others throw an exception
if (null == mCursor) {
```

```

/*
 * Insert code here to handle the error. Be sure not to use the cursor! You
 * call android.util.Log.e() to log this error.
 *
 */
// If the Cursor is empty, the provider found no matches
} else if (mCursor.getCount() < 1) {

/*
 * Insert code here to notify the user that the search was unsuccessful. This
 * is an error. You may want to offer the user the option to insert a new row,
 * or search term.
*/
}

} else {
    // Insert code here to do something with the results
}

```

This query is analogous to the SQL statement:

```
SELECT _ID, word, locale FROM words WHERE word = <userinput> ORDER BY word ASC;
```

In this SQL statement, the actual column names are used instead of contract class constants.

## Protecting against malicious input

If the data managed by the content provider is in an SQL database, including external untrusted data into raw SQL statements can lead to SQL injection.

Consider this selection clause:

```
// Constructs a selection clause by concatenating the user's input to the column
String mSelectionClause = "var = " + mUserInput;
```

If you do this, you're allowing the user to concatenate malicious SQL onto your SQL statement. For example, the user could enter "nothing; DROP TABLE \*;" for mUserInput, which would result in the selection clause var = nothing; DROP TABLE \*;. Since the selection clause is treated as an SQL statement, this might cause the provider to erase all of the tables in the underlying SQLite database (unless the provider is set up to catch [SQL injection](#) attempts).

To avoid this problem, use a selection clause that uses ? as a replaceable parameter and a separate array of selection arguments. When you do this, the user input is bound directly to the query rather than being interpreted as part of an SQL statement. Because it's not treated as SQL, the user input can't inject malicious SQL. Instead of using concatenation to include the user input, use this selection clause:

```
// Constructs a selection clause with a replaceable parameter
String mSelectionClause = "var = ?";
```

Set up the array of selection arguments like this:

```
// Defines an array to contain the selection arguments
String[] selectionArgs = {"");
```

Put a value in the selection arguments array like this:

```
// Sets the selection argument to the user's input  
selectionArgs[0] = mUserInput;
```

A selection clause that uses ? as a replaceable parameter and an array of selection arguments array are preferred way to specify a selection, even if the provider isn't based on an SQL database.

## Displaying query results

The [ContentResolver.query\(\)](#) client method always returns a [Cursor](#) containing the columns specified by the query's projection for the rows that match the query's selection criteria. A [Cursor](#) object provides random read access to the rows and columns it contains. Using [Cursor](#) methods, you can iterate over the rows in the results, determine the data type of each column, get the data out of a column, and examine other properties of the results. Some [Cursor](#) implementations automatically update the object when the provider's data changes, or trigger methods in an observer object when the [Cursor](#) changes, or both.

**Note:** A provider may restrict access to columns based on the nature of the object making the query. For example, the Contacts Provider restricts access for some columns to sync adapters, so it won't return them to an activity or service.

If no rows match the selection criteria, the provider returns a [Cursor](#) object for which [Cursor.getCount\(\)](#) is 0 (an empty cursor).

If an internal error occurs, the results of the query depend on the particular provider. It may choose to return null, or it may throw an [Exception](#).

Since a [Cursor](#) is a "list" of rows, a good way to display the contents of a [Cursor](#) is to link it to a [ListView](#) via a [SimpleCursorAdapter](#).

The following snippet continues the code from the previous snippet. It creates a [SimpleCursorAdapter](#) object containing the [Cursor](#) retrieved by the query, and sets this object to be the adapter for a [ListView](#):

```
// Defines a list of columns to retrieve from the Cursor and load into an output  
String[] mWordListColumns =  
{  
    UserDictionary.Words.WORD,      // Contract class constant containing the word  
    UserDictionary.Words.LOCALE   // Contract class constant containing the locale  
};  
  
// Defines a list of View IDs that will receive the Cursor columns for each row  
int[] mWordListItems = { R.id.dictWord, R.id.locale};  
  
// Creates a new SimpleCursorAdapter  
mCursorAdapter = new SimpleCursorAdapter(  
    getApplicationContext(),           // The application's Context object  
    R.layout.wordlistrow,            // A layout in XML for one row in the list  
    mCursor,                         // The result from the query  
    mWordListColumns,                // A string array of column names in the query result  
    mWordListItems,                  // An integer array of view IDs in the layout  
    0);                             // Flags (usually none are needed)  
  
// Sets the adapter for the ListView  
mWordList.setAdapter(mCursorAdapter);
```

**Note:** To back a [ListView](#) with a [Cursor](#), the cursor must contain a column named `_ID`. Because of this, the query shown previously retrieves the `_ID` column for the "words" table, even though the [ListView](#) doesn't display it. This restriction also explains why most providers have a `_ID` column for each of their tables.

## Getting data from query results

Rather than simply displaying query results, you can use them for other tasks. For example, you can retrieve spellings from the user dictionary and then look them up in other providers. To do this, you iterate over the rows in the [Cursor](#):

```
// Determine the column index of the column named "word"
int index = mCursor.getColumnIndex(UserDictionary.Words.WORD);

/*
 * Only executes if the cursor is valid. The User Dictionary Provider returns null
 * if an internal error occurs. Other providers may throw an Exception instead of null.
 */

if (mCursor != null) {
    /*
     * Moves to the next row in the cursor. Before the first movement in the cursor,
     * "row pointer" is -1, and if you try to retrieve data at that position you
     * get an exception.
     */
    while (mCursor.moveToNext()) {

        // Gets the value from the column.
        newWord = mCursor.getString(index);

        // Insert code here to process the retrieved word.

        ...
    }
} else {
    // Insert code here to report an error if the cursor is null or the provider
}
```

[Cursor](#) implementations contain several "get" methods for retrieving different types of data from the object. For example, the previous snippet uses [getString\(\)](#). They also have a [getType\(\)](#) method that returns a value indicating the data type of the column.

## Content Provider Permissions

A provider's application can specify permissions that other applications must have in order to access the provider's data. These permissions ensure that the user knows what data an application will try to access. Based on the provider's requirements, other applications request the permissions they need in order to access the provider. End users see the requested permissions when they install the application.

If a provider's application doesn't specify any permissions, then other applications have no access to the provider's data. However, components in the provider's application always have full read and write access, regardless of the specified permissions.

As noted previously, the User Dictionary Provider requires the `android.permission.READ_USER_DICTIONARY` permission to retrieve data from it. The provider has the separate `android.permission.WRITE_USER_DICTIONARY` permission for inserting, updating, or deleting data.

To get the permissions needed to access a provider, an application requests them with a [`<uses-permission>`](#) element in its manifest file. When the Android Package Manager installs the application, a user must approve all of the permissions the application requests. If the user approves all of them, Package Manager continues the installation; if the user doesn't approve them, Package Manager aborts the installation.

The following [`<uses-permission>`](#) element requests read access to the User Dictionary Provider:

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

The impact of permissions on provider access is explained in more detail in the [Security and Permissions](#) guide.

## Inserting, Updating, and Deleting Data

In the same way that you retrieve data from a provider, you also use the interaction between a provider client and the provider's [ContentProvider](#) to modify data. You call a method of [ContentResolver](#) with arguments that are passed to the corresponding method of [ContentProvider](#). The provider and provider client automatically handle security and inter-process communication.

### Inserting data

To insert data into a provider, you call the [`ContentResolver.insert\(\)`](#) method. This method inserts a new row into the provider and returns a content URI for that row. This snippet shows how to insert a new word into the User Dictionary Provider:

```
// Defines a new Uri object that receives the result of the insertion
Uri mNewUri;

...
// Defines an object to contain the new values to insert
ContentValues mNewValues = new ContentValues();

/*
 * Sets the values of each column and inserts the word. The arguments to the "p
 * method are "column name" and "value"
 */
mNewValues.put(UserDictionary.Words.APP_ID, "example.user");
mNewValues.put(UserDictionary.Words.LOCALE, "en_US");
mNewValues.put(UserDictionary.Words.WORD, "insert");
mNewValues.put(UserDictionary.Words.FREQUENCY, "100");

mNewUri = getContentResolver().insert(
    UserDictionary.Word.CONTENT_URI, // the user dictionary content URI
```

```
mNewValues // the values to insert  
);
```

The data for the new row goes into a single [ContentValues](#) object, which is similar in form to a one-row cursor. The columns in this object don't need to have the same data type, and if you don't want to specify a value at all, you can set a column to null using [ContentValues.putNull\(\)](#).

The snippet doesn't add the `_ID` column, because this column is maintained automatically. The provider assigns a unique value of `_ID` to every row that is added. Providers usually use this value as the table's primary key.

The content URI returned in `newUri` identifies the newly-added row, with the following format:

```
content://user_dictionary/words/<id_value>
```

The `<id_value>` is the contents of `_ID` for the new row. Most providers can detect this form of content URI automatically and then perform the requested operation on that particular row.

To get the value of `_ID` from the returned [Uri](#), call [ContentUris.parseId\(\)](#).

## Updating data

To update a row, you use a [ContentValues](#) object with the updated values just as you do with an insertion, and selection criteria just as you do with a query. The client method you use is [ContentResolver.update\(\)](#). You only need to add values to the [ContentValues](#) object for columns you're updating. If you want to clear the contents of a column, set the value to null.

The following snippet changes all the rows whose locale has the language "en" to a have a locale of null. The return value is the number of rows that were updated:

```
// Defines an object to contain the updated values  
ContentValues mUpdateValues = new ContentValues();  
  
// Defines selection criteria for the rows you want to update  
String mSelectionClause = UserDictionary.Words.LOCALE + "LIKE ?";  
String[] mSelectionArgs = {"en_%"};  
  
// Defines a variable to contain the number of updated rows  
int mRowsUpdated = 0;  
  
...  
  
/*  
 * Sets the updated value and updates the selected words.  
 */  
mUpdateValues.putNull(UserDictionary.Words.LOCALE);  
  
mRowsUpdated = getContentResolver().update(  
    UserDictionary.Words.CONTENT_URI, // the user dictionary content URI  
    mUpdateValues // the columns to update  
    mSelectionClause // the column to select on  
    mSelectionArgs // the value to compare to  
);
```

You should also sanitize user input when you call [ContentResolver.update\(\)](#). To learn more about this, read the section [Protecting against malicious input](#).

## Deleting data

Deleting rows is similar to retrieving row data: you specify selection criteria for the rows you want to delete and the client method returns the number of deleted rows. The following snippet deletes rows whose appid matches "user". The method returns the number of deleted rows.

```
// Defines selection criteria for the rows you want to delete
String mSelectionClause = UserDictionary.Words.APP_ID + " LIKE ?";
String[] mSelectionArgs = {"user"};

// Defines a variable to contain the number of rows deleted
int mRowsDeleted = 0;

...

// Deletes the words that match the selection criteria
mRowsDeleted = getContentResolver().delete(
    UserDictionary.Words.CONTENT_URI,      // the user dictionary content URI
    mSelectionClause                      // the column to select on
    mSelectionArgs                        // the value to compare to
);
```

You should also sanitize user input when you call [ContentResolver.delete\(\)](#). To learn more about this, read the section [Protecting against malicious input](#).

## Provider Data Types

Content providers can offer many different data types. The User Dictionary Provider offers only text, but providers can also offer the following formats:

- integer
- long integer (long)
- floating point
- long floating point (double)

Another data type that providers often use is Binary Large OBject (BLOB) implemented as a 64KB byte array. You can see the available data types by looking at the [Cursor](#) class "get" methods.

The data type for each column in a provider is usually listed in its documentation. The data types for the User Dictionary Provider are listed in the reference documentation for its contract class [UserDictionary.Words](#) (contract classes are described in the section [Contract Classes](#)). You can also determine the data type by calling [Cursor.getType\(\)](#).

Providers also maintain MIME data type information for each content URI they define. You can use the MIME type information to find out if your application can handle data that the provider offers, or to choose a type of handling based on the MIME type. You usually need the MIME type when you are working with a provider that contains complex data structures or files. For example, the [ContactsContract.Data](#) table in the Contacts Provider uses MIME types to label the type of contact data stored in each row. To get the MIME type corresponding to a content URI, call [ContentResolver.getType\(\)](#).

The section [MIME Type Reference](#) describes the syntax of both standard and custom MIME types.

## Alternative Forms of Provider Access

Three alternative forms of provider access are important in application development:

- [Batch access](#): You can create a batch of access calls with methods in the [ContentProviderOperation](#) class, and then apply them with [ContentResolver.applyBatch\(\)](#).
- Asynchronous queries: You should do queries in a separate thread. One way to do this is to use a [CursorLoader](#) object. The examples in the [Loaders](#) guide demonstrate how to do this.
- [Data access via intents](#): Although you can't send an intent directly to a provider, you can send an intent to the provider's application, which is usually the best-equipped to modify the provider's data.

Batch access and modification via intents are described in the following sections.

### Batch access

Batch access to a provider is useful for inserting a large number of rows, or for inserting rows in multiple tables in the same method call, or in general for performing a set of operations across process boundaries as a transaction (an atomic operation).

To access a provider in "batch mode", you create an array of [ContentProviderOperation](#) objects and then dispatch them to a content provider with [ContentResolver.applyBatch\(\)](#). You pass the content provider's *authority* to this method, rather than a particular content URI. This allows each [ContentProviderOperation](#) object in the array to work against a different table. A call to [ContentResolver.applyBatch\(\)](#) returns an array of results.

The description of the [ContactsContract.RawContacts](#) contract class includes a code snippet that demonstrates batch insertion. The [Contact Manager](#) sample application contains an example of batch access in its ContactAdder.java source file.

## Displaying data using a helper app

If your application *does* have access permissions, you still may want to use an intent to display data in another application. For example, the Calendar application accepts an [ACTION\\_VIEW](#) intent, which displays a particular date or event. This allows you to display calendar information without having to create your own UI. To learn more about this feature, see the [Calendar Provider](#) guide.

The application to which you send the intent doesn't have to be the application associated with the provider. For example, you can retrieve a contact from the Contact Provider, then send an [ACTION\\_VIEW](#) intent containing the content URI for the contact's image to an image viewer.

### Data access via intents

Intents can provide indirect access to a content provider. You allow the user to access data in a provider even if your application doesn't have access permissions, either by getting a result intent back from an application that has permissions, or by activating an application that has permissions and letting the user do work in it.

#### Getting access with temporary permissions

You can access data in a content provider, even if you don't have the proper access permissions, by sending an intent to an application that does have the permissions and receiving back a result intent containing "URI" per-

missions. These are permissions for a specific content URI that last until the activity that receives them is finished. The application that has permanent permissions grants temporary permissions by setting a flag in the result intent:

- **Read permission:** [FLAG\\_GRANT\\_READ\\_URI\\_PERMISSION](#)
- **Write permission:** [FLAG\\_GRANT\\_WRITE\\_URI\\_PERMISSION](#)

**Note:** These flags don't give general read or write access to the provider whose authority is contained in the content URI. The access is only for the URI itself.

A provider defines URI permissions for content URIs in its manifest, using the [android:grantUriPermission](#) attribute of the [<provider>](#) element, as well as the [<grant-uri-permission>](#) child element of the [<provider>](#) element. The URI permissions mechanism is explained in more detail in the [Security and Permissions](#) guide, in the section "URI Permissions".

For example, you can retrieve data for a contact in the Contacts Provider, even if you don't have the [READ\\_CONTACTS](#) permission. You might want to do this in an application that sends e-greetings to a contact on his or her birthday. Instead of requesting [READ\\_CONTACTS](#), which gives you access to all of the user's contacts and all of their information, you prefer to let the user control which contacts are used by your application. To do this, you use the following process:

1. Your application sends an intent containing the action [ACTION\\_PICK](#) and the "contacts" MIME type [CONTENT\\_ITEM\\_TYPE](#), using the method [startActivityForResult\(\)](#).
2. Because this intent matches the intent filter for the People app's "selection" activity, the activity will come to the foreground.
3. In the selection activity, the user selects a contact to update. When this happens, the selection activity calls [setResult\(resultcode, intent\)](#) to set up a intent to give back to your application. The intent contains the content URI of the contact the user selected, and the "extras" flags [FLAG\\_GRANT\\_READ\\_URI\\_PERMISSION](#). These flags grant URI permission to your app to read data for the contact pointed to by the content URI. The selection activity then calls [finish\(\)](#) to return control to your application.
4. Your activity returns to the foreground, and the system calls your activity's [onActivityResult\(\)](#) method. This method receives the result intent created by the selection activity in the People app.
5. With the content URI from the result intent, you can read the contact's data from the Contacts Provider, even though you didn't request permanent read access permission to the provider in your manifest. You can then get the contact's birthday information or his or her email address and then send the e-greeting.

## Using another application

A simple way to allow the user to modify data to which you don't have access permissions is to activate an application that has permissions and let the user do the work there.

For example, the Calendar application accepts an [ACTION\\_INSERT](#) intent, which allows you to activate the application's insert UI. You can pass "extras" data in this intent, which the application uses to pre-populate the UI. Because recurring events have a complex syntax, the preferred way of inserting events into the Calendar Provider is to activate the Calendar app with an [ACTION\\_INSERT](#) and then let the user insert the event there.

## Contract Classes

A contract class defines constants that help applications work with the content URIs, column names, intent actions, and other features of a content provider. Contract classes are not included automatically with a provider; the provider's developer has to define them and then make them available to other developers. Many of the

providers included with the Android platform have corresponding contract classes in the package [android.provider](#).

For example, the User Dictionary Provider has a contract class [UserDictionary](#) containing content URI and column name constants. The content URI for the "words" table is defined in the constant [UserDictionary.Words.CONTENT\\_URI](#). The [UserDictionary.Words](#) class also contains column name constants, which are used in the example snippets in this guide. For example, a query projection can be defined as:

```
String[] mProjection =  
{  
    UserDictionary.Words._ID,  
    UserDictionary.Words.WORD,  
    UserDictionary.Words.LOCALE  
};
```

Another contract class is [ContactsContract](#) for the Contacts Provider. The reference documentation for this class includes example code snippets. One of its subclasses, [ContactsContract.Insert](#), is a contract class that contains constants for intents and intent data.

## MIME Type Reference

Content providers can return standard MIME media types, or custom MIME type strings, or both.

MIME types have the format

*type*/*subtype*

For example, the well-known MIME type `text/html` has the `text` type and the `html` subtype. If the provider returns this type for a URI, it means that a query using that URI will return text containing HTML tags.

Custom MIME type strings, also called "vendor-specific" MIME types, have more complex *type* and *subtype* values. The *type* value is always

`vnd.android.cursor.dir`

for multiple rows, or

`vnd.android.cursor.item`

for a single row.

The *subtype* is provider-specific. The Android built-in providers usually have a simple subtype. For example, the when the Contacts application creates a row for a telephone number, it sets the following MIME type in the row:

`vnd.android.cursor.item/phone_v2`

Notice that the subtype value is simply `phone_v2`.

Other provider developers may create their own pattern of subtypes based on the provider's authority and table names. For example, consider a provider that contains train timetables. The provider's authority is `com.example.trains`, and it contains the tables `Line1`, `Line2`, and `Line3`. In response to the content URI

```
content://com.example.trains/Line1
```

for table Line1, the provider returns the MIME type

```
vnd.android.cursor.dir/vnd.example.line1
```

In response to the content URI

```
content://com.example.trains/Line2/5
```

for row 5 in table Line2, the provider returns the MIME type

```
vnd.android.cursor.item/vnd.example.line2
```

Most content providers define contract class constants for the MIME types they use. The Contacts Provider contract class [ContactsContract.RawContacts](#), for example, defines the constant [CONTENT\\_ITEM\\_TYPE](#) for the MIME type of a single raw contact row.

Content URIs for single rows are described in the section [Content URIs](#).

# Creating a Content Provider

## In this document

1. [Designing Data Storage](#)
2. [Designing Content URIs](#)
3. [Implementing the ContentProvider Class](#)
  1. [Required Methods](#)
  2. [Implementing the query\(\) method](#)
  3. [Implementing the insert\(\) method](#)
  4. [Implementing the delete\(\) method](#)
  5. [Implementing the update\(\) method](#)
  6. [Implementing the onCreate\(\) method](#)
4. [Implementing Content Provider MIME Types](#)
  1. [MIME types for tables](#)
  2. [MIME types for files](#)
5. [Implementing a Contract Class](#)
6. [Implementing Content Provider Permissions](#)
7. [The <provider> Element](#)
8. [Intents and Data Access](#)

## Key classes

1. [ContentProvider](#)
2. [Cursor](#)
3. [Uri](#)

## Related Samples

1. [Note Pad sample application](#)

## See also

1. [Content Provider Basics](#)
2. [Calendar Provider](#)

A content provider manages access to a central repository of data. You implement a provider as one or more classes in an Android application, along with elements in the manifest file. One of your classes implements a subclass [ContentProvider](#), which is the interface between your provider and other applications. Although content providers are meant to make data available to other applications, you may of course have activities in your application that allow the user to query and modify the data managed by your provider.

The rest of this topic is a basic list of steps for building a content provider and a list of APIs to use.

## Before You Start Building

Before you start building a provider, do the following:

1. **Decide if you need a content provider.** You need to build a content provider if you want to provide one or more of the following features:

- You want to offer complex data or files to other applications.
- You want to allow users to copy complex data from your app into other apps.
- You want to provide custom search suggestions using the search framework.

You *don't* need a provider to use an SQLite database if the use is entirely within your own application.

2. If you haven't done so already, read the topic [Content Provider Basics](#) to learn more about providers.

Next, follow these steps to build your provider:

1. Design the raw storage for your data. A content provider offers data in two ways:

#### File data

Data that normally goes into files, such as photos, audio, or videos. Store the files in your application's private space. In response to a request for a file from another application, your provider can offer a handle to the file.

#### "Structured" data

Data that normally goes into a database, array, or similar structure. Store the data in a form that's compatible with tables of rows and columns. A row represents an entity, such as a person or an item in inventory. A column represents some data for the entity, such as a person's name or an item's price. A common way to store this type of data is in an SQLite database, but you can use any type of persistent storage. To learn more about the storage types available in the Android system, see the section [Designing Data Storage](#).

2. Define a concrete implementation of the [ContentProvider](#) class and its required methods. This class is the interface between your data and the rest of the Android system. For more information about this class, see the section [Implementing the ContentProvider Class](#).
3. Define the provider's authority string, its content URIs, and column names. If you want the provider's application to handle intents, also define intent actions, extras data, and flags. Also define the permissions that you will require for applications that want to access your data. You should consider defining all of these values as constants in a separate contract class; later, you can expose this class to other developers. For more information about content URIs, see the section [Designing Content URIs](#). For more information about intents, see the section [Intents and Data Access](#).
4. Add other optional pieces, such as sample data or an implementation of [AbstractThreadedSyncAdapter](#) that can synchronize data between the provider and cloud-based data.

## Designing Data Storage

A content provider is the interface to data saved in a structured format. Before you create the interface, you must decide how to store the data. You can store the data in any form you like, and then design the interface to read and write the data as necessary.

These are some of the data storage technologies that are available in Android:

- The Android system includes an SQLite database API that Android's own providers use to store table-oriented data. The [SQLiteOpenHelper](#) class helps you create databases, and the [SQLiteOpenHelper](#) class is the base class for accessing databases.

Remember that you don't have to use a database to implement your repository. A provider appears externally as a set of tables, similar to a relational database, but this is not a requirement for the provider's internal implementation.

- For storing file data, Android has a variety of file-oriented APIs. To learn more about file storage, read the topic [Data Storage](#). If you're designing a provider that offers media-related data such as music or videos, you can have a provider that combines table data and files.
- For working with network-based data, use classes in [java.net](#) and [android.net](#). You can also synchronize network-based data to a local data store such as a database, and then offer the data as tables or files. The [Sample Sync Adapter](#) sample application demonstrates this type of synchronization.

## Data design considerations

Here are some tips for designing your provider's data structure:

- Table data should always have a "primary key" column that the provider maintains as a unique numeric value for each row. You can use this value to link the row to related rows in other tables (using it as a "foreign key"). Although you can use any name for this column, using [BaseColumns.ID](#) is the best choice, because linking the results of a provider query to a [ListView](#) requires one of the retrieved columns to have the name `_ID`.
- If you want to provide bitmap images or other very large pieces of file-oriented data, store the data in a file and then provide it indirectly rather than storing it directly in a table. If you do this, you need to tell users of your provider that they need to use a [ContentResolver](#) file method to access the data.
- Use the Binary Large OBject (BLOB) data type to store data that varies in size or has a varying structure. For example, you can use a BLOB column to store a [protocol buffer](#) or [JSON structure](#).

You can also use a BLOB to implement a *schema-independent* table. In this type of table, you define a primary key column, a MIME type column, and one or more generic columns as BLOB. The meaning of the data in the BLOB columns is indicated by the value in the MIME type column. This allows you to store different row types in the same table. The Contacts Provider's "data" table [ContactsContract.Data](#) is an example of a schema-independent table.

## Designing Content URIs

A **content URI** is a URI that identifies data in a provider. Content URIs include the symbolic name of the entire provider (its **authority**) and a name that points to a table or file (a **path**). The optional id part points to an individual row in a table. Every data access method of [ContentProvider](#) has a content URI as an argument; this allows you to determine the table, row, or file to access.

The basics of content URIs are described in the topic [Content Provider Basics](#).

### Designing an authority

A provider usually has a single authority, which serves as its Android-internal name. To avoid conflicts with other providers, you should use Internet domain ownership (in reverse) as the basis of your provider authority. Because this recommendation is also true for Android package names, you can define your provider authority as an extension of the name of the package containing the provider. For example, if your Android package name is `com.example.<appname>`, you should give your provider the authority `com.example.<appname>.provider`.

### Designing a path structure

Developers usually create content URIs from the authority by appending paths that point to individual tables. For example, if you have two tables `table1` and `table2`, you combine the authority from the previous example to yield the content URIs `com.example.<appname>.provider/table1` and `com.example.<appname>.provider/table2`. Paths aren't limited to a single segment, and there doesn't have to be a table for each level of the path.

## Handling content URI IDs

By convention, providers offer access to a single row in a table by accepting a content URI with an ID value for the row at the end of the URI. Also by convention, providers match the ID value to the table's `_ID` column, and perform the requested access against the row that matches.

This convention facilitates a common design pattern for apps accessing a provider. The app does a query against the provider and displays the resulting [Cursor](#) in a [ListView](#) using a [CursorAdapter](#). The definition of [CursorAdapter](#) requires one of the columns in the [Cursor](#) to be `_ID`.

The user then picks one of the displayed rows from the UI in order to look at or modify the data. The app gets the corresponding row from the [Cursor](#) backing the [ListView](#), gets the `_ID` value for this row, appends it to the content URI, and sends the access request to the provider. The provider can then do the query or modification against the exact row the user picked.

## Content URI patterns

To help you choose which action to take for an incoming content URI, the provider API includes the convenience class [UriMatcher](#), which maps content URI "patterns" to integer values. You can use the integer values in a `switch` statement that chooses the desired action for the content URI or URIs that match a particular pattern.

A content URI pattern matches content URIs using wildcard characters:

- `*`: Matches a string of any valid characters of any length.
- `#`: Matches a string of numeric characters of any length.

As an example of designing and coding content URI handling, consider a provider with the authority `com.example.app.provider` that recognizes the following content URIs pointing to tables:

- `content://com.example.app.provider/table1`: A table called `table1`.
- `content://com.example.app.provider/table2/dataset1`: A table called `dataset1`.
- `content://com.example.app.provider/table2/dataset2`: A table called `dataset2`.
- `content://com.example.app.provider/table3`: A table called `table3`.

The provider also recognizes these content URIs if they have a row ID appended to them, as for example `content://com.example.app.provider/table3/1` for the row identified by 1 in `table3`.

The following content URI patterns would be possible:

**`content://com.example.app.provider/*`**

Matches any content URI in the provider.

**`content://com.example.app.provider/table2/*:`**

Matches a content URI for the tables `dataset1` and `dataset2`, but doesn't match content URIs for `table1` or `table3`.

**`content://com.example.app.provider/table3/#:`** Matches a content URI for single rows in `table3`, such as `content://com.example.app.provider/table3/6` for the row identified by 6.

The following code snippet shows how the methods in [UriMatcher](#) work. This code handles URIs for an entire table differently from URIs for a single row, by using the content URI pattern `content://com.example.app.provider/*`.

`tent://<authority>/<path>` for tables, and `content://<authority>/<path>/<id>` for single rows.

The method `addURI()` maps an authority and path to an integer value. The method `match()` returns the integer value for a URI. A switch statement chooses between querying the entire table, and querying for a single record:

```
public class ExampleProvider extends ContentProvider {  
    ...  
    // Creates a UriMatcher object.  
    private static final UriMatcher sUriMatcher;  
    ...  
    /*  
     * The calls to addURI() go here, for all of the content URI patterns that  
     * should recognize. For this snippet, only the calls for table 3 are shown  
     */  
    ...  
    /*  
     * Sets the integer value for multiple rows in table 3 to 1. Notice that no  
     * in the path  
     */  
    sUriMatcher.addURI("com.example.app.provider", "table3", 1);  
  
    /*  
     * Sets the code for a single row to 2. In this case, the "#" wildcard is  
     * used. "content://com.example.app.provider/table3/3" matches, but  
     * "content://com.example.app.provider/table3" doesn't.  
     */  
    sUriMatcher.addURI("com.example.app.provider", "table3/#", 2);  
    ...  
    // Implements ContentProvider.query()  
    public Cursor query(  
        Uri uri,  
        String[] projection,  
        String selection,  
        String[] selectionArgs,  
        String sortOrder) {  
    ...  
        /*  
         * Choose the table to query and a sort order based on the code returned  
         * by UriMatcher. Here, too, only the statements for table 3 are shown.  
         */  
        switch (sUriMatcher.match(uri)) {  
  
            // If the incoming URI was for all of table3  
            case 1:  
  
                if (TextUtils.isEmpty(sortOrder)) sortOrder = "_ID ASC";  
                break;  
  
                // If the incoming URI was for a single row  
            case 2:  
        }  
    }  
}
```

```

/*
 * Because this URI was for a single row, the _ID value part is
 * present. Get the last path segment from the URI; this is the
 * Then, append the value to the WHERE clause for the query
 */
selection = selection + "_ID = " uri.getLastPathSegment();
break;

default:
...
// If the URI is not recognized, you should do some error handling
}
// call the code to actually do the query
}

```

Another class, [ContentUris](#), provides convenience methods for working with the `id` part of content URIs. The classes [Uri](#) and [Uri.Builder](#) include convenience methods for parsing existing [Uri](#) objects and building new ones.

## Implementing the ContentProvider Class

The [ContentProvider](#) instance manages access to a structured set of data by handling requests from other applications. All forms of access eventually call [ContentResolver](#), which then calls a concrete method of [ContentProvider](#) to get access.

### Required methods

The abstract class [ContentProvider](#) defines six abstract methods that you must implement as part of your own concrete subclass. All of these methods except [onCreate\(\)](#) are called by a client application that is attempting to access your content provider:

#### [query\(\)](#)

Retrieve data from your provider. Use the arguments to select the table to query, the rows and columns to return, and the sort order of the result. Return the data as a [Cursor](#) object.

#### [insert\(\)](#)

Insert a new row into your provider. Use the arguments to select the destination table and to get the column values to use. Return a content URI for the newly-inserted row.

#### [update\(\)](#)

Update existing rows in your provider. Use the arguments to select the table and rows to update and to get the updated column values. Return the number of rows updated.

#### [delete\(\)](#)

Delete rows from your provider. Use the arguments to select the table and the rows to delete. Return the number of rows deleted.

#### [getType\(\)](#)

Return the MIME type corresponding to a content URI. This method is described in more detail in the section [Implementing Content Provider MIME Types](#).

## [onCreate\(\)](#)

Initialize your provider. The Android system calls this method immediately after it creates your provider. Notice that your provider is not created until a [ContentResolver](#) object tries to access it.

Notice that these methods have the same signature as the identically-named [ContentResolver](#) methods.

Your implementation of these methods should account for the following:

- All of these methods except [onCreate\(\)](#) can be called by multiple threads at once, so they must be thread-safe. To learn more about multiple threads, see the topic [Processes and Threads](#).
- Avoid doing lengthy operations in [onCreate\(\)](#). Defer initialization tasks until they are actually needed. The section [Implementing the onCreate\(\) method](#) discusses this in more detail.
- Although you must implement these methods, your code does not have to do anything except return the expected data type. For example, you may want to prevent other applications from inserting data into some tables. To do this, you can ignore the call to [insert\(\)](#) and return 0.

## **Implementing the query() method**

The [ContentProvider.query\(\)](#) method must return a [Cursor](#) object, or if it fails, throw an [Exception](#). If you are using an SQLite database as your data storage, you can simply return the [Cursor](#) returned by one of the [query\(\)](#) methods of the [SQLiteDatabase](#) class. If the query does not match any rows, you should return a [Cursor](#) instance whose [getCount\(\)](#) method returns 0. You should return `null` only if an internal error occurred during the query process.

If you aren't using an SQLite database as your data storage, use one of the concrete subclasses of [Cursor](#). For example, the [MatrixCursor](#) class implements a cursor in which each row is an array of [Object](#). With this class, use [addRow\(\)](#) to add a new row.

Remember that the Android system must be able to communicate the [Exception](#) across process boundaries. Android can do this for the following exceptions that may be useful in handling query errors:

- [IllegalArgumentException](#) (You may choose to throw this if your provider receives an invalid content URI)
- [NullPointerException](#)

## **Implementing the insert() method**

The [insert\(\)](#) method adds a new row to the appropriate table, using the values in the [ContentValues](#) argument. If a column name is not in the [ContentValues](#) argument, you may want to provide a default value for it either in your provider code or in your database schema.

This method should return the content URI for the new row. To construct this, append the new row's `_ID` (or other primary key) value to the table's content URI, using [withAppendedId\(\)](#).

## **Implementing the delete() method**

The [delete\(\)](#) method does not have to physically delete rows from your data storage. If you are using a sync adapter with your provider, you should consider marking a deleted row with a "delete" flag rather than removing the row entirely. The sync adapter can check for deleted rows and remove them from the server before deleting them from the provider.

## Implementing the update() method

The [update\(\)](#) method takes the same [ContentValues](#) argument used by [insert\(\)](#), and the same selection and selectionArgs arguments used by [delete\(\)](#) and [ContentProvider.query\(\)](#). This may allow you to re-use code between these methods.

## Implementing the onCreate() method

The Android system calls [onCreate\(\)](#) when it starts up the provider. You should perform only fast-running initialization tasks in this method, and defer database creation and data loading until the provider actually receives a request for the data. If you do lengthy tasks in [onCreate\(\)](#), you will slow down your provider's startup. In turn, this will slow down the response from the provider to other applications.

For example, if you are using an SQLite database you can create a new [SQLiteOpenHelper](#) object in [ContentProvider.onCreate\(\)](#), and then create the SQL tables the first time you open the database. To facilitate this, the first time you call [getWritableDatabase\(\)](#), it automatically calls the [SQLiteOpenHelper.onCreate\(\)](#) method.

The following two snippets demonstrate the interaction between [ContentProvider.onCreate\(\)](#) and [SQLiteOpenHelper.onCreate\(\)](#). The first snippet is the implementation of [ContentProvider.onCreate\(\)](#):

```
public class ExampleProvider extends ContentProvider {

    /**
     * Defines a handle to the database helper object. The MainDatabaseHelper
     * is defined in the following snippet.
     */
    private MainDatabaseHelper mOpenHelper;

    // Defines the database name
    private static final String DBNAME = "mydb";

    // Holds the database object
    private SQLiteDatabase db;

    public boolean onCreate() {

        /**
         * Creates a new helper object. This method always returns quickly.
         * Notice that the database itself isn't created or opened
         * until SQLiteOpenHelper.getWritableDatabase is called
         */
        mOpenHelper = new SQLiteOpenHelper(
            getContext(),           // the application context
            DBNAME,                 // the name of the database)
            null,                   // uses the default SQLite cursor
            1                      // the version number
        );

        return true;
    }
}
```

```

...
// Implements the provider's insert method
public Cursor insert(Uri uri, ContentValues values) {
    // Insert code here to determine which table to open, handle error-checking
    ...

    /*
     * Gets a writeable database. This will trigger its creation if it does
     *
     */
    db = mOpenHelper.getWritableDatabase();
}

}

```

The next snippet is the implementation of [SQLiteOpenHelper.onCreate\(\)](#), including a helper class:

```

...
// A string that defines the SQL statement for creating a table
private static final String SQL_CREATE_MAIN = "CREATE TABLE " +
    "main " +                                // Table's name
    "(" +                                     // The columns in the table
    " _ID INTEGER PRIMARY KEY, " +
    " WORD TEXT" +
    " FREQUENCY INTEGER " +
    " LOCALE TEXT )";

...
/***
 * Helper class that actually creates and manages the provider's underlying database
 */
protected static final class MainDatabaseHelper extends SQLiteOpenHelper {

    /*
     * Instantiates an open helper for the provider's SQLite data repository
     * Do not do database creation and upgrade here.
     */
    MainDatabaseHelper(Context context) {
        super(context, DBNAME, null, 1);
    }

    /*
     * Creates the data repository. This is called when the provider attempts to
     * repository and SQLite reports that it doesn't exist.
     */
    public void onCreate(SQLiteDatabase db) {

        // Creates the main table
        db.execSQL(SQL_CREATE_MAIN);
    }
}
```

# Implementing ContentProvider MIME Types

The [ContentProvider](#) class has two methods for returning MIME types:

## [getType \(\)](#)

One of the required methods that you must implement for any provider.

## [getStreamTypes \(\)](#)

A method that you're expected to implement if your provider offers files.

## MIME types for tables

The [getType \(\)](#) method returns a [String](#) in MIME format that describes the type of data returned by the content URI argument. The [Uri](#) argument can be a pattern rather than a specific URI; in this case, you should return the type of data associated with content URIs that match the pattern.

For common types of data such as as text, HTML, or JPEG, [getType \(\)](#) should return the standard MIME type for that data. A full list of these standard types is available on the [IANA MIME Media Types](#) website.

For content URIs that point to a row or rows of table data, [getType \(\)](#) should return a MIME type in Android's vendor-specific MIME format:

- Type part: vnd
- Subtype part:
  - If the URI pattern is for a single row: `android.cursor.item/`
  - If the URI pattern is for more than one row: `android.cursor.dir/`
- Provider-specific part: `vnd.<name>.<type>`

You supply the `<name>` and `<type>`. The `<name>` value should be globally unique, and the `<type>` value should be unique to the corresponding URI pattern. A good choice for `<name>` is your company's name or some part of your application's Android package name. A good choice for the `<type>` is a string that identifies the table associated with the URI.

For example, if a provider's authority is `com.example.app.provider`, and it exposes a table named `table1`, the MIME type for multiple rows in `table1` is:

```
vnd.android.cursor.dir/vnd.com.example.provider.table1
```

For a single row of `table1`, the MIME type is:

```
vnd.android.cursor.item/vnd.com.example.provider.table1
```

## MIME types for files

If your provider offers files, implement [getStreamTypes \(\)](#). The method returns a [String](#) array of MIME types for the files your provider can return for a given content URI. You should filter the MIME types you offer by the MIME type filter argument, so that you return only those MIME types that the client wants to handle.

For example, consider a provider that offers photo images as files in `.jpg`, `.png`, and `.gif` format. If an application calls [ContentResolver.getStreamTypes \(\)](#) with the filter string `image/*` (something that is an "image"), then the [ContentProvider.getStreamTypes \(\)](#) method should return the array:

```
{ "image/jpeg", "image/png", "image/gif" }
```

If the app is only interested in .jpg files, then it can call [ContentResolver.getStreamTypes\(\)](#) with the filter string \*\\jpeg, and [ContentProvider.getStreamTypes\(\)](#) should return:

```
{"image/jpeg"}
```

If your provider doesn't offer any of the MIME types requested in the filter string, [getStreamTypes\(\)](#) should return null.

## Implementing a Contract Class

A contract class is a `public final` class that contains constant definitions for the URIs, column names, MIME types, and other meta-data that pertain to the provider. The class establishes a contract between the provider and other applications by ensuring that the provider can be correctly accessed even if there are changes to the actual values of URIs, column names, and so forth.

A contract class also helps developers because it usually has mnemonic names for its constants, so developers are less likely to use incorrect values for column names or URIs. Since it's a class, it can contain Javadoc documentation. Integrated development environments such as Eclipse can auto-complete constant names from the contract class and display Javadoc for the constants.

Developers can't access the contract class's class file from your application, but they can statically compile it into their application from a .jar file you provide.

The [ContactsContract](#) class and its nested classes are examples of contract classes.

## Implementing Content Provider Permissions

Permissions and access for all aspects of the Android system are described in detail in the topic [Security and Permissions](#). The topic [Data Storage](#) also describes the security and permissions in effect for various types of storage. In brief, the important points are:

- By default, data files stored on the device's internal storage are private to your application and provider.
- [SQLiteDatabase](#) databases you create are private to your application and provider.
- By default, data files that you save to external storage are *public* and *world-readable*. You can't use a content provider to restrict access to files in external storage, because other applications can use other API calls to read and write them.
- The method calls for opening or creating files or SQLite databases on your device's internal storage can potentially give both read and write access to all other applications. If you use an internal file or database as your provider's repository, and you give it "world-readable" or "world-writeable" access, the permissions you set for your provider in its manifest won't protect your data. The default access for files and databases in internal storage is "private", and for your provider's repository you shouldn't change this.

If you want to use content provider permissions to control access to your data, then you should store your data in internal files, SQLite databases, or the "cloud" (for example, on a remote server), and you should keep files and databases private to your application.

## Implementing permissions

All applications can read from or write to your provider, even if the underlying data is private, because by default your provider does not have permissions set. To change this, set permissions for your provider in your manifest file, using attributes or child elements of the [`<provider>`](#) element. You can set permissions that apply to the entire provider, or to certain tables, or even to certain records, or all three.

You define permissions for your provider with one or more [`<permission>`](#) elements in your manifest file. To make the permission unique to your provider, use Java-style scoping for the [`android:name`](#) attribute. For example, name the read permission `com.example.app.provider.permission.READ_PROVIDER`.

The following list describes the scope of provider permissions, starting with the permissions that apply to the entire provider and then becoming more fine-grained. More fine-grained permissions take precedence over ones with larger scope:

### Single read-write provider-level permission

One permission that controls both read and write access to the entire provider, specified with the [`an-`](#)  
[`android:permission`](#) attribute of the [`<provider>`](#) element.

### Separate read and write provider-level permission

A read permission and a write permission for the entire provider. You specify them with the [`an-`](#)  
[`android:readPermission`](#) and [`android:writePermission`](#) attributes of the [`<provider>`](#) ele-  
ment. They take precedence over the permission required by [`android:permission`](#).

### Path-level permission

Read, write, or read/write permission for a content URI in your provider. You specify each URI you want to control with a [`<path-permission>`](#) child element of the [`<provider>`](#) element. For each content URI you specify, you can specify a read/write permission, a read permission, or a write permission, or all three. The read and write permissions take precedence over the read/write permission. Also, path-level permission takes precedence over provider-level permissions.

### Temporary permission

A permission level that grants temporary access to an application, even if the application doesn't have the permissions that are normally required. The temporary access feature reduces the number of permissions an application has to request in its manifest. When you turn on temporary permissions, the only applications that need "permanent" permissions for your provider are ones that continually access all your data.

Consider the permissions you need to implement an email provider and app, when you want to allow an outside image viewer application to display photo attachments from your provider. To give the image viewer the necessary access without requiring permissions, set up temporary permissions for content URIs for photos. Design your email app so that when the user wants to display a photo, the app sends an intent containing the photo's content URI and permission flags to the image viewer. The image viewer can then query your email provider to retrieve the photo, even though the viewer doesn't have the normal read permission for your provider.

To turn on temporary permissions, either set the [`android:grantUriPermissions`](#) attribute of the [`<provider>`](#) element, or add one or more [`<grant-uri-permission>`](#) child elements to your [`<provider>`](#) element. If you use temporary permissions, you have to call [`Con-`](#)  
[`text.revokeUriPermission\(\)`](#) whenever you remove support for a content URI from your provider, and the content URI is associated with a temporary permission.

The attribute's value determines how much of your provider is made accessible. If the attribute is set to `true`, then the system will grant temporary permission to your entire provider, overriding any other permissions that are required by your provider-level or path-level permissions.

If this flag is set to `false`, then you must add [`<grant-uri-permission>`](#) child elements to your [`<provider>`](#) element. Each child element specifies the content URI or URIs for which temporary access is granted.

To delegate temporary access to an application, an intent must contain the [FLAG\\_GRANT\\_READ\\_URI\\_PERMISSION](#) or the [FLAG\\_GRANT\\_WRITE\\_URI\\_PERMISSION](#) flags, or both. These are set with the [setFlags\(\)](#) method.

If the [android:grantUriPermissions](#) attribute is not present, it's assumed to be `false`.

## The `<provider>` Element

Like [Activity](#) and [Service](#) components, a subclass of [ContentProvider](#) must be defined in the manifest file for its application, using the [<provider>](#) element. The Android system gets the following information from the element:

### Authority ([android:authorities](#))

Symbolic names that identify the entire provider within the system. This attribute is described in more detail in the section [Designing Content URIs](#).

### Provider class name ([android:name](#))

The class that implements [ContentProvider](#). This class is described in more detail in the section [Implementing the ContentProvider Class](#).

### Permissions

Attributes that specify the permissions that other applications must have in order to access the provider's data:

- [android:grantUriPermissions](#): Temporary permission flag.
- [android:permission](#): Single provider-wide read/write permission.
- [android:readPermission](#): Provider-wide read permission.
- [android:writePermission](#): Provider-wide write permission.

Permissions and their corresponding attributes are described in more detail in the section [Implementing Content Provider Permissions](#).

### Startup and control attributes

These attributes determine how and when the Android system starts the provider, the process characteristics of the provider, and other run-time settings:

- [android:enabled](#): Flag allowing the system to start the provider.
- [android:exported](#): Flag allowing other applications to use this provider.
- [android:initOrder](#): The order in which this provider should be started, relative to other providers in the same process.
- [android:multiProcess](#): Flag allowing the system to start the provider in the same process as the calling client.
- [android:process](#): The name of the process in which the provider should run.
- [android:syncable](#): Flag indicating that the provider's data is to be sync'ed with data on a server.

The attributes are fully documented in the dev guide topic for the [<provider>](#) element.

### Informational attributes

An optional icon and label for the provider:

- [android:icon](#): A drawable resource containing an icon for the provider. The icon appears next to the provider's label in the list of apps in *Settings > Apps > All*.
- [android:label](#): An informational label describing the provider or its data, or both. The label appears in the list of apps in *Settings > Apps > All*.

The attributes are fully documented in the dev guide topic for the [`<provider>`](#) element.

## Intents and Data Access

Applications can access a content provider indirectly with an [Intent](#). The application does not call any of the methods of [ContentResolver](#) or [ContentProvider](#). Instead, it sends an intent that starts an activity, which is often part of the provider's own application. The destination activity is in charge of retrieving and displaying the data in its UI. Depending on the action in the intent, the destination activity may also prompt the user to make modifications to the provider's data. An intent may also contain "extras" data that the destination activity displays in the UI; the user then has the option of changing this data before using it to modify the data in the provider.

You may want to use intent access to help ensure data integrity. Your provider may depend on having data inserted, updated, and deleted according to strictly defined business logic. If this is the case, allowing other applications to directly modify your data may lead to invalid data. If you want developers to use intent access, be sure to document it thoroughly. Explain to them why intent access using your own application's UI is better than trying to modify the data with their code.

Handling an incoming intent that wishes to modify your provider's data is no different from handling other intents. You can learn more about using intents by reading the topic [Intents and Intent Filters](#).

# Calendar Provider

## In this document

1. [Basics](#)
2. [User Permissions](#)
3. [Calendars table](#)
  1. [Querying a calendar](#)
  2. [Modifying a calendar](#)
  3. [Inserting a calendar](#)
4. [Events table](#)
  1. [Adding Events](#)
  2. [Updating Events](#)
  3. [Deleting Events](#)
5. [Attendees table](#)
  1. [Adding Attendees](#)
6. [Reminders table](#)
  1. [Adding Reminders](#)
7. [Instances table](#)
  1. [Querying the Instances table](#)
8. [Calendar Intents](#)
  1. [Using an intent to insert an event](#)
  2. [Using an intent to edit an event](#)
  3. [Using intents to view calendar data](#)
9. [Sync Adapters](#)

## Key classes

1. [CalendarContract.Calendars](#)
2. [CalendarContract.Events](#)
3. [CalendarContract.Attendees](#)
4. [CalendarContract.Reminders](#)

The Calendar Provider is a repository for a user's calendar events. The Calendar Provider API allows you to perform query, insert, update, and delete operations on calendars, events, attendees, reminders, and so on.

The Calender Provider API can be used by applications and sync adapters. The rules vary depending on what type of program is making the calls. This document focuses primarily on using the Calendar Provider API as an application. For a discussion of how sync adapters are different, see [Sync Adapters](#).

Normally, to read or write calendar data, an application's manifest must include the proper permissions, described in [User Permissions](#). To make performing common operations easier, the Calendar Provider offers a set of intents, as described in [Calendar Intents](#). These intents take users to the Calendar application to insert, view, and edit events. The user interacts with the Calendar application and then returns to the original application. Thus your application doesn't need to request permissions, nor does it need to provide a user interface to view or create events.

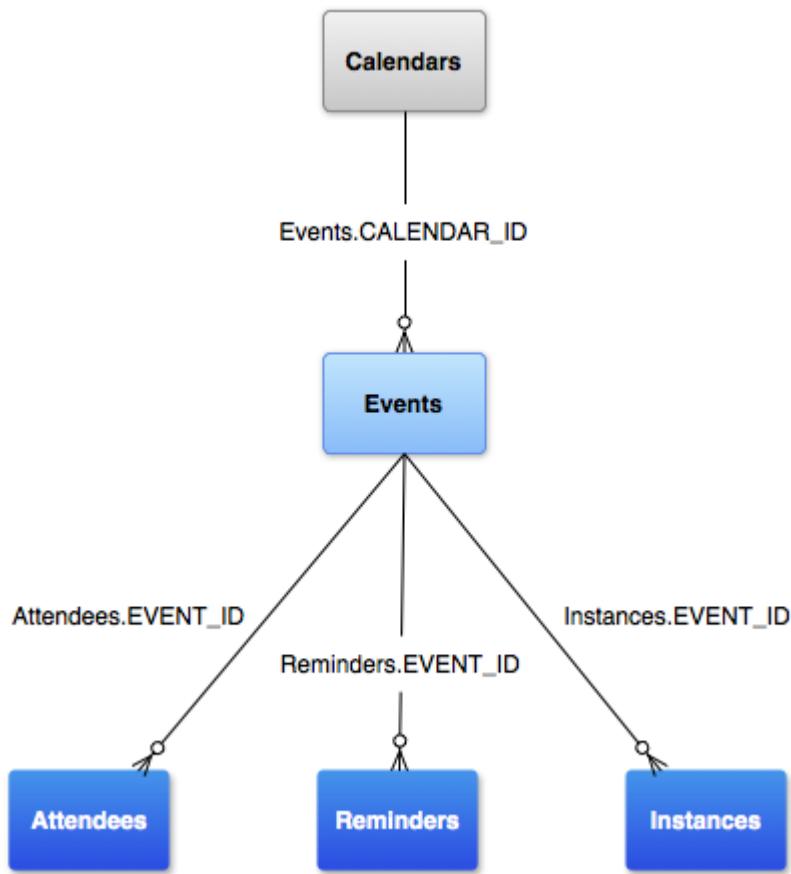
## Basics

[Content providers](#) store data and make it accessible to applications. The content providers offered by the Android platform (including the Calendar Provider) typically expose data as a set of tables based on a relational

database model, where each row is a record and each column is data of a particular type and meaning. Through the Calendar Provider API, applications and sync adapters can get read/write access to the database tables that hold a user's calendar data.

Every content provider exposes a public URI (wrapped as a [Uri](#) object) that uniquely identifies its data set. A content provider that controls multiple data sets (multiple tables) exposes a separate URI for each one. All URIs for providers begin with the string "content://" . This identifies the data as being controlled by a content provider. The Calendar Provider defines constants for the URIs for each of its classes (tables). These URIs have the format <class>.CONTENT\_URI . For example, [Events.CONTENT\\_URI](#).

Figure 1 shows a graphical representation of the Calendar Provider data model. It shows the main tables and the fields that link them to each other.



**Figure 1.** Calendar Provider data model.

A user can have multiple calendars, and different calendars can be associated with different types of accounts (Google Calendar, Exchange, and so on).

The [CalendarContract](#) defines the data model of calendar and event related information. This data is stored in a number of tables, listed below.

Table (Class)	Description
<a href="#">CalendarContract.Calendars</a>	This table holds the calendar-specific information. Each row in this table contains the details for a single calendar, such as the name, color, sync information, and so on.
<a href="#">CalendarContract.Events</a>	This table holds the event-specific information. Each row in this table has the information for a single event—for example, event title, location, start time, end time, and so on. The event can occur one-time or can recur multiple times. Attendees, reminders, and extended properties

are stored in separate tables. They each have an [EVENT\\_ID](#) that references the [\\_ID](#) in the Events table.

This table holds the start and end time for each occurrence of an event. Each row in this table represents a single event occurrence. For one-time events there is a 1:1 mapping of instances to events. For recurring events, multiple rows are automatically generated that correspond to multiple occurrences of that event.

#### [CalendarContract Instances](#)

This table holds the event attendee (guest) information. Each row represents a single guest of an event. It specifies the type of guest and the guest's attendance response for the event.

#### [CalendarContract Attendees](#)

This table holds the alert/notification data. Each row represents a single alert for an event. An event can have multiple reminders. The maximum number of reminders per event is specified in [MAX REMINDERS](#), which is set by the sync adapter that owns the given calendar. Reminders are specified in minutes before the event and have a method that determines how the user will be alerted.

#### [CalendarContract Reminders](#)

The Calendar Provider API is designed to be flexible and powerful. At the same time, it's important to provide a good end user experience and protect the integrity of the calendar and its data. To this end, here are some things to keep in mind when using the API:

- **Inserting, updating, and viewing calendar events.** To directly insert, modify, and read events from the Calendar Provider, you need the appropriate [permissions](#). However, if you're not building a full-fledged calendar application or sync adapter, requesting these permissions isn't necessary. You can instead use intents supported by Android's Calendar application to hand off read and write operations to that application. When you use the intents, your application sends users to the Calendar application to perform the desired operation in a pre-filled form. After they're done, they're returned to your application. By designing your application to perform common operations through the Calendar, you provide users with a consistent, robust user interface. This is the recommended approach. For more information, see [Calendar Intents](#).
- **Sync adapters.** A sync adapter synchronizes the calendar data on a user's device with another server or data source. In the [CalendarContract Calendars](#) and [CalendarContract Events](#) tables, there are columns that are reserved for the sync adapters to use. The provider and applications should not modify them. In fact, they are not visible unless they are accessed as a sync adapter. For more information about sync adapters, see [Sync Adapters](#).

## User Permissions

To read calendar data, an application must include the [READ CALENDAR](#) permission in its manifest file. It must include the [WRITE CALENDAR](#) permission to delete, insert or update calendar data:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"...
    <uses-sdk android:minSdkVersion="14" />
    <uses-permission android:name="android.permission.READ_CALENDAR" />
    <uses-permission android:name="android.permission.WRITE_CALENDAR" />
    ...
</manifest>
```

# Calendars Table

The [CalendarContract.Calendars](#) table contains details for individual calendars. The following Calendars columns are writable by both an application and a [sync adapter](#). For a full list of supported fields, see the [CalendarContract.Calendars](#) reference.

Constant	Description
<a href="#">NAME</a>	The name of the calendar.
<a href="#">CALENDAR_DISPLAY_NAME</a>	The name of this calendar that is displayed to the user.
<a href="#">VISIBLE</a>	A boolean indicating whether the calendar is selected to be displayed. A value of 0 indicates that events associated with this calendar should not be shown. A value of 1 indicates that events associated with this calendar should be shown. This value affects the generation of rows in the <a href="#">CalendarContract.Instances</a> table.
<a href="#">SYNC_EVENTS</a>	A boolean indicating whether the calendar should be synced and have its events stored on the device. A value of 0 says do not sync this calendar or store its events on the device. A value of 1 says sync events for this calendar and store its events on the device.

## Querying a calendar

Here is an example that shows how to get the calendars that are owned by a particular user. For simplicity's sake, in this example the query operation is shown in the user interface thread ("main thread"). In practice, this should be done in an asynchronous thread instead of on the main thread. For more discussion, see [Loaders](#). If you are not just reading data but modifying it, see [AsyncQueryHandler](#).

```
// Projection array. Creating indices for this array instead of doing
// dynamic lookups improves performance.
public static final String[] EVENT_PROJECTION = new String[] {
    Calendars._ID,                                // 0
    Calendars.ACCOUNT_NAME,                         // 1
    Calendars.CALENDAR_DISPLAY_NAME,                // 2
    Calendars.OWNER_ACCOUNT                        // 3
};

// The indices for the projection array above.
private static final int PROJECTION_ID_INDEX = 0;
private static final int PROJECTION_ACCOUNT_NAME_INDEX = 1;
private static final int PROJECTION_DISPLAY_NAME_INDEX = 2;
private static final int PROJECTION_OWNER_ACCOUNT_INDEX = 3;
```

## Why must you include ACCOUNT\_TYPE?

If you query on a [Calendars.ACCOUNT\\_NAME](#), you must also include [Calendars.ACCOUNT\\_TYPE](#) in the selection. That is because a given account is only considered unique given both its [ACCOUNT\\_NAME](#) and its [ACCOUNT\\_TYPE](#). The [ACCOUNT\\_TYPE](#) is the string corresponding to the account authenticator that was used when the account was registered with the [AccountManager](#). There is also a special type of account called [ACCOUNT\\_TYPE\\_LOCAL](#) for calendars not associated with a device account. [ACCOUNT\\_TYPE\\_LOCAL](#) accounts do not get synced.

In the next part of the example, you construct your query. The selection specifies the criteria for the query. In this example the query is looking for calendars that have the [ACCOUNT\\_NAME](#) "sampleuser@google.com", the

ACCOUNT\_TYPE "com.google", and the OWNER\_ACCOUNT "sampleuser@google.com". If you want to see all calendars that a user has viewed, not just calendars the user owns, omit the OWNER\_ACCOUNT. The query returns a [Cursor](#) object that you can use to traverse the result set returned by the database query. For more discussion of using queries in content providers, see [Content Providers](#).

```
// Run query
Cursor cur = null;
ContentResolver cr = getContentResolver();
Uri uri = Calendars.CONTENT_URI;
String selection = "(((" + Calendars.ACCOUNT_NAME + " = ?) AND (" +
    + Calendars.ACCOUNT_TYPE + " = ?) AND (" +
    + Calendars.OWNER_ACCOUNT + " = ?))";
String[] selectionArgs = new String[] {"sampleuser@gmail.com", "com.google",
    "sampleuser@gmail.com"};
// Submit the query and get a Cursor object back.
cur = cr.query(uri, EVENT_PROJECTION, selection, selectionArgs, null);
```

This next section uses the cursor to step through the result set. It uses the constants that were set up at the beginning of the example to return the values for each field.

```
// Use the cursor to step through the returned records
while (cur.moveToFirst()) {
    long calID = 0;
    String displayName = null;
    String accountName = null;
    String ownerName = null;

    // Get the field values
    calID = cur.getLong(PROJECTION_ID_INDEX);
    displayName = cur.getString(PROJECTION_DISPLAY_NAME_INDEX);
    accountName = cur.getString(PROJECTION_ACCOUNT_NAME_INDEX);
    ownerName = cur.getString(PROJECTION_OWNER_ACCOUNT_INDEX);

    // Do something with the values...
    ...

}
```

## Modifying a calendar

To perform an update of an calendar, you can provide the [\\_ID](#) of the calendar either as an appended ID to the Uri ([withAppendedId\(\)](#)) or as the first selection item. The selection should start with "\_id=?", and the first selectionArg should be the [\\_ID](#) of the calendar. You can also do updates by encoding the ID in the URI. This example changes a calendar's display name using the ([withAppendedId\(\)](#)) approach:

```
private static final String DEBUG_TAG = "MyActivity";
...
long calID = 2;
ContentValues values = new ContentValues();
// The new display name for the calendar
values.put(Calendars.CALENDAR_DISPLAY_NAME, "Trevor's Calendar");
Uri updateUri = ContentUris.withAppendedId(Calendars.CONTENT_URI, calID);
int rows = getContentResolver().update(updateUri, values, null, null);
Log.i(DEBUG_TAG, "Rows updated: " + rows);
```

## Inserting a calendar

Calendars are designed to be primarily managed by a sync adapter, so you should only insert new calendars as a sync adapter. For the most part, applications can only make superficial changes to calendars, such as changing the display name. If an application needs to create a local calendar, it can do this by performing the calendar insertion as a sync adapter, using an [ACCOUNT\\_TYPE](#) of [ACCOUNT\\_TYPE\\_LOCAL](#). [ACCOUNT\\_TYPE\\_LOCAL](#) is a special account type for calendars that are not associated with a device account. Calendars of this type are not synced to a server. For a discussion of sync adapters, see [Sync Adapters](#).

## Events Table

The [CalendarContract.Events](#) table contains details for individual events. To add, update, or delete events, an application must include the [WRITE\\_CALENDAR](#) permission in its [manifest file](#).

The following Events columns are writable by both an application and a sync adapter. For a full list of supported fields, see the [CalendarContract.Events](#) reference.

Constant	Description
<a href="#">CALENDAR_ID</a>	The <a href="#">ID</a> of the calendar the event belongs to.
<a href="#">ORGANIZER</a>	Email of the organizer (owner) of the event.
<a href="#">TITLE</a>	The title of the event.
<a href="#">EVENT_LOCATION</a>	Where the event takes place.
<a href="#">DESCRIPTION</a>	The description of the event.
<a href="#">DTSTART</a>	The time the event starts in UTC milliseconds since the epoch.
<a href="#">DTEND</a>	The time the event ends in UTC milliseconds since the epoch.
<a href="#">EVENT_TIMEZONE</a>	The time zone for the event.
<a href="#">EVENT_END_TIMEZONE</a>	The time zone for the end time of the event.
<a href="#">DURATION</a>	The duration of the event in <a href="#">RFC5545</a> format. For example, a value of "PT1H" states that the event should last one hour, and a value of "P2W" indicates a duration of 2 weeks.
<a href="#">ALL_DAY</a>	A value of 1 indicates this event occupies the entire day, as defined by the local time zone. A value of 0 indicates it is a regular event that may start and end at any time during a day.
<a href="#">RRULE</a>	The recurrence rule for the event format. For example, "FREQ=WEEKLY;COUNT=10;WKST=SU". You can find more examples <a href="#">here</a> .
<a href="#">RDATE</a>	The recurrence dates for the event. You typically use <a href="#">RDATE</a> in conjunction with <a href="#">RRULE</a> to define an aggregate set of repeating occurrences. For more discussion, see the <a href="#">RFC5545 spec</a> .
<a href="#">AVAILABILITY</a>	If this event counts as busy time or is free time that can be scheduled over.
<a href="#">GUESTS_CAN_MODIFY</a>	Whether guests can modify the event.
<a href="#">GUESTS_CAN_INVITE_OTHERS</a>	Whether guests can invite other guests.
<a href="#">GUESTS_CAN_SEE_GUESTS</a>	Whether guests can see the list of attendees.

## Adding Events

When your application inserts a new event, we recommend that you use an [INSERT](#) Intent, as described in [Using an intent to insert an event](#). However, if you need to, you can insert events directly. This section describes how to do this.

Here are the rules for inserting a new event:

- You must include [CALENDAR\\_ID](#) and [DTSTART](#).
- You must include an [EVENT\\_TIMEZONE](#). To get a list of the system's installed time zone IDs, use [getAvailableIDs\(\)](#). Note that this rule does not apply if you're inserting an event through the [INSERT](#) Intent, described in [Using an intent to insert an event](#)—in that scenario, a default time zone is supplied.
- For non-recurring events, you must include [DTEND](#).
- For recurring events, you must include a [DURATION](#) in addition to [RRULE](#) or [RDATE](#). Note that this rule does not apply if you're inserting an event through the [INSERT](#) Intent, described in [Using an intent to insert an event](#)—in that scenario, you can use an [RRULE](#) in conjunction with [DTSTART](#) and [DTEND](#), and the Calendar application converts it to a duration automatically.

Here is an example of inserting an event. This is being performed in the UI thread for simplicity. In practice, inserts and updates should be done in an asynchronous thread to move the action into a background thread. For more information, see [AsyncQueryHandler](#).

```
long calID = 3;
long startMillis = 0;
long endMillis = 0;
Calendar beginTime = Calendar.getInstance();
beginTime.set(2012, 9, 14, 7, 30);
startMillis = beginTime.getTimeInMillis();
Calendar endTime = Calendar.getInstance();
endTime.set(2012, 9, 14, 8, 45);
endMillis = endTime.getTimeInMillis();
...

ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
values.put(Events.DTSTART, startMillis);
values.put(Events.DTEND, endMillis);
values.put(Events.TITLE, "Jazzercise");
values.put(Events.DESCRIPTION, "Group workout");
values.put(Events.CALENDAR_ID, calID);
values.put(Events.EVENT_TIMEZONE, "America/Los_Angeles");
Uri uri = cr.insert(Events.CONTENT_URI, values);

// get the event ID that is the last element in the Uri
long eventId = Long.parseLong(uri.getLastPathSegment());
//
// ... do something with event ID
//
//
```

**Note:** See how this example captures the event ID after the event is created. This is the easiest way to get an event ID. You often need the event ID to perform other calendar operations—for example, to add attendees or reminders to an event.

## Updating Events

When your application wants to allow the user to edit an event, we recommend that you use an [EDIT](#) Intent, as described in [Using an intent to edit an event](#). However, if you need to, you can edit events directly. To perform an update of an Event, you can provide the `_ID` of the event either as an appended ID to the Uri ([withAppendedId\(\)](#)) or as the first selection item. The selection should start with `"_id=?"`, and the first selec-

tionArg should be the `_ID` of the event. You can also do updates using a selection with no ID. Here is an example of updating an event. It changes the title of the event using the [withAppendedId\(\)](#) approach:

```
private static final String DEBUG_TAG = "MyActivity";
...
long eventID = 188;
...
ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
Uri updateUri = null;
// The new title for the event
values.put(Events.TITLE, "Kickboxing");
updateUri = ContentUris.withAppendedId(Events.CONTENT_URI, eventID);
int rows = getContentResolver().update(updateUri, values, null, null);
Log.i(DEBUG_TAG, "Rows updated: " + rows);
```

## Deleting Events

You can delete an event either by its `_ID` as an appended ID on the URI, or by using standard selection. If you use an appended ID, you can't also do a selection. There are two versions of delete: as an application and as a sync adapter. An application delete sets the `deleted` column to 1. This flag that tells the sync adapter that the row was deleted and that this deletion should be propagated to the server. A sync adapter delete removes the event from the database along with all its associated data. Here is an example of application deleting an event through its `_ID`:

```
private static final String DEBUG_TAG = "MyActivity";
...
long eventID = 201;
...
ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
Uri deleteUri = null;
deleteUri = ContentUris.withAppendedId(Events.CONTENT_URI, eventID);
int rows = getContentResolver().delete(deleteUri, null, null);
Log.i(DEBUG_TAG, "Rows deleted: " + rows);
```

## Attendees Table

Each row of the [CalendarContract.Attendees](#) table represents a single attendee or guest of an event. Calling [query\(\)](#) returns a list of attendees for the event with the given `EVENT_ID`. This `EVENT_ID` must match the `_ID` of a particular event.

The following table lists the writable fields. When inserting a new attendee, you must include all of them except `ATTENDEE_NAME`.

Constant	Description
<code>EVENT_ID</code>	The ID of the event.
<code>ATTENDEE_NAME</code>	The name of the attendee.
<code>ATTENDEE_EMAIL</code>	The email address of the attendee.
<code>ATTENDEE_RELATIONSHIP</code>	The relationship of the attendee to the event. One of:

- [RELATIONSHIP ATTENDEE](#)
- [RELATIONSHIP NONE](#)
- [RELATIONSHIP ORGANIZER](#)
- [RELATIONSHIP PERFORMER](#)
- [RELATIONSHIP SPEAKER](#)

The type of attendee. One of:

#### [ATTENDEE\\_TYPE](#)

- [TYPE REQUIRED](#)
- [TYPE OPTIONAL](#)

The attendance status of the attendee. One of:

#### [ATTENDEE\\_STATUS](#)

- [ATTENDEE\\_STATUS ACCEPTED](#)
- [ATTENDEE\\_STATUS DECLINED](#)
- [ATTENDEE\\_STATUS INVITED](#)
- [ATTENDEE\\_STATUS NONE](#)
- [ATTENDEE\\_STATUS TENTATIVE](#)

## Adding Attendees

Here is an example that adds a single attendee to an event. Note that the [EVENT\\_ID](#) is required:

```
long eventID = 202;
...
ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
values.put(Attendees.ATTENDEE_NAME, "Trevor");
values.put(Attendees.ATTENDEE_EMAIL, "trevor@example.com");
values.put(Attendees.ATTENDEE_RELATIONSHIP, Attendees.RELATIONSHIP_ATTENDEE);
values.put(Attendees.ATTENDEE_TYPE, Attendees.TYPE_OPTIONAL);
values.put(Attendees.ATTENDEE_STATUS, Attendees.ATTENDEE_STATUS_INVITED);
values.put(Attendees.EVENT_ID, eventID);
Uri uri = cr.insert(Attendees.CONTENT_URI, values);
```

## Reminders Table

Each row of the [CalendarContract.Reminders](#) table represents a single reminder for an event. Calling [query\(\)](#) returns a list of reminders for the event with the given [EVENT\\_ID](#).

The following table lists the writable fields for reminders. All of them must be included when inserting a new reminder. Note that sync adapters specify the types of reminders they support in the [CalendarContract.Calendars](#) table. See [ALLOWED REMINDERS](#) for details.

Constant	Description
<a href="#">EVENT_ID</a>	The ID of the event.
<a href="#">MINUTES</a>	The minutes prior to the event that the reminder should fire.

The alarm method, as set on the server. One of:

#### [METHOD](#)

- [METHOD ALERT](#)
- [METHOD DEFAULT](#)

- [METHOD\\_EMAIL](#)
- [METHOD\\_SMS](#)

## Adding Reminders

This example adds a reminder to an event. The reminder fires 15 minutes before the event.

```
long eventID = 221;
...
ContentResolver cr = getContentResolver();
ContentValues values = new ContentValues();
values.put(Reminders.MINUTES, 15);
values.put(Reminders.EVENT_ID, eventID);
values.put(Reminders.METHOD, Reminders.METHOD_ALERT);
Uri uri = cr.insert(Reminders.CONTENT_URI, values);
```

## Instances Table

The [CalendarContract.Instances](#) table holds the start and end time for occurrences of an event. Each row in this table represents a single event occurrence. The instances table is not writable and only provides a way to query event occurrences.

The following table lists some of the fields you can query on for an instance. Note that time zone is defined by [KEY\\_TIMEZONE\\_TYPE](#) and [KEY\\_TIMEZONE\\_INSTANCES](#).

Constant	Description
<a href="#">BEGIN</a>	The beginning time of the instance, in UTC milliseconds.
<a href="#">END</a>	The ending time of the instance, in UTC milliseconds.
<a href="#">END_DAY</a>	The Julian end day of the instance, relative to the Calendar's time zone.
<a href="#">END_MINUTE</a>	The end minute of the instance measured from midnight in the Calendar's time zone.
<a href="#">EVENT_ID</a>	The _ID of the event for this instance.
<a href="#">START_DAY</a>	The Julian start day of the instance, relative to the Calendar's time zone.
<a href="#">START_MINUTE</a>	The start minute of the instance measured from midnight, relative to the Calendar's time zone.

## Querying the Instances table

To query the Instances table, you need to specify a range time for the query in the URI. In this example, [CalendarContract.Instances](#) gets access to the [TITLE](#) field through its implementation of the [CalendarContract.EventsColumns](#) interface. In other words, [TITLE](#) is returned through a database view, not through querying the raw [CalendarContract.Instances](#) table.

```
private static final String DEBUG_TAG = "MyActivity";
public static final String[] INSTANCE_PROJECTION = new String[] {
    Instances.EVENT_ID,           // 0
    Instances.BEGIN,              // 1
    Instances.TITLE               // 2
};

// The indices for the projection array above.
private static final int PROJECTION_ID_INDEX = 0;
private static final int PROJECTION_BEGIN_INDEX = 1;
```

```
private static final int PROJECTION_TITLE_INDEX = 2;
...

// Specify the date range you want to search for recurring
// event instances
Calendar beginTime = Calendar.getInstance();
beginTime.set(2011, 9, 23, 8, 0);
long startMillis = beginTime.getTimeInMillis();
Calendar endTime = Calendar.getInstance();
endTime.set(2011, 10, 24, 8, 0);
long endMillis = endTime.getTimeInMillis();

Cursor cur = null;
ContentResolver cr = getContentResolver();

// The ID of the recurring event whose instances you are searching
// for in the Instances table
String selection = Instances.EVENT_ID + " = ?";
String[] selectionArgs = new String[] {"207"};

// Construct the query with the desired date range.
Uri.Builder builder = Instances.CONTENT_URI.buildUpon();
ContentUris.appendId(builder, startMillis);
ContentUris.appendId(builder, endMillis);

// Submit the query
cur = cr.query(builder.build(),
    INSTANCE_PROJECTION,
    selection,
    selectionArgs,
    null);

while (cur.moveToFirst()) {
    String title = null;
    long eventID = 0;
    long beginVal = 0;

    // Get the field values
    eventID = cur.getLong(PROJECTION_ID_INDEX);
    beginVal = cur.getLong(PROJECTION_BEGIN_INDEX);
    title = cur.getString(PROJECTION_TITLE_INDEX);

    // Do something with the values.
    Log.i(DEBUG_TAG, "Event: " + title);
    Calendar calendar = Calendar.getInstance();
    calendar.setTimeInMillis(beginVal);
    DateFormat formatter = new SimpleDateFormat("MM/dd/yyyy");
    Log.i(DEBUG_TAG, "Date: " + formatter.format(calendar.getTime()));
}
```

# Calendar Intents

Your application doesn't need [permissions](#) to read and write calendar data. It can instead use intents supported by Android's Calendar application to hand off read and write operations to that application. The following table lists the intents supported by the Calendar Provider:

Action	URI	Description	Extras
<a href="#">VIEW</a>	content://com.android.calendar/time/<ms_since_epoch>	Open calendar to the time specified by <ms_since_epoch>. You can also refer to the URI with <a href="#">CalendarContract.CONTENT_URI</a> . For an example of using this intent, see <a href="#">Using intents to view calendar data</a> .	None.
<a href="#">VIEW</a>	content://com.android.calendar/events/<event_id>	View the event specified by <event_id>. You can also refer to the URI with <a href="#">Events.CONTENT_URI</a> . For an example of using this intent, see <a href="#">Using intents to view calendar data</a> .	<a href="#">CalendarContract.EXTRA_EVENT</a>
<a href="#">EDIT</a>	content://com.android.calendar/events/<event_id>	Edit the event specified by <event_id>. You can also refer to the URI with <a href="#">Events.CONTENT_URI</a> . For an example of using this intent, see <a href="#">Using an intent to edit an event</a> .	<a href="#">CalendarContract.EXTRA_EVENT</a>
<a href="#">EDIT</a>	content://com.android.calendar/events	Create an event. You can also refer to the URI with <a href="#">Events.CONTENT_URI</a> . For an example of using this intent, see <a href="#">Using an intent to insert an event</a> .	Any of the extras listed in the table below.

The following table lists the intent extras supported by the Calendar Provider:

Intent Extra	Description
<a href="#">Events.TITLE</a>	Name for the event.
<a href="#">CalendarContract.EXTRA_EVENT_BEGIN_TIME</a>	Event begin time in milliseconds from the epoch.
<a href="#">CalendarContract.EXTRA_EVENT_END_TIME</a>	Event end time in milliseconds from the epoch.
<a href="#">CalendarContract.EXTRA_EVENT_ALL_DAY</a>	A boolean that indicates that an event is all day. Value can be <code>true</code> or <code>false</code> .
<a href="#">Events.EVENT_LOCATION</a>	Location of the event.
<a href="#">Events.DESCRIPTION</a>	Event description.
<a href="#">Intent.EXTRA_EMAIL</a>	Email addresses of those to invite as a comma-separated list.

[Events.RRULE](#)

[Events.ACCESS\\_LEVEL](#)

[Events.AVAILABILITY](#)

The recurrence rule for the event.

Whether the event is private or public.

If this event counts as busy time or is free time that can be scheduled over.

The following sections describe how to use these intents.

## Using an intent to insert an event

Using the [INSERT](#) Intent lets your application hand off the event insertion task to the Calendar itself. With this approach, your application doesn't even need to have the [WRITE\\_CALENDAR](#) permission included in its [manifest file](#).

When users run an application that uses this approach, the application sends them to the Calendar to finish adding the event. The [INSERT](#) Intent uses extra fields to pre-populate a form with the details of the event in the Calendar. Users can then cancel the event, edit the form as needed, or save the event to their calendars.

Here is a code snippet that schedules an event on January 19, 2012, that runs from 7:30 a.m. to 8:30 a.m. Note the following about this code snippet:

- It specifies [Events.CONTENT\\_URI](#) as the Uri.
- It uses the [CalendarContract.EXTRA\\_EVENT\\_BEGIN\\_TIME](#) and [CalendarContract.EXTRA\\_EVENT\\_END\\_TIME](#) extra fields to pre-populate the form with the time of the event. The values for these times must be in UTC milliseconds from the epoch.
- It uses the [Intent.EXTRA\\_EMAIL](#) extra field to provide a comma-separated list of invitees, specified by email address.

```
Calendar beginTime = Calendar.getInstance();
beginTime.set(2012, 0, 19, 7, 30);
Calendar endTime = Calendar.getInstance();
endTime.set(2012, 0, 19, 8, 30);
Intent intent = new Intent(Intent.ACTION_INSERT)
    .setData(Events.CONTENT_URI)
    .putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME, beginTime.getTimeInMillis())
    .putExtra(CalendarContract.EXTRA_EVENT_END_TIME, endTime.getTimeInMillis())
    .putExtra(Events.TITLE, "Yoga")
    .putExtra(Events.DESCRIPTION, "Group class")
    .putExtra(Events.EVENT_LOCATION, "The gym")
    .putExtra(Events.AVAILABILITY, Events.AVAILABILITY_BUSY)
    .putExtra(Intent.EXTRA_EMAIL, "rowan@example.com,trevor@example.com");
startActivity(intent);
```

## Using an intent to edit an event

You can update an event directly, as described in [Updating events](#). But using the [EDIT](#) Intent allows an application that doesn't have permission to hand off event editing to the Calendar application. When users finish editing their event in Calendar, they're returned to the original application.

Here is an example of an intent that sets a new title for a specified event and lets users edit the event in the Calendar.

```
long eventID = 208;
Uri uri = ContentUris.withAppendedId(Events.CONTENT_URI, eventID);
```

```
Intent intent = new Intent(Intent.ACTION_EDIT)
    .setData(uri)
    .putExtra(Events.TITLE, "My New Title");
startActivity(intent);
```

## Using intents to view calendar data

Calender Provider offers two different ways to use the [VIEW](#) Intent:

- To open the Calendar to a particular date.
- To view an event.

Here is an example that shows how to open the Calendar to a particular date:

```
// A date-time specified in milliseconds since the epoch.
long startMillis;
...
Uri.Builder builder = CalendarContract.CONTENT_URI.buildUpon();
builder.appendPath("time");
ContentUris.appendId(builder, startMillis);
Intent intent = new Intent(Intent.ACTION_VIEW)
    .setData(builder.build());
startActivity(intent);
```

Here is an example that shows how to open an event for viewing:

```
long eventID = 208;
...
Uri uri = ContentUris.withAppendedId(Events.CONTENT_URI, eventID);
Intent intent = new Intent(Intent.ACTION_VIEW)
    .setData(uri);
startActivity(intent);
```

## Sync Adapters

There are only minor differences in how an application and a sync adapter access the Calendar Provider:

- A sync adapter needs to specify that it's a sync adapter by setting [CALLER\\_IS\\_SYNCADAPTER](#) to true.
- A sync adapter needs to provide an [ACCOUNT\\_NAME](#) and an [ACCOUNT\\_TYPE](#) as query parameters in the URI.
- A sync adapter has write access to more columns than an application or widget. For example, an application can only modify a few characteristics of a calendar, such as its name, display name, visibility setting, and whether the calendar is synced. By comparison, a sync adapter can access not only those columns, but many others, such as calendar color, time zone, access level, location, and so on. However, a sync adapter is restricted to the [ACCOUNT\\_NAME](#) and [ACCOUNT\\_TYPE](#) it specified.

Here is a helper method you can use to return a URI for use with a sync adapter:

```
static Uri asSyncAdapter(Uri uri, String account, String accountType) {
    return uri.buildUpon()
        .appendQueryParameter(android.provider.CalendarContract.CALLER_IS_SYNCADAPTER,
            "1")
        .appendQueryParameter(CalendarContract Calendars.ACCOUNT_NAME, account)
```

```
.appendQueryParameter(CalendarSyncAdapter.ACCOUNT_TYPE, accountType).build();  
}
```

For a sample implementation of a sync adapter (not specifically related to Calendar), see [SampleSyncAdapter](#).

# Contacts Provider

## Quickview

- Android's repository of information about people.
- Syncs with the web.
- Integrates social stream data.

## In this document

1. [Contacts Provider Organization](#)
2. [Raw contacts](#)
3. [Data](#)
4. [Contacts](#)
5. [Data From Sync Adapters](#)
6. [Required Permissions](#)
7. [The User Profile](#)
8. [Contacts Provider Metadata](#)
9. [Contacts Provider Access](#)
- 10.
11. [Contacts Provider Sync Adapters](#)
12. [Social Stream Data](#)
13. [Additional Contacts Provider Features](#)

## Key classes

1. [ContactsContract.Contacts](#)
2. [ContactsContract.RawContacts](#)
3. [ContactsContract.Data](#)
4. [ContactsContract.StreamItems](#)

## Related Samples

1. [Contact Manager](#)
2. [Sample Sync Adapter](#)

## See Also

1. [Content Provider Basics](#)

The Contacts Provider is a powerful and flexible Android component that manages the device's central repository of data about people. The Contacts Provider is the source of data you see in the device's contacts application, and you can also access its data in your own application and transfer data between the device and online services. The provider accommodates a wide range of data sources and tries to manage as much data as possible for each person, with the result that its organization is complex. Because of this, the provider's API includes an extensive set of contract classes and interfaces that facilitate both data retrieval and modification.

This guide describes the following:

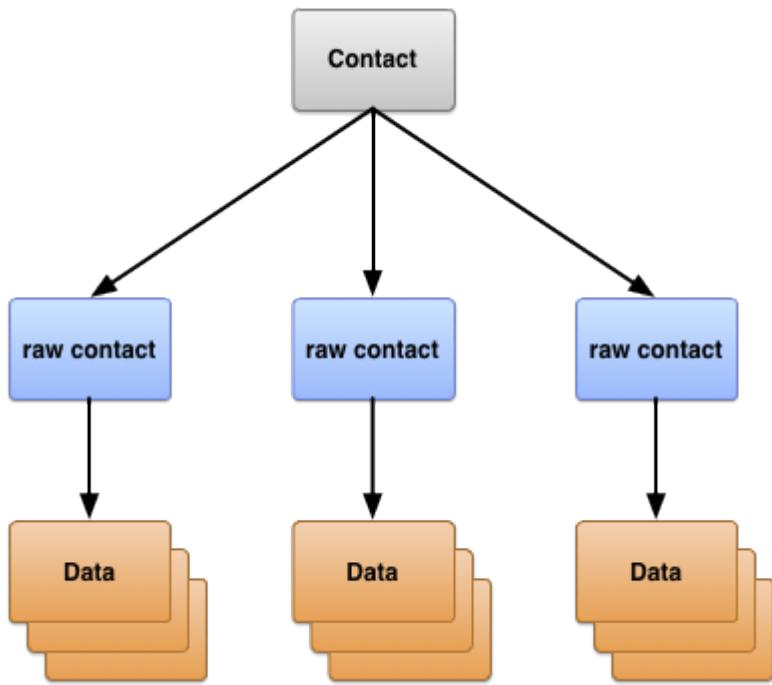
- The basic provider structure.

- How to retrieve data from the provider.
- How to modify data in the provider.
- How to write a sync adapter for synchronizing data from your server to the Contacts Provider.

This guide assumes that you know the basics of Android content providers. To learn more about Android content providers, read the [Content Provider Basics](#) guide. The [Sample Sync Adapter](#) sample app is an example of using a sync adapter to transfer data between the Contacts Provider and a sample application hosted by Google Web Services.

## Contacts Provider Organization

The Contacts Provider is an Android content provider component. It maintains three types of data about a person, each of which corresponds to a table offered by the provider, as illustrated in figure 1:



**Figure 1.** Contacts Provider table structure.

The three tables are commonly referred to by the names of their contract classes. The classes define constants for content URIs, column names, and column values used by the tables:

### [`ContactsContract.Contacts`](#) table

Rows representing different people, based on aggregations of raw contact rows.

### [`ContactsContract.RawContacts`](#) table

Rows containing a summary of a person's data, specific to a user account and type.

### [`ContactsContract.Data`](#) table

Rows containing the details for raw contact, such as email addresses or phone numbers.

The other tables represented by contract classes in [`ContactsContract`](#) are auxiliary tables that the Contacts Provider uses to manage its operations or support specific functions in the device's contacts or telephony applications.

# Raw contacts

A raw contact represents a person's data coming from a single account type and account name. Because the Contacts Provider allows more than one online service as the source of data for a person, the Contacts Provider allows multiple raw contacts for the same person. Multiple raw contacts also allow a user to combine a person's data from more than one account from the same account type.

Most of the data for a raw contact isn't stored in the [ContactsContract.RawContacts](#) table. Instead, it's stored in one or more rows in the [ContactsContract.Data](#) table. Each data row has a column [Data.RAW\\_CONTACT\\_ID](#) that contains the [RawContacts.ID](#) value of its parent [ContactsContract.RawContacts](#) row.

## Important raw contact columns

The important columns in the [ContactsContract.RawContacts](#) table are listed in table 1. Please read the notes that follow after the table:

**Table 1.** Important raw contact columns.

Column name	Use	Notes
<a href="#">ACCOUNT_NAME</a>	The account name for the account type that's the source of this raw contact. For example, the account name of a Google account is one of the device owner's Gmail addresses. See the next entry for <a href="#">ACCOUNT_TYPE</a> for more information.	The format of this name is specific to its account type. It is not necessarily an email address.
<a href="#">ACCOUNT_TYPE</a>	The account type that's the source of this raw contact. For example, the account type of a Google account is com.google. Always qualify your account type with a domain identifier for a domain you own or control. This will ensure that your account type is unique.	An account type that offers contacts data usually has an associated sync adapter that synchronizes with the Contacts Provider.
<a href="#">DELETED</a>	The "deleted" flag for a raw contact.	This flag allows the Contacts Provider to maintain the row internally until sync adapters are able to delete the row from their servers and then finally delete the row from the repository.

## Notes

The following are important notes about the [ContactsContract.RawContacts](#) table:

- A raw contact's name is not stored in its row in [ContactsContract.RawContacts](#). Instead, it's stored in the [ContactsContract.Data](#) table, in a [ContactsContract.CommonDataKinds.StructuredName](#) row. A raw contact has only one row of this type in the [ContactsContract.Data](#) table.
- **Caution:** To use your own account data in a raw contact row, it must first be registered with the [AccountManager](#). To do this, prompt users to add the account type and their account name to the list of accounts. If you don't do this, the Contacts Provider will automatically delete your raw contact row.

For example, if you want your app to maintain contacts data for your web-based service with the domain com.example.dataservice, and the user's account for your service is becky.sharp@dataservice.example.com, the user must first add the account "type"

(com.example.dataservice) and account "name" (becky.smart@dataservice.example.com) before your app can add raw contact rows. You can explain this requirement to the user in documentation, or you can prompt the user to add the type and name, or both. Account types and account names are described in more detail in the next section.

## Sources of raw contacts data

To understand how raw contacts work, consider the user "Emily Dickinson" who has the following three user accounts defined on her device:

- emily.dickinson@gmail.com
- emilyd@gmail.com
- Twitter account "belle\_of\_amherst"

This user has enabled *Sync Contacts* for all three of these accounts in the *Accounts* settings.

Suppose Emily Dickinson opens a browser window, logs into Gmail as emily.dickinson@gmail.com, opens Contacts, and adds "Thomas Higginson". Later on, she logs into Gmail as emilyd@gmail.com and sends an email to "Thomas Higginson", which automatically adds him as a contact. She also follows "colonel\_tom" (Thomas Higginson's Twitter ID) on Twitter.

The Contacts Provider creates three raw contacts as a result of this work:

1. A raw contact for "Thomas Higginson" associated with emily.dickinson@gmail.com. The user account type is Google.
2. A second raw contact for "Thomas Higginson" associated with emilyd@gmail.com. The user account type is also Google. There is a second raw contact even though the name is identical to a previous name, because the person was added for a different user account.
3. A third raw contact for "Thomas Higginson" associated with "belle\_of\_amherst". The user account type is Twitter.

## Data

As noted previously, the data for a raw contact is stored in a [ContactsContract.Data](#) row that is linked to the raw contact's \_ID value. This allows a single raw contact to have multiple instances of the same type of data such as email addresses or phone numbers. For example, if "Thomas Higginson" for emilyd@gmail.com (the raw contact row for Thomas Higginson associated with the Google account emilyd@gmail.com) has a home email address of thigg@gmail.com and a work email address of thomas.higginson@gmail.com, the Contacts Provider stores the two email address rows and links them both to the raw contact.

Notice that different types of data are stored in this single table. Display name, phone number, email, postal address, photo, and website detail rows are all found in the [ContactsContract.Data](#) table. To help manage this, the [ContactsContract.Data](#) table has some columns with descriptive names, and others with generic names. The contents of a descriptive-name column have the same meaning regardless of the type of data in the row, while the contents of a generic-name column have different meanings depending on the type of data.

## Descriptive column names

Some examples of descriptive column names are:

### [RAW CONTACT ID](#)

The value of the \_ID column of the raw contact for this data.

## MIMETYPE

The type of data stored in this row, expressed as a custom MIME type. The Contacts Provider uses the MIME types defined in the subclasses of [ContactsContract.CommonDataKinds](#). These MIME types are open source, and can be used by any application or sync adapter that works with the Contacts Provider.

## IS\_PRIMARY

If this type of data row can occur more than once for a raw contact, the [IS\\_PRIMARY](#) column flags the data row that contains the primary data for the type. For example, if the user long-presses a phone number for a contact and selects **Set default**, then the [ContactsContract.Data](#) row containing that number has its [IS\\_PRIMARY](#) column set to a non-zero value.

## Generic column names

There are 15 generic columns named DATA1 through DATA15 that are generally available and an additional four generic columns SYNC1 through SYNC4 that should only be used by sync adapters. The generic column name constants always work, regardless of the type of data the row contains.

The DATA1 column is indexed. The Contacts Provider always uses this column for the data that the provider expects will be the most frequent target of a query. For example, in an email row, this column contains the actual email address.

By convention, the column DATA15 is reserved for storing Binary Large Object (BLOB) data such as photo thumbnails.

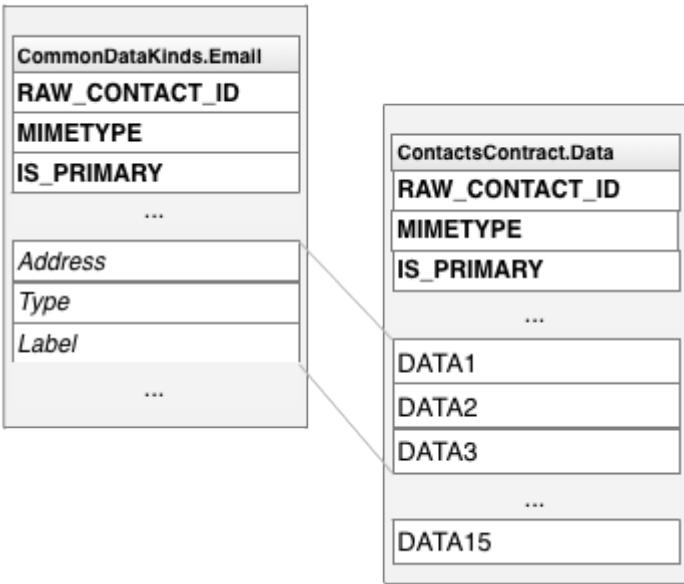
## Type-specific column names

To facilitate working with the columns for a particular type of row, the Contacts Provider also provides type-specific column name constants, defined in subclasses of [ContactsContract.CommonDataKinds](#). The constants simply give a different constant name to the same column name, which helps you access data in a row of a particular type.

For example, the [ContactsContract.CommonDataKinds.Email](#) class defines type-specific column name constants for a [ContactsContract.Data](#) row that has the MIME type [Email.CONTENT\\_ITEM\\_TYPE](#). The class contains the constant [ADDRESS](#) for the email address column. The actual value of [ADDRESS](#) is "data1", which is the same as the column's generic name.

**Caution:** Don't add your own custom data to the [ContactsContract.Data](#) table using a row that has one of the provider's pre-defined MIME types. If you do, you may lose the data or cause the provider to malfunction. For example, you should not add a row with the MIME type [Email.CONTENT\\_ITEM\\_TYPE](#) that contains a user name instead of an email address in the column DATA1. If you use your own custom MIME type for the row, then you are free to define your own type-specific column names and use the columns however you wish.

Figure 2 shows how descriptive columns and data columns appear in a [ContactsContract.Data](#) row, and how type-specific column names "overlay" the generic column names



**Figure 2.** Type-specific column names and generic column names.

## Type-specific column name classes

Table 2 lists the most commonly-used type-specific column name classes:

**Table 2.** Type-specific column name classes

Mapping class	Type of data	Notes
<a href="#">ContactsContract.CommonDataKinds.StructuredName</a>	The name data for the raw contact associated with this data row.	A raw contact has only one of these rows.
<a href="#">ContactsContract.CommonDataKinds.Photo</a>	The main photo for the raw contact associated with this data row.	A raw contact has only one of these rows.
<a href="#">ContactsContract.CommonDataKinds.Email</a>	An email address for the raw contact associated with this data row.	A raw contact can have multiple email addresses.
<a href="#">ContactsContract.CommonDataKinds.StructuredPostal</a>	An postal address for the raw contact associated with this data row.	A raw contact can have multiple postal addresses.
<a href="#">ContactsContract.CommonDataKinds.GroupMembership</a>	An identifier that links the raw contact to one of the groups it belongs to.	Groups are an optional feature of an account. They're used to categorize contacts based on their group membership.

groups in the Contacts Provider. described in more detail in the section [Contact groups](#).

## Contacts

The Contacts Provider combines the raw contact rows across all account types and account names to form a **contact**. This facilitates displaying and modifying all the data a user has collected for a person. The Contacts Provider manages the creation of new contact rows, and the aggregation of raw contacts with an existing contact row. Neither applications nor sync adapters are allowed to add contacts, and some columns in a contact row are read-only.

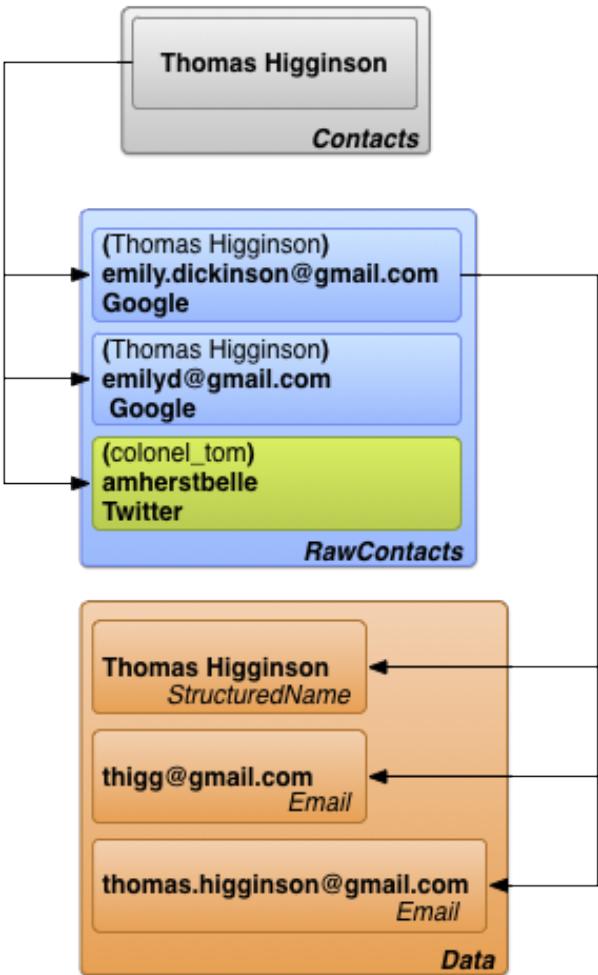
**Note:** If you try to add a contact to the Contacts Provider with an [insert\(\)](#), you'll get an [UnsupportedOperationException](#) exception. If you try to update a column that's listed as "read-only," the update is ignored.

The Contacts Provider creates a new contact in response to the addition of a new raw contact that doesn't match any existing contacts. The provider also does this if an existing raw contact's data changes in such a way that it no longer matches the contact to which it was previously attached. If an application or sync adapter creates a new raw contact that *does* match an existing contact, the new raw contact is aggregated to the existing contact.

The Contacts Provider links a contact row to its raw contact rows with the contact row's `_ID` column in the [Contacts](#) table. The `CONTACT_ID` column of the raw contacts table [ContactsContract.RawContacts](#) contains `_ID` values for the contacts row associated with each raw contacts row.

The [ContactsContract.Contacts](#) table also has the column `LOOKUP_KEY` that is a "permanent" link to the contact row. Because the Contacts Provider maintains contacts automatically, it may change a contact row's `_ID` value in response to an aggregation or sync. Even if this happens, the content URI [CONTENT\\_LOOKUP\\_URI](#) combined with contact's `LOOKUP_KEY` will still point to the contact row, so you can use `LOOKUP_KEY` to maintain links to "favorite" contacts, and so forth. This column has its own format that is unrelated to the format of the `_ID` column.

Figure 3 shows how the three main tables relate to each other.



**Figure 3.** Contacts, Raw Contacts, and Details table relationships.

## Data From Sync Adapters

Users enter contacts data directly into the device, but data also flows into the Contacts Provider from web services via **sync adapters**, which automate the transfer of data between the device and services. Sync adapters run in the background under the control of the system, and they call [ContentResolver](#) methods to manage data.

In Android, the web service that a sync adapter works with is identified by an account type. Each sync adapter works with one account type, but it can support multiple account names for that type. Account types and account names are described briefly in the section [Sources of raw contacts data](#). The following definitions offer more detail, and describe how account type and name relate to sync adapters and services.

### Account type

Identifies a service in which the user has stored data. Most of the time, the user has to authenticate with the service. For example, Google Contacts is an account type, identified by the code `google.com`. This value corresponds to the account type used by [AccountManager](#).

### Account name

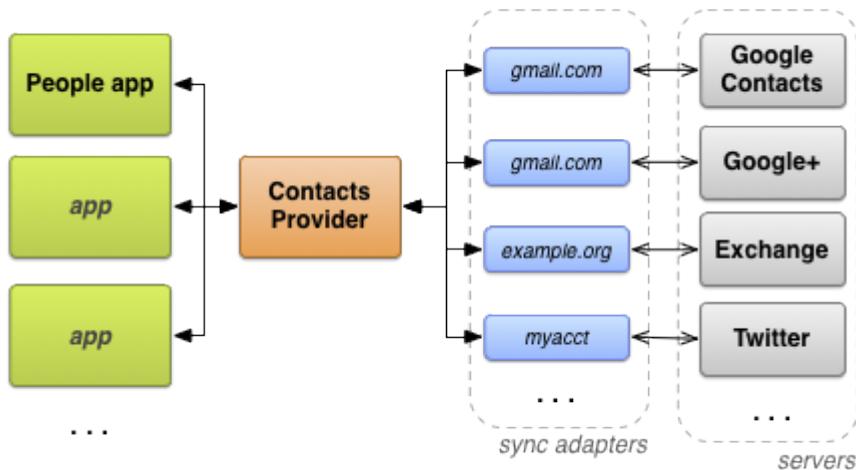
Identifies a particular account or login for an account type. Google Contacts accounts are the same as Google accounts, which have an email address as an account name. Other services may use a single-word username or numeric id.

Account types don't have to be unique. A user can configure multiple Google Contacts accounts and download their data to the Contacts Provider; this may happen if the user has one set of personal contacts for a personal

account name, and another set for work. Account names are usually unique. Together, they identify a specific data flow between the Contacts Provider and an external service.

If you want to transfer your service's data to the Contacts Provider, you need to write your own sync adapter. This is described in more detail in the section [Contacts Provider Sync Adapters](#).

Figure 4 shows how the Contacts Provider fits into the flow of data about people. In the box marked "sync adapters," each adapter is labeled by its account type.



**Figure 4.** The Contacts Provider flow of data.

## Required Permissions

Applications that want to access the Contacts Provider must request the following permissions:

### Read access to one or more tables

[READ\\_CONTACTS](#), specified in `AndroidManifest.xml` with the [`<uses-permission>`](#) element as `<uses-permission android:name="android.permission.READ_CONTACTS">`.

### Write access to one or more tables

[WRITE\\_CONTACTS](#), specified in `AndroidManifest.xml` with the [`<uses-permission>`](#) element as `<uses-permission android:name="android.permission.WRITE_CONTACTS">`.

These permissions do not extend to the user profile data. The user profile and its required permissions are discussed in the following section, [The User Profile](#).

Remember that the user's contacts data is personal and sensitive. Users are concerned about their privacy, so they don't want applications collecting data about them or their contacts. If it's not obvious why you need permission to access their contacts data, they may give your application low ratings or simply refuse to install it.

## The User Profile

The [ContactsContract.Contacts](#) table has a single row containing profile data for the device's user. This data describes the device's user rather than one of the user's contacts. The profile contacts row is linked to a raw contacts row for each system that uses a profile. Each profile raw contact row can have multiple data rows. Constants for accessing the user profile are available in the [ContactsContract.Profile](#) class.

Access to the user profile requires special permissions. In addition to the [READ\\_CONTACTS](#) and [WRITE\\_CONTACTS](#) permissions needed to read and write, access to the user profile requires the [READ\\_PROFILE](#) and [WRITE\\_PROFILE](#) permissions for read and write access, respectively.

Remember that you should consider a user's profile to be sensitive. The permission [READ\\_PROFILE](#) allows you to access the device user's personally-identifying data. Make sure to tell the user why you need user profile access permissions in the description of your application.

To retrieve the contact row that contains the user's profile, call [ContentResolver.query\(\)](#). Set the content URI to [CONTENT\\_URI](#) and don't provide any selection criteria. You can also use this content URI as the base URI for retrieving raw contacts or data for the profile. For example, this snippet retrieves data for the profile:

```
// Sets the columns to retrieve for the user profile
mProjection = new String[]
{
    Profile._ID,
    Profile.DISPLAY_NAME_PRIMARY,
    Profile.LOOKUP_KEY,
    Profile.PHOTO_THUMBNAIL_URI
};

// Retrieves the profile from the Contacts Provider
mProfileCursor =
    getContentResolver().query(
        Profile.CONTENT_URI,
        mProjection ,
        null,
        null,
        null);
```

**Note:** If you retrieve multiple contact rows, and you want to determine if one of them is the user profile, test the row's [IS\\_USER\\_PROFILE](#) column. This column is set to "1" if the contact is the user profile.

## Contacts Provider Metadata

The Contacts Provider manages data that keeps track of the state of contacts data in the repository. This metadata about the repository is stored in various places, including the Raw Contacts, Data, and Contacts table rows, the [ContactsContract.Settings](#) table, and the [ContactsContract.SyncState](#) table. The following table shows the effect of each of these pieces of metadata:

**Table 3.** Metadata in the Contacts Provider

Table	Column	Values	Meaning
<a href="#">ContactsContract.RawContacts</a>	<a href="#">DIRTY</a>	"0" - not changed since the last sync. "1" - changed since last sync, needs to	Marks raw contacts that were changed synced back to the server. The value is contacts Provider when Android application
<a href="#">ContactsContract</a>	<a href="#">SyncState</a>		Sync adapters that modify the raw contacts append the string <a href="#">CALLER_IS_SYNCING</a> they use. This prevents the provider from wise, sync adapter modifications appear

<a href="#">ContactsContract.RawContacts.VERSION</a>		be synced back to the server. are sent to the server, even though the modification.
<a href="#">ContactsContract.Data</a>	<a href="#">DATA_VERSION</a>	The version number of this row or its related data changes.
<a href="#">ContactsContract.RawContacts.SOURCE_ID</a>		The version number of this data row is changed.
<a href="#">ContactsContract.Groups</a>	<a href="#">GROUP_VISIBLE</a>	A string value that uniquely identifies this raw contact to the account in which it was created.
<a href="#">ContactsContract.Settings</a>	<a href="#">UNGROUPED_VISIBLE</a>	<p>"0" - Contacts in this group should not be visible in Android application UIs.</p> <p>"1" - Contacts in this group are allowed to be visible in application UIs.</p> <p>"0" - For this account and</p> <p>By default, contacts are invisible if not to a group (Group membership for a more</p>

<code>ContactsContract.SyncState</code>	(all)	account type, contacts that don't belong to a group are invisible to Android application UIs. "1" - For this account and account type, contacts that don't belong to a group are visible to application UIs.	<code>tract.CommonDataKinds.GroupContactsContract.Data</code> table). <code>ContactsContract.Settings</code> table count, you can force contacts without this flag is to show contacts from serv
---	-------	---	---

## Contacts Provider Access

This section describes guidelines for accessing data from the Contacts Provider, focusing on the following:

- Entity queries.
- Batch modification.
- Retrieval and modification with intents.
- Data integrity.

Making modifications from a sync adapter is also covered in more detail in the section [Contacts Provider Sync Adapters](#).

### Querying entities

Because the Contacts Provider tables are organized in a hierarchy, it's often useful to retrieve a row and all of the "child" rows that are linked to it. For example, to display all the information for a person, you may want to retrieve all the `ContactsContract.RawContacts` rows for a single `ContactsContract.Contacts` row, or all the `ContactsContract.CommonDataKinds.Email` rows for a single `ContactsContract.RawContacts` row. To facilitate this, the Contacts Provider offers **entity** constructs, which act like database joins between tables.

An entity is like a table composed of selected columns from a parent table and its child table. When you query an entity, you supply a projection and search criteria based on the columns available from the entity. The result

is a [Cursor](#) that contains one row for each child table row that was retrieved. For example, if you query [ContactsContract.Contacts.Entity](#) for a contact name and all the [ContactsContract.CommonDataKinds.Email](#) rows for all the raw contacts for that name, you get back a [Cursor](#) containing one row for each [ContactsContract.CommonDataKinds.Email](#) row.

Entities simplify queries. Using an entity, you can retrieve all of the contacts data for a contact or raw contact at once, instead of having to query the parent table first to get an ID, and then having to query the child table with that ID. Also, the Contacts Provider processes a query against an entity in a single transaction, which ensures that the retrieved data is internally consistent.

**Note:** An entity usually doesn't contain all the columns of the parent and child table. If you attempt to work with a column name that isn't in the list of column name constants for the entity, you'll get an [Exception](#).

The following snippet shows how to retrieve all the raw contact rows for a contact. The snippet is part of a larger application that has two activities, "main" and "detail". The main activity shows a list of contact rows; when the user select one, the activity sends its ID to the detail activity. The detail activity uses the [ContactsContract.Contacts.Entity](#) to display all of the data rows from all of the raw contacts associated with the selected contact.

This snippet is taken from the "detail" activity:

```
...
/*
 * Appends the entity path to the URI. In the case of the Contacts Provider
 * expected URI is content://com.google.contacts/#/entity (# is the ID value)
 */
mContactUri = Uri.withAppendedPath(
    mContactUri,
    ContactsContract.Contacts.Entity.CONTENT_DIRECTORY);

// Initializes the loader identified by LOADER_ID.
getLoaderManager().initLoader(
    LOADER_ID, // The identifier of the loader to initialize
    null,      // Arguments for the loader (in this case, none)
    this);     // The context of the activity

// Creates a new cursor adapter to attach to the list view
mCursorAdapter = new SimpleCursorAdapter(
    this,                      // the context of the activity
    R.layout.detail_list_item, // the view item containing the detail
    mCursor,                  // the backing cursor
    mFromColumns,             // the columns in the cursor that provide
    mToViews,                 // the views in the view item that display
    0);                      // flags

// Sets the ListView's backing adapter.
mRawContactList.setAdapter(mCursorAdapter);
...
@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {

    /*
     * Sets the columns to retrieve.
     * RAW_CONTACT_ID is included to identify the raw contact associated with t
```

```

 * DATA1 contains the first column in the data row (usually the most import
 * MIMETYPE indicates the type of data in the data row.
 */
String[] projection =
{
    ContactsContract.Contacts.Entity.RAW_CONTACT_ID,
    ContactsContract.Contacts.Entity.DATA1,
    ContactsContract.Contacts.Entity.MIMETYPE
};

/*
 * Sorts the retrieved cursor by raw contact id, to keep all data rows for
 * contact collated together.
*/
String sortOrder =
    ContactsContract.Contacts.Entity.RAW_CONTACT_ID +
    " ASC";

/*
 * Returns a new CursorLoader. The arguments are similar to
 * ContentResolver.query(), except for the Context argument, which supplies
 * the ContentResolver to use.
*/
return new CursorLoader(
    getApplicationContext(), // The activity's context
    mContactUri,           // The entity content URI for a single co
    projection,            // The columns to retrieve
    null,                  // Retrieve all the raw contacts and thei
    null,                  //
    sortOrder);           // Sort by the raw contact ID.
}

```

When the load is finished, [LoaderManager](#) invokes a callback to [onLoadFinished\(\)](#). One of the incoming arguments to this method is a [Cursor](#) with the results of the query. In your own app, you can get the data from this [Cursor](#) to display it or work with it further.

## Batch modification

Whenever possible, you should insert, update, and delete data in the Contacts Provider in "batch mode", by creating an [ArrayList](#) of [ContentProviderOperation](#) objects and calling [applyBatch\(\)](#). Because the Contacts Provider performs all of the operations in an [applyBatch\(\)](#) in a single transaction, your modifications will never leave the contacts repository in an inconsistent state. A batch modification also facilitates inserting a raw contact and its detail data at the same time.

**Note:** To modify a *single* raw contact, consider sending an intent to the device's contacts application rather than handling the modification in your app. Doing this is described in more detail in the section [Retrieval and modification with intents](#).

## Yield points

A batch modification containing a large number of operations can block other processes, resulting in a bad overall user experience. To organize all the modifications you want to perform in as few separate lists as possible, and at the same time prevent them from blocking the system, you should set **yield points** for one or more

operations. A yield point is a [ContentProviderOperation](#) object that has its [isYieldAllowed\(\)](#) value set to `true`. When the Contacts Provider encounters a yield point, it pauses its work to let other processes run and closes the current transaction. When the provider starts again, it continues with the next operation in the [ArrayList](#) and starts a new transaction.

Yield points do result in more than one transaction per call to [applyBatch\(\)](#). Because of this, you should set a yield point for the last operation for a set of related rows. For example, you should set a yield point for the last operation in a set that adds a raw contact rows and its associated data rows, or the last operation for a set of rows related to a single contact.

Yield points are also a unit of atomic operation. All accesses between two yield points will either succeed or fail as a single unit. If you don't set any yield points, the smallest atomic operation is the entire batch of operations. If you do use yield points, you prevent operations from degrading system performance, while at the same time ensuring that a subset of operations is atomic.

## Modification back references

When you're inserting a new raw contact row and its associated data rows as a set of [ContentProviderOperation](#) objects, you have to link the data rows to the raw contact row by inserting the raw contact's [\\_ID](#) value as the [RAW\\_CONTACT\\_ID](#) value. However, this value isn't available when you're creating the [ContentProviderOperation](#) for the data row, because you haven't yet applied the [ContentProviderOperation](#) for the raw contact row. To work around this, the [ContentProviderOperation.Builder](#) class has the method [WithValueBackReference\(\)](#). This method allows you to insert or modify a column with the result of a previous operation.

The [WithValueBackReference\(\)](#) method has two arguments:

### key

The key of a key-value pair. The value of this argument should be the name of a column in the table that you're modifying.

### previousResult

The 0-based index of a value in the array of [ContentProviderResult](#) objects from [applyBatch\(\)](#). As the batch operations are applied, the result of each operation is stored in an intermediate array of results. The `previousResult` value is the index of one of these results, which is retrieved and stored with the `key` value. This allows you to insert a new raw contact record and get back its [\\_ID](#) value, then make a "back reference" to the value when you add a [ContactsContract.Data](#) row.

The entire result array is created when you first call [applyBatch\(\)](#), with a size equal to the size of the [ArrayList](#) of [ContentProviderOperation](#) objects you provide. However, all the elements in the result array are set to `null`, and if you try to do a back reference to a result for an operation that hasn't yet been applied, [WithValueBackReference\(\)](#) throws an [Exception](#).

The following snippets show how to insert a new raw contact and data in batch. They includes code that establishes a yield point and uses a back reference. The snippets are an expanded version of the `createContactEntry()` method, which is part of the `ContactAdder` class in the [Contact Manager](#) sample application.

The first snippet retrieves contact data from the UI. At this point, the user has already selected the account for which the new raw contact should be added.

```
// Creates a contact entry from the current UI values, using the currently-selected account
protected void createContactEntry() {
```

```

/*
 * Gets values from the UI
 */
String name = mContactNameEditText.getText().toString();
String phone = mContactPhoneEditText.getText().toString();
String email = mContactEmailEditText.getText().toString();

int phoneType = mContactPhoneTypes.get(
    mContactPhoneTypeSpinner.getSelectedItemPosition());

int emailType = mContactEmailTypes.get(
    mContactEmailTypeSpinner.getSelectedItemPosition());

```

The next snippet creates an operation to insert the raw contact row into the [ContactsContract.RawContacts](#) table:

```

/*
 * Prepares the batch operation for inserting a new raw contact and its data.
 * the Contacts Provider does not have any data for this person, you can't
 * only a raw contact. The Contacts Provider will then add a Contact automatically.
 */

// Creates a new array of ContentProviderOperation objects.
ArrayList<ContentProviderOperation> ops =
    new ArrayList<ContentProviderOperation>();

/*
 * Creates a new raw contact with its account type (server type) and account
 * (user's account). Remember that the display name is not stored in this raw
 * StructuredName data row. No other data is required.
 */
ContentProviderOperation.Builder op =
    ContentProviderOperation.newInsert(ContactsContract.RawContacts.CONTENT_URI)
        .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, mSelectedAccountType)
        .withValue(ContactsContract.RawContacts.ACCOUNT_NAME, mSelectedAccountName);

// Builds the operation and adds it to the array of operations
ops.add(op.build());

```

Next, the code creates data rows for the display name, phone, and email rows.

Each operation builder object uses [WithValueBackReference\(\)](#) to get the [RAW\\_CONTACT\\_ID](#). The reference points back to the [ContentProviderResult](#) object from the first operation, which adds the raw contact row and returns its new [ID](#) value. As a result, each data row is automatically linked by its [RAW\\_CONTACT\\_ID](#) to the new [ContactsContract.RawContacts](#) row to which it belongs.

The [ContentProviderOperation.Builder](#) object that adds the email row is flagged with [withYieldAllowed\(\)](#), which sets a yield point:

```

// Creates the display name for the new raw contact, as a StructuredName data row
op =
    ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
        /*
         * withValueBackReference sets the value of the first argument to the
         * RAW_CONTACT_ID returned by the previous ContentProviderOperation
         */
        .withValueBackReference(0, op);

```

```
* the ContentProviderResult indexed by the second argument. In this
* call, the raw contact ID column of the StructuredName data row is the
* value of the result returned by the first operation, which is then
* actually adds the raw contact row.
*/
.writeValueBackReference(ContactContract.Data.RAW_CONTACT_ID, 0)

// Sets the data row's MIME type to StructuredName
.writeValue(ContactContract.Data.MIMETYPE,
    ContactContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE)

// Sets the data row's display name to the name in the UI.
.writeValue(ContactContract.CommonDataKinds.StructuredName.DISPLAY_NAME)

// Builds the operation and adds it to the array of operations
ops.add(op.build());

// Inserts the specified phone number and type as a Phone data row
op =
    ContentProviderOperation.newInsert(ContactContract.Data.CONTENT_URI)
/*
 * Sets the value of the raw contact id column to the new raw contact
 * by the first operation in the batch.
*/
.writeValueBackReference(ContactContract.Data.RAW_CONTACT_ID, 0)

// Sets the data row's MIME type to Phone
.writeValue(ContactContract.Data.MIMETYPE,
    ContactContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE)

// Sets the phone number and type
.writeValue(ContactContract.CommonDataKinds.Phone.NUMBER, phone)
.writeValue(ContactContract.CommonDataKinds.Phone.TYPE, phoneType);

// Builds the operation and adds it to the array of operations
ops.add(op.build());

// Inserts the specified email and type as a Phone data row
op =
    ContentProviderOperation.newInsert(ContactContract.Data.CONTENT_URI)
/*
 * Sets the value of the raw contact id column to the new raw contact
 * by the first operation in the batch.
*/
.writeValueBackReference(ContactContract.Data.RAW_CONTACT_ID, 0)

// Sets the data row's MIME type to Email
.writeValue(ContactContract.Data.MIMETYPE,
    ContactContract.CommonDataKinds.Email.CONTENT_ITEM_TYPE)

// Sets the email address and type
.writeValue(ContactContract.CommonDataKinds.Email.ADDRESS, email)
.writeValue(ContactContract.CommonDataKinds.Email.TYPE, emailType);
```

```

/*
 * Demonstrates a yield point. At the end of this insert, the batch operation
 * will yield priority to other threads. Use after every set of operations
 * single contact, to avoid degrading performance.
 */
op.withYieldAllowed(true);

// Builds the operation and adds it to the array of operations
ops.add(op.build());

```

The last snippet shows the call to [applyBatch\(\)](#) that inserts the new raw contact and data rows.

```

// Ask the Contacts Provider to create a new contact
Log.d(TAG, "Selected account: " + mSelectedAccount.getName() + " (" +
        mSelectedAccount.getType() + ")");
Log.d(TAG, "Creating contact: " + name);

/*
 * Applies the array of ContentProviderOperation objects in batch. The result
 * discarded.
 */
try {

    getContentResolver().applyBatch(ContactsContract.AUTHORITY, ops);
} catch (Exception e) {

    // Display a warning
    Context ctx = getApplicationContext();

    CharSequence txt = getString(R.string.contactCreationFailure);
    int duration = Toast.LENGTH_SHORT;
    Toast toast = Toast.makeText(ctx, txt, duration);
    toast.show();

    // Log exception
    Log.e(TAG, "Exception encountered while inserting contact: " + e);
}
}

```

Batch operations also allow you to implement **optimistic concurrency control**, a method of applying modification transactions without having to lock the underlying repository. To use this method, you apply the transaction and then check for other modifications that may have been made at the same time. If you find an inconsistent modification has occurred, you roll back your transaction and retry it.

Optimistic concurrency control is useful for a mobile device, where there's only one user at a time, and simultaneous accesses to a data repository are rare. Because locking isn't used, no time is wasted on setting locks or waiting for other transactions to release their locks.

To use optimistic concurrency control while updating a single [ContactsContract.RawContacts](#) row, follow these steps:

1. Retrieve the raw contact's [VERSION](#) column along with the other data you retrieve.

2. Create a `ContentProviderOperation.Builder` object suitable for enforcing a constraint, using the method `newAssertQuery(Uri)`. For the content URI, use `RawContacts.CONTENT_URI` with the raw contact's `_ID` appended to it.
3. For the `ContentProviderOperation.Builder` object, call `WithValue()` to compare the `VERSION` column to the version number you just retrieved.
4. For the same `ContentProviderOperation.Builder`, call `WithExpectedCount()` to ensure that only one row is tested by this assertion.
5. Call `build()` to create the `ContentProviderOperation` object, then add this object as the first object in the `ArrayList` that you pass to `applyBatch()`.
6. Apply the batch transaction.

If the raw contact row is updated by another operation between the time you read the row and the time you attempt to modify it, the "assert" `ContentProviderOperation` will fail, and the entire batch of operations will be backed out. You can then choose to retry the batch or take some other action.

The following snippet demonstrates how to create an "assert" `ContentProviderOperation` after querying for a single raw contact using a `CursorLoader`:

```
/*
 * The application uses CursorLoader to query the raw contacts table. The system
 * will be notified when the load is finished.
 */
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {

    // Gets the raw contact's _ID and VERSION values
    mRawContactID = cursor.getLong(cursor.getColumnIndex(BaseColumns._ID));
    mVersion = cursor.getInt(cursor.getColumnIndex(SyncColumns.VERSION));
}

...
// Sets up a Uri for the assert operation
Uri rawContactUri = ContentUris.withAppendedId(RawContacts.CONTENT_URI, mRawContactID);

// Creates a builder for the assert operation
ContentProviderOperation.Builder assertOp = ContentProviderOperation.netAssertOp();

// Adds the assertions to the assert operation: checks the version and count of rows
assertOpWithValue(SyncColumns.VERSION, mVersion);
assertOpWithExpectedCount(1);

// Creates an ArrayList to hold the ContentProviderOperation objects
ArrayList ops = new ArrayList<ContentProviderOperation>;

ops.add(assertOp.build());

// You would add the rest of your batch operations to "ops" here
...

// Applies the batch. If the assert fails, an Exception is thrown
try
{
    ContentProviderResult[] results =
}
```

```

        getContentResolver().applyBatch(AUTHORITY, ops);

    } catch (OperationApplicationException e) {
        // Actions you want to take if the assert operation fails go here
    }
}

```

## Retrieval and modification with intents

Sending an intent to the device's contacts application allows you to access the Contacts Provider indirectly. The intent starts the device's contacts application UI, in which users can do contacts-related work. With this type of access, users can:

- Pick a contact from a list and have it returned to your app for further work.
- Edit an existing contact's data.
- Insert a new raw contact for any of their accounts.
- Delete a contact or contacts data.

If the user is inserting or updating data, you can collect the data first and send it as part of the intent.

When you use intents to access the Contacts Provider via the device's contacts application, you don't have to write your own UI or code for accessing the provider. You also don't have to request permission to read or write to the provider. The device's contacts application can delegate read permission for a contact to you, and because you're making modifications to the provider through another application, you don't have to have write permissions.

The general process of sending an intent to access a provider is described in detail in the [Content Provider Basics](#) guide in the section "Data access via intents." The action, MIME type, and data values you use for the available tasks are summarized in Table 4, while the extras values you can use with `putExtra()` are listed in the reference documentation for [ContactsContract.Intents.Insert](#):

**Table 4.** Contacts Provider Intents.

Task	Action	Data	MIME type
Pick a contact from a list	<a href="#">ACTION_PICK</a>	One of: <ul style="list-style-type: none"> <li>• <a href="#">Contacts.CONTENT_URI</a>, which displays a list of contacts.</li> <li>• <a href="#">Phone.CONTENT_URI</a>, which displays a list of phone numbers for a raw contact.</li> <li>• <a href="#">StructuredPostal.CONTENT_URI</a>, which displays a list of postal addresses for a raw contact.</li> <li>• <a href="#">Email.CONTENT_URI</a>, which displays a list of email addresses for a raw contact.</li> </ul>	Not used
Insert a new contact	<a href="#">Insert.ACTION</a>	N/A	<a href="#">RawContacts.CONTENT</a> MIME type for a set of raw

## raw contact

Edit a contact [ACTION\\_EDIT](#)

[CONTENT\\_LOOKUP\\_URI](#) for the contact. The editor activity will allow the user to edit any of the data associated with this contact.

Display a picker that can also add data.

[ACTION\\_INSERT\\_OR\\_EDIT](#) N/A

[CONTENT\\_ITEM\\_TYPE](#)

The device's contacts app doesn't allow you to delete a raw contact or any of its data with an intent. Instead, to delete a raw contact, use [ContentResolver.delete\(\)](#) or [ContentProviderOperation.newDelete\(\)](#).

The following snippet shows how to construct and send an intent that inserts a new raw contact and data:

```
// Gets values from the UI
String name = mContactNameEditText.getText().toString();
String phone = mContactPhoneEditText.getText().toString();
String email = mContactEmailEditText.getText().toString();

String company = mCompanyName.getText().toString();
String jobtitle = mJobTitle.getText().toString();
```

```
// Creates a new intent for sending to the device's contacts application
Intent insertIntent = new Intent(ContactsContract.Intents.Insert.ACTION);

// Sets the MIME type to the one expected by the insertion activity
insertIntent.setType(ContactsContract.RawContacts.CONTENT_TYPE);

// Sets the new contact name
insertIntent.putExtra(ContactsContract.Intents.Insert.NAME, name);

// Sets the new company and job title
insertIntent.putExtra(ContactsContract.Intents.Insert.COMPANY, company);
insertIntent.putExtra(ContactsContract.Intents.Insert.JOB_TITLE, jobtitle);

/*
 * Demonstrates adding data rows as an array list associated with the DATA key
 */

// Defines an array list to contain the ContentValues objects for each row
ArrayList<ContentValues> contactData = new ArrayList<ContentValues>();

/*
 * Defines the raw contact row
 */

// Sets up the row as a ContentValues object
ContentValues rawContactRow = new ContentValues();

// Adds the account type and name to the row
rawContactRow.put(ContactsContract.RawContacts.ACCOUNT_TYPE, mSelectedAccount.o
rawContactRow.put(ContactsContract.RawContacts.ACCOUNT_NAME, mSelectedAccount.o

// Adds the row to the array
contactData.add(rawContactRow);

/*
 * Sets up the phone number data row
 */

// Sets up the row as a ContentValues object
ContentValues phoneRow = new ContentValues();

// Specifies the MIME type for this data row (all data rows must be marked by t
phoneRow.put(
    ContactsContract.Data.MIMETYPE,
    ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE
);

// Adds the phone number and its type to the row
phoneRow.put(ContactsContract.CommonDataKinds.Phone.NUMBER, phone);

// Adds the row to the array
contactData.add(phoneRow);
```

```

/*
 * Sets up the email data row
 */

// Sets up the row as a ContentValues object
ContentValues emailRow = new ContentValues();

// Specifies the MIME type for this data row (all data rows must be marked by this)
emailRow.put(
    ContactsContract.Data.MIMETYPE,
    ContactsContract.CommonDataKinds.Email.CONTENT_ITEM_TYPE
);

// Adds the email address and its type to the row
emailRow.put(ContactsContract.CommonDataKinds.Email.ADDRESS, email);

// Adds the row to the array
contactData.add(emailRow);

/*
 * Adds the array to the intent's extras. It must be a parcelable object in order
 * to travel between processes. The device's contacts app expects its key to be
 * Intents.Insert.DATA
 */
insertIntent.putParcelableArrayListExtra(ContactsContract.Inserts.Insert.DATA,
    contactData);

// Send out the intent to start the device's contacts app in its add contact activity
startActivity(insertIntent);

```

## Data integrity

Because the contacts repository contains important and sensitive data that users expect to be correct and up-to-date, the Contacts Provider has well-defined rules for data integrity. It's your responsibility to conform to these rules when you modify contacts data. The important rules are listed here:

### **Always add a [ContactsContract.CommonDataKinds.StructuredName](#) row for every [ContactsContract.RawContacts](#) row you add.**

A [ContactsContract.RawContacts](#) row without a [ContactsContract.CommonDataKinds.StructuredName](#) row in the [ContactsContract.Data](#) table may cause problems during aggregation.

### **Always link new [ContactsContract.Data](#) rows to their parent [ContactsContract.RawContacts](#) row.**

A [ContactsContract.Data](#) row that isn't linked to a [ContactsContract.RawContacts](#) won't be visible in the device's contacts application, and it might cause problems with sync adapters.

### **Change data only for those raw contacts that you own.**

Remember that the Contacts Provider is usually managing data from several different account types/online services. You need to ensure that your application only modifies or deletes data for rows that belong to you, and that it only inserts data with an account type and name that you control.

**Always use the constants defined in [ContactsContract](#) and its subclasses for authorities, content URIs, URI paths, column names, MIME types, and [TYPE](#) values.**

Using these constants helps you to avoid errors. You'll also be notified with compiler warnings if any of the constants is deprecated.

## Custom data rows

By creating and using your own custom MIME types, you can insert, edit, delete, and retrieve your own data rows in the [ContactsContract.Data](#) table. Your rows are limited to using the column defined in [ContactsContract.DataColumns](#), although you can map your own type-specific column names to the default column names. In the device's contacts application, the data for your rows is displayed but can't be edited or deleted, and users can't add additional data. To allow users to modify your custom data rows, you must provide an editor activity in your own application.

To display your custom data, provide a `contacts.xml` file containing a `<ContactsAccountType>` element and one or more of its `<ContactsDataKind>` child elements. This is described in more detail in the section [<ContactsDataKind> element](#).

To learn more about custom MIME types, read the [Creating a Content Provider](#) guide.

## Contacts Provider Sync Adapters

The Contacts Provider is specifically designed for handling **synchronization** of contacts data between a device and an online service. This allows users to download existing data to a new device and upload existing data to a new account. Synchronization also ensures that users have the latest data at hand, regardless of the source of additions and changes. Another advantage of synchronization is that it makes contacts data available even when the device is not connected to the network.

Although you can implement synchronization in a variety of ways, the Android system provides a plug-in synchronization framework that automates the following tasks:

- Checking network availability.
- Scheduling and executing synchronization, based on user preferences.
- Restarting synchronizations that have stopped.

To use this framework, you supply a sync adapter plug-in. Each sync adapter is unique to a service and content provider, but can handle multiple account names for the same service. The framework also allows multiple sync adapters for the same service and provider.

## Sync adapter classes and files

You implement a sync adapter as a subclass of [AbstractThreadingSyncAdapter](#) and install it as part of an Android application. The system learns about the sync adapter from elements in your application manifest, and from a special XML file pointed to by the manifest. The XML file defines the account type for the online service and the authority for the content provider, which together uniquely identify the adapter. The sync adapter does not become active until the user adds an account for the sync adapter's account type and enables synchronization for the content provider the sync adapter syncs with. At that point, the system starts managing the adapter, calling it as necessary to synchronize between the content provider and the server.

**Note:** Using an account type as part of the sync adapter's identification allows the system to detect and group together sync adapters that access different services from the same organization. For example, sync adapters for Google online services all have the same account type `com.google`. When users add a Google account to

their devices, all of the installed sync adapters for Google services are listed together; each sync adapter listed syncs with a different content provider on the device.

Because most services require users to verify their identity before accessing data, the Android system offers an authentication framework that is similar to, and often used in conjunction with, the sync adapter framework. The authentication framework uses plug-in authenticators that are subclasses of [AbstractAccountAuthenticator](#). An authenticator verifies the user's identity in the following steps:

1. Collects the user's name, password or similar information (the user's **credentials**).
2. Sends the credentials to the service
3. Examines the service's reply.

If the service accepts the credentials, the authenticator can store the credentials for later use. Because of the plug-in authenticator framework, the [AccountManager](#) can provide access to any authtokens an authenticator supports and chooses to expose, such as OAuth2 authtokens.

Although authentication is not required, most contacts services use it. However, you're not required to use the Android authentication framework to do authentication.

## Sync adapter implementation

To implement a sync adapter for the Contacts Provider, you start by creating an Android application that contains the following:

### A [Service](#) component that responds to requests from the system to bind to the sync adapter.

When the system wants to run a synchronization, it calls the service's [onBind\(\)](#) method to get an [IBinder](#) for the sync adapter. This allows the system to do cross-process calls to the adapter's methods.

In the [Sample Sync Adapter](#) sample app, the class name of this service is `com.example.android.samplesync.syncadapter.SyncService`.

### The actual sync adapter, implemented as a concrete subclass of [AbstractThreadingSyncAdapter](#).

This class does the work of downloading data from the server, uploading data from the device, and resolving conflicts. The main work of the adapter is done in the method [onPerformSync\(\)](#). This class must be instantiated as a singleton.

In the [Sample Sync Adapter](#) sample app, the sync adapter is defined in the class `com.example.android.samplesync.syncadapter.SyncAdapter`.

### A subclass of [Application](#).

This class acts as a factory for the sync adapter singleton. Use the [onCreate\(\)](#) method to instantiate the sync adapter, and provide a static "getter" method to return the singleton to the [onBind\(\)](#) method of the sync adapter's service.

### Optional: A [Service](#) component that responds to requests from the system for user authentication.

[AccountManager](#) starts this service to begin the authentication process. The service's [onCreate\(\)](#) method instantiates an authenticator object. When the system wants to authenticate a user account for the application's sync adapter, it calls the service's [onBind\(\)](#) method to get an [IBinder](#) for the authenticator. This allows the system to do cross-process calls to the authenticator's methods..

In the [Sample Sync Adapter](#) sample app, the class name of this service is `com.example.android.samplesync.authenticator.AuthenticationService`.

## **Optional: A concrete subclass of [AbstractAccountAuthenticator](#) that handles requests for authentication.**

This class provides methods that the [AccountManager](#) invokes to authenticate the user's credentials with the server. The details of the authentication process vary widely, based on the server technology in use. You should refer to the documentation for your server software to learn more about authentication.

In the [Sample Sync Adapter](#) sample app, the authenticator is defined in the class `com.example.android.samplesync.authenticator.Authenticator`.

## **XML files that define the sync adapter and authenticator to the system.**

The sync adapter and authenticator service components described previously are defined in `<service>` elements in the application manifest. These elements contain `<meta-data>` child elements that provide specific data to the system:

- The `<meta-data>` element for the sync adapter service points to the XML file `res/xml/syncadapter.xml`. In turn, this file specifies a URI for the web service that will be synchronized with the Contacts Provider, and an account type for the web service.
- **Optional:** The `<meta-data>` element for the authenticator points to the XML file `res/xml/authenticator.xml`. In turn, this file specifies the account type that this authenticator supports, as well as UI resources that appear during the authentication process. The account type specified in this element must be the same as the account type specified for the sync adapter.

# Social Stream Data

The [ContactsContract.StreamItems](#) and [ContactsContract.StreamItemPhotos](#) tables manage incoming data from social networks. You can write a sync adapter that adds stream data from your own network to these tables, or you can read stream data from these tables and display it in your own application, or both. With these features, your social networking services and applications can be integrated into Android's social networking experience.

## Social stream text

Stream items are always associated with a raw contact. The `RAW_CONTACT_ID` links to the `_ID` value for the raw contact. The account type and account name of the raw contact are also stored in the stream item row.

Store the data from your stream in the following columns:

### **ACCOUNT\_TYPE**

**Required.** The user's account type for the raw contact associated with this stream item. Remember to set this value when you insert a stream item.

### **ACCOUNT\_NAME**

**Required.** The user's account name for the raw contact associated with this stream item. Remember to set this value when you insert a stream item.

## Identifier columns

**Required.** You must insert the following identifier columns when you insert a stream item:

- `CONTACT_ID`: The `_ID` value of the contact that this stream item is associated with.
- `CONTACT_LOOKUP_KEY`: The `LOOKUP_KEY` value of the contact this stream item is associated with.
- `RAW_CONTACT_ID`: The `_ID` value of the raw contact that this stream item is associated with.

### **COMMENTS**

Optional. Stores summary information that you can display at the beginning of a stream item.

## **TEXT**

The text of the stream item, either the content that was posted by the source of the item, or a description of some action that generated the stream item. This column can contain any formatting and embedded resource images that can be rendered by [fromHtml\(\)](#). The provider may truncate or ellipsize long content, but it will try to avoid breaking tags.

## **TIMESTAMP**

A text string containing the time the stream item was inserted or updated, in the form of *milliseconds* since epoch. Applications that insert or update stream items are responsible for maintaining this column; it is not automatically maintained by the Contacts Provider.

To display identifying information for your stream items, use the [RES\\_ICON](#), [RES\\_LABEL](#), and [RES\\_PACKAGE](#) to link to resources in your application.

The [ContactsContract.StreamItems](#) table also contains the columns [SYNC1](#) through [SYNC4](#) for the exclusive use of sync adapters.

## **Social stream photos**

The [ContactsContract.StreamItemPhotos](#) table stores photos associated with a stream item. The table's [STREAM\\_ITEM\\_ID](#) column links to values in the [\\_ID](#) column of [ContactsContract.StreamItems](#) table. Photo references are stored in the table in these columns:

### **PHOTO column (a BLOB).**

A binary representation of the photo, resized by the provider for storage and display. This column is available for backwards compatibility with previous versions of the Contacts Provider that used it for storing photos. However, in the current version you should not use this column to store photos. Instead, use either [PHOTO\\_FILE\\_ID](#) or [PHOTO\\_URI](#) (both of which are described in the following points) to store photos in a file. This column now contains a thumbnail of the photo, which is available for reading.

### **PHOTO\_FILE\_ID**

A numeric identifier of a photo for a raw contact. Append this value to the constant [DisplayPhoto.CONTENT\\_URI](#) to get a content URI pointing to a single photo file, and then call [openAssetFileDescriptor\(\)](#) to get a handle to the photo file.

### **PHOTO\_URI**

A content URI pointing directly to the photo file for the photo represented by this row. Call [openAssetFileDescriptor\(\)](#) with this URI to get a handle to the photo file.

## **Using the social stream tables**

These tables work the same as the other main tables in the Contacts Provider, except that:

- These tables require additional access permissions. To read from them, your application must have the permission [READ\\_SOCIAL\\_STREAM](#). To modify them, your application must have the permission [WRITE\\_SOCIAL\\_STREAM](#).
- For the [ContactsContract.StreamItems](#) table, the number of rows stored for each raw contact is limited. Once this limit is reached, the Contacts Provider makes space for new stream item rows by automatically deleting the rows having the oldest [TIMESTAMP](#). To get the limit, issue a query to the content URI [CONTENT\\_LIMIT\\_URI](#). You can leave all the arguments other than the content URI set to null. The query returns a Cursor containing a single row, with the single column [MAX\\_ITEMS](#).

The class [ContactsContract.StreamItems.StreamItemPhotos](#) defines a sub-table of [ContactsContract.StreamItemPhotos](#) containing the photo rows for a single stream item.

## Social stream interactions

The social stream data managed by the Contacts Provider, in conjunction with the device's contacts application, offers a powerful way to connect your social networking system with existing contacts. The following features are available:

- By syncing your social networking service to the Contacts Provider with a sync adapter, you can retrieve recent activity for a user's contacts and store it in the [ContactsContract.StreamItems](#) and [ContactsContract.StreamItemPhotos](#) tables for later use.
- Besides regular synchronization, you can trigger your sync adapter to retrieve additional data when the user selects a contact to view. This allows your sync adapter to retrieve high-resolution photos and the most recent stream items for the contact.
- By registering a notification with the device's contacts application and the Contacts Provider, you can *receive* an intent when a contact is viewed, and at that point update the contact's status from your service. This approach may be faster and use less bandwidth than doing a full sync with a sync adapter.
- Users can add a contact to your social networking service while looking at the contact in the device's contacts application. You enable this with the "invite contact" feature, which you enable with a combination of an activity that adds an existing contact to your network, and an XML file that provides the device's contacts application and the Contacts Provider with the details of your application.

Regular synchronization of stream items with the Contacts Provider is the same as other synchronizations. To learn more about synchronization, see the section [Contacts Provider Sync Adapters](#). Registering notifications and inviting contacts are covered in the next two sections.

## Registering to handle social networking views

To register your sync adapter to receive notifications when the user views a contact that's managed by your sync adapter:

1. Create a file named `contacts.xml` in your project's `res/xml/` directory. If you already have this file, you can skip this step.
2. In this file, add the element `<ContactsAccountType` `xmlns:android="http://schemas.android.com/apk/res/android">`. If this element already exists, you can skip this step.
3. To register a service that is notified when the user opens a contact's detail page in the device's contacts application, add the attribute `viewContactNotifyService="serviceclass"` to the element, where `serviceclass` is the fully-qualified classname of the service that should receive the intent from the device's contacts application. For the notifier service, use a class that extends [IntentService](#), to allow the service to receive intents. The data in the incoming intent contains the content URI of the raw contact the user clicked. From the notifier service, you can bind to and then call your sync adapter to update the data for the raw contact.

To register an activity to be called when the user clicks on a stream item or photo or both:

1. Create a file named `contacts.xml` in your project's `res/xml/` directory. If you already have this file, you can skip this step.
2. In this file, add the element `<ContactsAccountType` `xmlns:android="http://schemas.android.com/apk/res/android">`. If this element already exists, you can skip this step.

3. To register one of your activities to handle the user clicking on a stream item in the device's contacts application, add the attribute `viewStreamItemActivity="activityclass"` to the element, where `activityclass` is the fully-qualified classname of the activity that should receive the intent from the device's contacts application.
4. To register one of your activities to handle the user clicking on a stream photo in the device's contacts application, add the attribute `viewStreamItemPhotoActivity="activityclass"` to the element, where `activityclass` is the fully-qualified classname of the activity that should receive the intent from the device's contacts application.

The `<ContactsAccountType>` element is described in more detail in the section [<ContactsAccountType> element](#).

The incoming intent contains the content URI of the item or photo that the user clicked. To have separate activities for text items and for photos, use both attributes in the same file.

## Interacting with your social networking service

Users don't have to leave the device's contacts application to invite a contact to your social networking site. Instead, you can have the device's contacts app send an intent for inviting the contact to one of your activities. To set this up:

1. Create a file named `contacts.xml` in your project's `res/xml/` directory. If you already have this file, you can skip this step.
2. In this file, add the element `<ContactsAccountType xmlns:android="http://schemas.android.com/apk/res/android">`. If this element already exists, you can skip this step.
3. Add the following attributes:
  - `inviteContactActivity="activityclass"`
  - `inviteContactActionLabel="@string/invite_action_label"`

The `activityclass` value is the fully-qualified classname of the activity that should receive the intent. The `invite_action_label` value is a text string that's displayed in the **Add Connection** menu in the device's contacts application.

**Note:** `ContactsSource` is a deprecated tag name for `ContactsAccountType`.

## contacts.xml reference

The file `contacts.xml` contains XML elements that control the interaction of your sync adapter and application with the contacts application and the Contacts Provider. These elements are described in the following sections.

### <ContactsAccountType> element

The `<ContactsAccountType>` element controls the interaction of your application with the contacts application. It has the following syntax:

```
<ContactsAccountType
    xmlns:android="http://schemas.android.com/apk/res/android"
    inviteContactActivity="activity_name"
    inviteContactActionLabel="invite_command_text"
    viewContactNotifyService="view_notify_service"
    viewGroupActivity="group_view_activity"
    viewGroupActionLabel="group_action_text"
```

```
viewStreamItemActivity="viewstream_activity_name"  
viewStreamItemPhotoActivity="viewphotostream_activity_name">
```

**contained in:**

res/xml/contacts.xml

**can contain:**

```
<ContactsDataKind>
```

**Description:**

Declares Android components and UI labels that allow users to invite one of their contacts to a social network, notify users when one of their social networking streams is updated, and so forth.

Notice that the attribute prefix `android:` is not necessary for the attributes of `<ContactsAccountType>`.

**Attributes:**

**inviteContactActivity**

The fully-qualified class name of the activity in your application that you want to activate when the user selects **Add connection** from the device's contacts application.

**inviteContactActionLabel**

A text string that is displayed for the activity specified in `inviteContactActivity`, in the **Add connection** menu. For example, you can use the string "Follow in my network". You can use a string resource identifier for this label.

**viewContactNotifyService**

The fully-qualified class name of a service in your application that should receive notifications when the user views a contact. This notification is sent by the device's contacts application; it allows your application to postpone data-intensive operations until they're needed. For example, your application can respond to this notification by reading in and displaying the contact's high-resolution photo and most recent social stream items. This feature is described in more detail in the section [Social stream interactions](#). You can see an example of the notification service in the `NotifierService.java` file in the [SampleSyncAdapter](#) sample app.

**viewGroupActivity**

The fully-qualified class name of an activity in your application that can display group information. When the user clicks the group label in the device's contacts application, the UI for this activity is displayed.

**viewGroupActionLabel**

The label that the contacts application displays for a UI control that allows the user to look at groups in your application.

For example, if you install the Google+ application on your device and you sync Google+ with the contacts application, you'll see Google+ circles listed as groups in your contacts application's **Groups** tab. If you click on a Google+ circle, you'll see people in that circle listed as a "group". At the top of the display, you'll see a Google+ icon; if you click it, control switches to the Google+ app. The contacts application does this with the `viewGroupActivity`, using the Google+ icon as the value of `viewGroupActionLabel`.

A string resource identifier is allowed for this attribute.

## **viewStreamItemActivity**

The fully-qualified class name of an activity in your application that the device's contacts application launches when the user clicks a stream item for a raw contact.

## **viewStreamItemPhotoActivity**

The fully-qualified class name of an activity in your application that the device's contacts application launches when the user clicks a photo in the stream item for a raw contact.

## **<ContactsDataKind> element**

The `<ContactsDataKind>` element controls the display of your application's custom data rows in the contacts application's UI. It has the following syntax:

```
<ContactsDataKind
    android:mimeType="MIMETYPE"
    android:icon="icon_resources"
    android:summaryColumn="column_name"
    android:detailColumn="column_name">
```

### **contained in:**

`<ContactsAccountType>`

### **Description:**

Use this element to have the contacts application display the contents of a custom data row as part of the details of a raw contact. Each `<ContactsDataKind>` child element of `<ContactsAccountType>` represents a type of custom data row that your sync adapter adds to the [ContactsContract.Data](#) table. Add one `<ContactsDataKind>` element for each custom MIME type you use. You don't have to add the element if you have a custom data row for which you don't want to display data.

### **Attributes:**

#### **android:mimeType**

The custom MIME type you've defined for one of your custom data row types in the [ContactsContract.Data](#) table. For example, the value `vnd.android.cursor.item/vnd.example.locationstatus` could be a custom MIME type for a data row that records a contact's last known location.

#### **android:icon**

An Android [drawable resource](#) that the contacts application displays next to your data. Use this to indicate to the user that the data comes from your service.

#### **android:summaryColumn**

The column name for the first of two values retrieved from the data row. The value is displayed as the first line of the entry for this data row. The first line is intended to be used as a summary of the data, but that is optional. See also [android:detailColumn](#).

#### **android:detailColumn**

The column name for the second of two values retrieved from the data row. The value is displayed as the second line of the entry for this data row. See also `android:summaryColumn`.

# Additional Contacts Provider Features

Besides the main features described in previous sections, the Contacts Provider offers these useful features for working with contacts data:

- Contact groups
- Photo features

## Contact groups

The Contacts Provider can optionally label collections of related contacts with **group** data. If the server associated with a user account wants to maintain groups, the sync adapter for the account's account type should transfer groups data between the Contacts Provider and the server. When users add a new contact to the server and then put this contact in a new group, the sync adapter must add the new group to the [ContactsContract.Groups](#) table. The group or groups a raw contact belongs to are stored in the [ContactsContract.Data](#) table, using the [ContactsContract.CommonDataKinds.GroupMembership](#) MIME type.

If you're designing a sync adapter that will add raw contact data from server to the Contacts Provider, and you aren't using groups, then you need to tell the Provider to make your data visible. In the code that is executed when a user adds an account to the device, update the [ContactsContract.Settings](#) row that the Contacts Provider adds for the account. In this row, set the value of the [Settings.UNGROUPED\\_VISIBLE](#) column to 1. When you do this, the Contacts Provider will always make your contacts data visible, even if you don't use groups.

## Contact photos

The [ContactsContract.Data](#) table stores photos as rows with MIME type [Photo.CONTENT\\_ITEM\\_TYPE](#). The row's [CONTACT\\_ID](#) column is linked to the [ID](#) column of the raw contact to which it belongs. The class [ContactsContract.Contacts.Photo](#) defines a sub-table of [ContactsContract.Contacts](#) containing photo information for a contact's primary photo, which is the primary photo of the contact's primary raw contact. Similarly, the class [ContactsContract.RawContacts.DisplayPhoto](#) defines a sub-table of [ContactsContract.RawContacts](#) containing photo information for a raw contact's primary photo.

The reference documentation for [ContactsContract.Contacts.Photo](#) and [ContactsContract.RawContacts.DisplayPhoto](#) contain examples of retrieving photo information. There is no convenience class for retrieving the primary thumbnail for a raw contact, but you can send a query to the [ContactsContract.Data](#) table, selecting on the raw contact's [ID](#), the [Photo.CONTENT\\_ITEM\\_TYPE](#), and the [IS\\_PRIMARY](#) column to find the raw contact's primary photo row.

Social stream data for a person may also include photos. These are stored in the [ContactsContract.StreamItemPhotos](#) table, which is described in more detail in the section [Social stream photos](#).

# Intents and Intent Filters

## In this document

1. [Intent Objects](#)
2. [Intent Resolution](#)
3. [Intent filters](#)
4. [Common cases](#)
5. [Using intent matching](#)
6. [Note Pad Example](#)

## Key classes

1. [Intent](#)
2. [IntentFilter](#)
3. [BroadcastReceiver](#)
4. [PackageManager](#)

Three of the core components of an application — activities, services, and broadcast receivers — are activated through messages, called *intents*. Intent messaging is a facility for late run-time binding between components in the same or different applications. The intent itself, an [Intent](#) object, is a passive data structure holding an abstract description of an operation to be performed — or, often in the case of broadcasts, a description of something that has happened and is being announced. There are separate mechanisms for delivering intents to each type of component:

- An Intent object is passed to [Context.startActivity\(\)](#) or [Activity.startActivityForResult\(\)](#) to launch an activity or get an existing activity to do something new. (It can also be passed to [Activity.setResult\(\)](#) to return information to the activity that called `startActivityForResult()`.)
- An Intent object is passed to [Context.startService\(\)](#) to initiate a service or deliver new instructions to an ongoing service. Similarly, an intent can be passed to [Context.bindService\(\)](#) to establish a connection between the calling component and a target service. It can optionally initiate the service if it's not already running.
- Intent objects passed to any of the broadcast methods (such as [Context.sendBroadcast\(\)](#), [Context.sendOrderedBroadcast\(\)](#), or [Context.sendStickyBroadcast\(\)](#)) are delivered to all interested broadcast receivers. Many kinds of broadcasts originate in system code.

In each case, the Android system finds the appropriate activity, service, or set of broadcast receivers to respond to the intent, instantiating them if necessary. There is no overlap within these messaging systems: Broadcast intents are delivered only to broadcast receivers, never to activities or services. An intent passed to `startActivity()` is delivered only to an activity, never to a service or broadcast receiver, and so on.

This document begins with a description of Intent objects. It then describes the rules Android uses to map intents to components — how it resolves which component should receive an intent message. For intents that don't explicitly name a target component, this process involves testing the Intent object against *intent filters* associated with potential targets.

# Intent Objects

An [Intent](#) object is a bundle of information. It contains information of interest to the component that receives the intent (such as the action to be taken and the data to act on) plus information of interest to the Android system (such as the category of component that should handle the intent and instructions on how to launch a target activity). Principally, it can contain the following:

## Component name

The name of the component that should handle the intent. This field is a [ComponentName](#) object — a combination of the fully qualified class name of the target component (for example "com.example.project.app.FreneticActivity") and the package name set in the manifest file of the application where the component resides (for example, "com.example.project"). The package part of the component name and the package name set in the manifest do not necessarily have to match.

The component name is optional. If it is set, the Intent object is delivered to an instance of the designated class. If it is not set, Android uses other information in the Intent object to locate a suitable target — see [Intent Resolution](#), later in this document.

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

## Action

A string naming the action to be performed — or, in the case of broadcast intents, the action that took place and is being reported. The Intent class defines a number of action constants, including these:

Constant	Target component	Action
ACTION_CALL	activity	Initiate a phone call.
ACTION_EDIT	activity	Display data for the user to edit.
ACTION_MAIN	activity	Start up as the initial activity of a task, with no data input and no returned output.
ACTION_SYNC	activity	Synchronize data on a server with data on the mobile device.
ACTION_BATTERY_LOW	broadcast receiver	A warning that the battery is low.
ACTION_HEADSET_PLUG	broadcast receiver	A headset has been plugged into the device, or unplugged from it.
ACTION_SCREEN_ON	broadcast receiver	The screen has been turned on.
ACTION_TIMEZONE_CHANGED	broadcast receiver	The setting for the time zone has changed.

See the [Intent](#) class description for a list of pre-defined constants for generic actions. Other actions are defined elsewhere in the Android API. You can also define your own action strings for activating the components in your application. Those you invent should include the application package as a prefix — for example: "com.example.project.SHOW\_COLOR".

The action largely determines how the rest of the intent is structured — particularly the [data](#) and [extras](#) fields — much as a method name determines a set of arguments and a return value. For this reason, it's a good idea to use action names that are as specific as possible, and to couple them tightly to the other fields

of the intent. In other words, instead of defining an action in isolation, define an entire protocol for the Intent objects your components can handle.

The action in an Intent object is set by the [setAction\(\)](#) method and read by [getAction\(\)](#).

## Data

The URI of the data to be acted on and the MIME type of that data. Different actions are paired with different kinds of data specifications. For example, if the action field is ACTION\_EDIT, the data field would contain the URI of the document to be displayed for editing. If the action is ACTION\_CALL, the data field would be a tel: URI with the number to call. Similarly, if the action is ACTION\_VIEW and the data field is an http: URI, the receiving activity would be called upon to download and display whatever data the URI refers to.

When matching an intent to a component that is capable of handling the data, it's often important to know the type of data (its MIME type) in addition to its URI. For example, a component able to display image data should not be called upon to play an audio file.

In many cases, the data type can be inferred from the URI — particularly content: URIs, which indicate that the data is located on the device and controlled by a content provider (see the [separate discussion on content providers](#)). But the type can also be explicitly set in the Intent object. The [setData\(\)](#) method specifies data only as a URI, [setType\(\)](#) specifies it only as a MIME type, and [setDataAndType\(\)](#) specifies it as both a URI and a MIME type. The URI is read by [getData\(\)](#) and the type by [getType\(\)](#).

## Category

A string containing additional information about the kind of component that should handle the intent. Any number of category descriptions can be placed in an Intent object. As it does for actions, the Intent class defines several category constants, including these:

Constant	Meaning
CATEGORY_BROWSABLE	The target activity can be safely invoked by the browser to display data referenced by a link — for example, an image or an e-mail message.
CATEGORY_GADGET	The activity can be embedded inside of another activity that hosts gadgets.
CATEGORY_HOME	The activity displays the home screen, the first screen the user sees when the device is turned on or when the <i>Home</i> button is pressed.
CATEGORY_LAUNCHER	The activity can be the initial activity of a task and is listed in the top-level application launcher.
CATEGORY_PREFERENCE	The target activity is a preference panel.

See the [Intent](#) class description for the full list of categories.

The [addCategory\(\)](#) method places a category in an Intent object, [removeCategory\(\)](#) deletes a category previously added, and [getCategories\(\)](#) gets the set of all categories currently in the object.

## Extras

Key-value pairs for additional information that should be delivered to the component handling the intent. Just as some actions are paired with particular kinds of data URIs, some are paired with particular extras. For example, an ACTION\_TIMEZONE\_CHANGED intent has a "time-zone" extra that identifies the new time zone, and ACTION\_HEADSET\_PLUG has a "state" extra indicating whether the headset is now plugged in or unplugged, as well as a "name" extra for the type of headset. If you were to invent a SHOW\_COLOR action, the color value would be set in an extra key-value pair.

The Intent object has a series of `put...()` methods for inserting various types of extra data and a similar set of `get...()` methods for reading the data. These methods parallel those for [Bundle](#) objects. In fact, the extras can be installed and read as a Bundle using the [putExtras\(\)](#) and [getExtras\(\)](#) methods.

## Flags

Flags of various sorts. Many instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's launched (for example, whether it belongs in the list of recent activities). All these flags are defined in the Intent class.

The Android system and the applications that come with the platform employ Intent objects both to send out system-originated broadcasts and to activate system-defined components. To see how to structure an intent to activate a system component, consult the [list of intents](#) in the reference.

## Intent Resolution

Intents can be divided into two groups:

- *Explicit intents* designate the target component by its name (the [component name field](#), mentioned earlier, has a value set). Since component names would generally not be known to developers of other applications, explicit intents are typically used for application-internal messages — such as an activity starting a subordinate service or launching a sister activity.
- *Implicit intents* do not name a target (the field for the component name is blank). Implicit intents are often used to activate components in other applications.

Android delivers an explicit intent to an instance of the designated target class. Nothing in the Intent object other than the component name matters for determining which component should get the intent.

A different strategy is needed for implicit intents. In the absence of a designated target, the Android system must find the best component (or components) to handle the intent — a single activity or service to perform the requested action or the set of broadcast receivers to respond to the broadcast announcement. It does so by comparing the contents of the Intent object to *intent filters*, structures associated with components that can potentially receive intents. Filters advertise the capabilities of a component and delimit the intents it can handle. They open the component to the possibility of receiving implicit intents of the advertised type. If a component does not have any intent filters, it can receive only explicit intents. A component with filters can receive both explicit and implicit intents.

Only three aspects of an Intent object are consulted when the object is tested against an intent filter:

action  
data (both URI and data type)  
category

The extras and flags play no part in resolving which component receives an intent.

## Intent filters

To inform the system which implicit intents they can handle, activities, services, and broadcast receivers can have one or more intent filters. Each filter describes a capability of the component, a set of intents that the component is willing to receive. It, in effect, filters in intents of a desired type, while filtering out unwanted intents — but only unwanted implicit intents (those that don't name a target class). An explicit intent is always delivered to its target, no matter what it contains; the filter is not consulted. But an implicit intent is delivered to a component only if it can pass through one of the component's filters.

A component has separate filters for each job it can do, each face it can present to the user. For example, the NoteEditor activity of the sample Note Pad application has two filters — one for starting up with a specific note that the user can view or edit, and another for starting with a new, blank note that the user can fill in and save. (All of Note Pad's filters are described in the [Note Pad Example](#) section, later.)

## Filters and security

An intent filter cannot be relied on for security. While it opens a component to receiving only certain kinds of implicit intents, it does nothing to prevent explicit intents from targeting the component. Even though a filter restricts the intents a component will be asked to handle to certain actions and data sources, someone could always put together an explicit intent with a different action and data source, and name the component as the target.

An intent filter is an instance of the [IntentFilter](#) class. However, since the Android system must know about the capabilities of a component before it can launch that component, intent filters are generally not set up in Java code, but in the application's manifest file (AndroidManifest.xml) as [`<intent-filter>`](#) elements. (The one exception would be filters for broadcast receivers that are registered dynamically by calling [`Context.registerReceiver\(\)`](#); they are directly created as IntentFilter objects.)

A filter has fields that parallel the action, data, and category fields of an Intent object. An implicit intent is tested against the filter in all three areas. To be delivered to the component that owns the filter, it must pass all three tests. If it fails even one of them, the Android system won't deliver it to the component — at least not on the basis of that filter. However, since a component can have multiple intent filters, an intent that does not pass through one of a component's filters might make it through on another.

Each of the three tests is described in detail below:

### Action test

An [`<intent-filter>`](#) element in the manifest file lists actions as [`<action>`](#) subelements. For example:

```
<intent-filter . . . >
    <action android:name="com.example.project.SHOW_CURRENT" />
    <action android:name="com.example.project.SHOW_RECENT" />
    <action android:name="com.example.project.SHOW_PENDING" />
    . . .
</intent-filter>
```

As the example shows, while an Intent object names just a single action, a filter may list more than one. The list cannot be empty; a filter must contain at least one [`<action>`](#) element, or it will block all intents.

To pass this test, the action specified in the Intent object must match one of the actions listed in the filter. If the object or the filter does not specify an action, the results are as follows:

- If the filter fails to list any actions, there is nothing for an intent to match, so all intents fail the test. No intents can get through the filter.
- On the other hand, an Intent object that doesn't specify an action automatically passes the test — as long as the filter contains at least one action.

### Category test

An [`<intent-filter>`](#) element also lists categories as subelements. For example:

```
<intent-filter . . . >
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    . . .
</intent-filter>
```

Note that the constants described earlier for actions and categories are not used in the manifest file. The full string values are used instead. For instance, the "android.intent.category.BROWSABLE" string in the example above corresponds to the `CATEGORY_BROWSABLE` constant mentioned earlier in this document. Similarly, the string "android.intent.action.EDIT" corresponds to the `ACTION_EDIT` constant.

For an intent to pass the category test, every category in the Intent object must match a category in the filter. The filter can list additional categories, but it cannot omit any that are in the intent.

In principle, therefore, an Intent object with no categories should always pass this test, regardless of what's in the filter. That's mostly true. However, with one exception, Android treats all implicit intents passed to [startActivity\(\)](#) as if they contained at least one category: "android.intent.category.DEFAULT" (the `CATEGORY_DEFAULT` constant). Therefore, activities that are willing to receive implicit intents must include "android.intent.category.DEFAULT" in their intent filters. (Filters with "android.intent.action.MAIN" and "android.intent.category.LAUNCHER" settings are the exception. They mark activities that begin new tasks and that are represented on the launcher screen. They can include "android.intent.category.DEFAULT" in the list of categories, but don't need to.) See [Using intent matching](#), later, for more on these filters.)

## Data test

Like the action and categories, the data specification for an intent filter is contained in a subelement. And, as in those cases, the subelement can appear multiple times, or not at all. For example:

```
<intent-filter . . . >
    <data android:mimeType="video/mpeg" android:scheme="http" . . . />
    <data android:mimeType="audio/mpeg" android:scheme="http" . . . />
    . . .
</intent-filter>
```

Each [`<data>`](#) element can specify a URI and a data type (MIME media type). There are separate attributes — `scheme`, `host`, `port`, and `path` — for each part of the URI:

`scheme://host:port/path`

For example, in the following URI,

`content://com.example.project:200/folder/subfolder/etc`

the scheme is "content", the host is "com.example.project", the port is "200", and the path is "folder/subfolder/etc". The host and port together constitute the URI *authority*; if a host is not specified, the port is ignored.

Each of these attributes is optional, but they are not independent of each other: For an authority to be meaningful, a scheme must also be specified. For a path to be meaningful, both a scheme and an authority must be specified.

When the URI in an Intent object is compared to a URI specification in a filter, it's compared only to the parts of the URI actually mentioned in the filter. For example, if a filter specifies only a scheme, all URIs with that scheme match the filter. If a filter specifies a scheme and an authority but no path, all URIs with the same scheme and authority match, regardless of their paths. If a filter specifies a scheme, an authority, and a path, only URIs with the same scheme, authority, and path match. However, a path specification in the filter can contain wildcards to require only a partial match of the path.

The `type` attribute of a `<data>` element specifies the MIME type of the data. It's more common in filters than a URI. Both the Intent object and the filter can use a "\*" wildcard for the subtype field — for example, "text/\*" or "audio/\*" — indicating any subtype matches.

The data test compares both the URI and the data type in the Intent object to a URI and data type specified in the filter. The rules are as follows:

- a. An Intent object that contains neither a URI nor a data type passes the test only if the filter likewise does not specify any URIs or data types.
- b. An Intent object that contains a URI but no data type (and a type cannot be inferred from the URI) passes the test only if its URI matches a URI in the filter and the filter likewise does not specify a type. This will be the case only for URIs like `mailto:` and `tel:` that do not refer to actual data.
- c. An Intent object that contains a data type but not a URI passes the test only if the filter lists the same data type and similarly does not specify a URI.
- d. An Intent object that contains both a URI and a data type (or a data type can be inferred from the URI) passes the data type part of the test only if its type matches a type listed in the filter. It passes the URI part of the test either if its URI matches a URI in the filter or if it has a `content:` or `file:` URI and the filter does not specify a URI. In other words, a component is presumed to support `content:` and `file:` data if its filter lists only a data type.

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

## Common cases

The last rule shown above for the data test, rule (d), reflects the expectation that components are able to get local data from a file or content provider. Therefore, their filters can list just a data type and do not need to explicitly name the `content:` and `file:` schemes. This is a typical case. A `<data>` element like the following, for example, tells Android that the component can get image data from a content provider and display it:

```
<data android:mimeType="image/*" />
```

Since most available data is dispensed by content providers, filters that specify a data type but not a URI are perhaps the most common.

Another common configuration is filters with a scheme and a data type. For example, a `<data>` element like the following tells Android that the component can get video data from the network and display it:

```
<data android:scheme="http" android:type="video/*" />
```

Consider, for example, what the browser application does when the user follows a link on a web page. It first tries to display the data (as it could if the link was to an HTML page). If it can't display the data, it puts together an implicit intent with the scheme and data type and tries to start an activity that can do the job. If there are no

takers, it asks the download manager to download the data. That puts it under the control of a content provider, so a potentially larger pool of activities (those with filters that just name a data type) can respond.

Most applications also have a way to start fresh, without a reference to any particular data. Activities that can initiate applications have filters with "android.intent.action.MAIN" specified as the action. If they are to be represented in the application launcher, they also specify the "android.intent.category.LAUNCHER" category:

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

## Using intent matching

Intents are matched against intent filters not only to discover a target component to activate, but also to discover something about the set of components on the device. For example, the Android system populates the application launcher, the top-level screen that shows the applications that are available for the user to launch, by finding all the activities with intent filters that specify the "android.intent.action.MAIN" action and "android.intent.category.LAUNCHER" category (as illustrated in the previous section). It then displays the icons and labels of those activities in the launcher. Similarly, it discovers the home screen by looking for the activity with "android.intent.category.HOME" in its filter.

Your application can use intent matching in a similar way. The [PackageManager](#) has a set of `query...` methods that return all components that can accept a particular intent, and a similar series of `resolve...` methods that determine the best component to respond to an intent. For example, [queryIntentActivities\(\)](#) returns a list of all activities that can perform the intent passed as an argument, and [queryIntentServices\(\)](#) returns a similar list of services. Neither method activates the components; they just list the ones that can respond. There's a similar method, [queryBroadcastReceivers\(\)](#), for broadcast receivers.

## Note Pad Example

The Note Pad sample application enables users to browse through a list of notes, view details about individual items in the list, edit the items, and add a new item to the list. This section looks at the intent filters declared in its manifest file. (If you're working offline in the SDK, you can find all the source files for this sample application, including its manifest file, at `<sdk>/samples/NotePad/index.html`. If you're viewing the documentation online, the source files are in the [Tutorials and Sample Code](#) section [here](#).)

In its manifest file, the Note Pad application declares three activities, each with at least one intent filter. It also declares a content provider that manages the note data. Here is the manifest file in its entirety:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.notepad">
    <application android:icon="@drawable/app_notes"
        android:label="@string/app_name" >

        <provider android:name="NotePadProvider"
            android:authorities="com.google.provider.NotePad" />

        <activity android:name="NotesList" android:label="@string/title_notes_1"
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="android.intent.action.PICK" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.GET_CONTENT" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
</activity>

<activity android:name="NoteEditor"
          android:theme="@android:style/Theme.Light"
          android:label="@string/title_note" >
    <intent-filter android:label="@string/resolve_edit">
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="com.android.notepad.action.EDIT_NOTE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.INSERT" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
    </intent-filter>
</activity>

<activity android:name="TitleEditor"
          android:label="@string/title_edit_title"
          android:theme="@android:style/Theme.Dialog">
    <intent-filter android:label="@string/resolve_title">
        <action android:name="com.android.notepad.action.EDIT_TITLE" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.ALTERNATIVE" />
        <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
</activity>

</application>
</manifest>

```

The first activity, NotesList, is distinguished from the other activities by the fact that it operates on a directory of notes (the note list) rather than on a single note. It would generally serve as the initial user interface into the application. It can do three things as described by its three intent filters:

1. <intent-filter>
 

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

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

This filter declares the main entry point into the Note Pad application. The standard `MAIN` action is an entry point that does not require any other information in the Intent (no data specification, for example), and the `LAUNCHER` category says that this entry point should be listed in the application launcher.

2. 

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.PICK" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

This filter declares the things that the activity can do on a directory of notes. It can allow the user to view or edit the directory (via the `VIEW` and `EDIT` actions), or to pick a particular note from the directory (via the `PICK` action).

The `mimeType` attribute of the [`<data>`](#) element specifies the kind of data that these actions operate on. It indicates that the activity can get a Cursor over zero or more items (`vnd.android.cursor.dir`) from a content provider that holds Note Pad data (`vnd.google.note`). The Intent object that launches the activity would include a `content:` URI specifying the exact data of this type that the activity should open.

Note also the `DEFAULT` category supplied in this filter. It's there because the [`Context.startActivity\(\)`](#) and [`Activity.startActivityForResult\(\)`](#) methods treat all intents as if they contained the `DEFAULT` category — with just two exceptions:

- Intents that explicitly name the target activity
- Intents consisting of the `MAIN` action and `LAUNCHER` category

Therefore, the `DEFAULT` category is *required* for all filters — except for those with the `MAIN` action and `LAUNCHER` category. (Intent filters are not consulted for explicit intents.)

3. 

```
<intent-filter>
    <action android:name="android.intent.action.GET_CONTENT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

This filter describes the activity's ability to return a note selected by the user without requiring any specification of the directory the user should choose from. The `GET_CONTENT` action is similar to the `PICK` action. In both cases, the activity returns the URI for a note selected by the user. (In each case, it's returned to the activity that called [`startActivityForResult\(\)`](#) to start the `NoteList` activity.) Here, however, the caller specifies the type of data desired instead of the directory of data the user will be picking from.

The data type, `vnd.android.cursor.item/vnd.google.note`, indicates the type of data the activity can return — a URI for a single note. From the returned URI, the caller can get a Cursor for exactly one item (`vnd.android.cursor.item`) from the content provider that holds Note Pad data (`vnd.google.note`).

In other words, for the `PICK` action in the previous filter, the data type indicates the type of data the activity could display to the user. For the `GET_CONTENT` filter, it indicates the type of data the activity can return to the caller.

Given these capabilities, the following intents will resolve to the `NotesList` activity:

**action: `android.intent.action.MAIN`**

Launches the activity with no data specified.

**action: `android.intent.action.MAIN`**

**category: `android.intent.category.LAUNCHER`**

Launches the activity with no data selected specified. This is the actual intent used by the Launcher to populate its top-level list. All activities with filters that match this action and category are added to the list.

**action: `android.intent.action.VIEW`**

**data: `content://com.google.provider.NotePad/notes`**

Asks the activity to display a list of all the notes under `content://com.google.provider.NotePad/notes`. The user can then browse through the list and get information about the items in it.

**action: `android.intent.action.PICK`**

**data: `content://com.google.provider.NotePad/notes`**

Asks the activity to display a list of the notes under `content://com.google.provider.NotePad/notes`. The user can then pick a note from the list, and the activity will return the URI for that item back to the activity that started the `NoteList` activity.

**action: `android.intent.action.GET_CONTENT`**

**data type: `vnd.android.cursor.item/vnd.google.note`**

Asks the activity to supply a single item of Note Pad data.

The second activity, `NoteEditor`, shows users a single note entry and allows them to edit it. It can do two things as described by its two intent filters:

1. 

```
<intent-filter android:label="@string/resolve_edit">
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="com.android.notepad.action.EDIT_NOTE" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

The first, primary, purpose of this activity is to enable the user to interact with a single note — to either `VIEW` the note or `EDIT` it. (The `EDIT_NOTE` category is a synonym for `EDIT`.) The intent would contain the URI for data matching the MIME type `vnd.android.cursor.item/vnd.google.note` — that is, the URI for a single, specific note. It would typically be a URI that was returned by the `PICK` or `GET_CONTENT` actions of the `NoteList` activity.

As before, this filter lists the `DEFAULT` category so that the activity can be launched by intents that don't explicitly specify the `NoteEditor` class.

2. 

```
<intent-filter>
    <action android:name="android.intent.action.INSERT" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

```
<data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

The secondary purpose of this activity is to enable the user to create a new note, which it will `INSERT` into an existing directory of notes. The intent would contain the URI for data matching the MIME type `vnd.android.cursor.dir/vnd.google.note` — that is, the URI for the directory where the note should be placed.

Given these capabilities, the following intents will resolve to the `NoteEditor` activity:

**action: `android.intent.action.VIEW`**

**data: `content://com.google.provider.NotePad/notes/ID`**

Asks the activity to display the content of the note identified by *ID*. (For details on how `content:` URIs specify individual members of a group, see [Content Providers](#).)

**action: `android.intent.action.EDIT`**

**data: `content://com.google.provider.NotePad/notes/ID`**

Asks the activity to display the content of the note identified by *ID*, and to let the user edit it. If the user saves the changes, the activity updates the data for the note in the content provider.

**action: `android.intent.action.INSERT`**

**data: `content://com.google.provider.NotePad/notes`**

Asks the activity to create a new, empty note in the notes list at `content://com.google.provider.NotePad/notes` and allow the user to edit it. If the user saves the note, its URI is returned to the caller.

The last activity, `TitleEditor`, enables the user to edit the title of a note. This could be implemented by directly invoking the activity (by explicitly setting its component name in the Intent), without using an intent filter. But here we take the opportunity to show how to publish alternative operations on existing data:

```
<intent-filter android:label="@string/resolve_title">
    <action android:name="com.android.notepad.action.EDIT_TITLE" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.ALTERNATIVE" />
    <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

The single intent filter for this activity uses a custom action called `"com.android.notepad.action.EDIT_TITLE"`. It must be invoked on a specific note (data type `vnd.android.cursor.item/vnd.google.note`), like the previous `VIEW` and `EDIT` actions. However, here the activity displays the title contained in the note data, not the content of the note itself.

In addition to supporting the usual `DEFAULT` category, the title editor also supports two other standard categories: `ALTERNATIVE` and `SELECTED_ALTERNATIVE`. These categories identify activities that can be presented to users in a menu of options (much as the `LAUNCHER` category identifies activities that should be presented to user in the application launcher). Note that the filter also supplies an explicit label (via `android:label="@string/resolve_title"`) to better control what users see when presented with this activity as an alternative action to the data they are currently viewing. (For more information on these categories and building options menus, see the [PackageManager.queryIntentActivityOptions\(\)](#) and [Menu.addIntentOptions\(\)](#) methods.)

Given these capabilities, the following intent will resolve to the `TitleEditor` activity:

**action: com.android.notePad.action.EDIT\_TITLE**

**data: content://com.google.provider.NotePad/notes/*ID***

Asks the activity to display the title associated with note *ID*, and allow the user to edit the title.

# Processes and Threads

## Quickview

- Every application runs in its own process and all components of the application run in that process, by default
- Any slow, blocking operations in an activity should be done in a new thread, to avoid slowing down the user interface

## In this document

1. [Processes](#)
  1. [Process lifecycle](#)
2. [Threads](#)
  1. [Worker threads](#)
  2. [Thread-safe methods](#)
3. [Interprocess Communication](#)

When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application with a single thread of execution. By default, all components of the same application run in the same process and thread (called the "main" thread). If an application component starts and there already exists a process for that application (because another component from the application exists), then the component is started within that process and uses the same thread of execution. However, you can arrange for different components in your application to run in separate processes, and you can create additional threads for any process.

This document discusses how processes and threads work in an Android application.

## Processes

By default, all components of the same application run in the same process and most applications should not change this. However, if you find that you need to control which process a certain component belongs to, you can do so in the manifest file.

The manifest entry for each type of component element—[`<activity>`](#), [`<service>`](#), [`<receiver>`](#), and [`<provider>`](#)—supports an `android:process` attribute that can specify a process in which that component should run. You can set this attribute so that each component runs in its own process or so that some components share a process while others do not. You can also set `android:process` so that components of different applications run in the same process—provided that the applications share the same Linux user ID and are signed with the same certificates.

The [`<application>`](#) element also supports an `android:process` attribute, to set a default value that applies to all components.

Android might decide to shut down a process at some point, when memory is low and required by other processes that are more immediately serving the user. Application components running in the process that's killed are consequently destroyed. A process is started again for those components when there's again work for them to do.

When deciding which processes to kill, the Android system weighs their relative importance to the user. For example, it more readily shuts down a process hosting activities that are no longer visible on screen, compared to

a process hosting visible activities. The decision whether to terminate a process, therefore, depends on the state of the components running in that process. The rules used to decide which processes to terminate is discussed below.

## Process lifecycle

The Android system tries to maintain an application process for as long as possible, but eventually needs to remove old processes to reclaim memory for new or more important processes. To determine which processes to keep and which to kill, the system places each process into an "importance hierarchy" based on the components running in the process and the state of those components. Processes with the lowest importance are eliminated first, then those with the next lowest importance, and so on, as necessary to recover system resources.

There are five levels in the importance hierarchy. The following list presents the different types of processes in order of importance (the first process is *most important* and is *killed last*):

### 1. Foreground process

A process that is required for what the user is currently doing. A process is considered to be in the foreground if any of the following conditions are true:

- It hosts an [Activity](#) that the user is interacting with (the [Activity](#)'s [onResume\(\)](#) method has been called).
- It hosts a [Service](#) that's bound to the activity that the user is interacting with.
- It hosts a [Service](#) that's running "in the foreground"—the service has called [startForeground\(\)](#).
- It hosts a [Service](#) that's executing one of its lifecycle callbacks ([onCreate\(\)](#), [onStart\(\)](#), or [onDestroy\(\)](#)).
- It hosts a [BroadcastReceiver](#) that's executing its [onReceive\(\)](#) method.

Generally, only a few foreground processes exist at any given time. They are killed only as a last resort—if memory is so low that they cannot all continue to run. Generally, at that point, the device has reached a memory paging state, so killing some foreground processes is required to keep the user interface responsive.

### 2. Visible process

A process that doesn't have any foreground components, but still can affect what the user sees on screen. A process is considered to be visible if either of the following conditions are true:

- It hosts an [Activity](#) that is not in the foreground, but is still visible to the user (its [onPause\(\)](#) method has been called). This might occur, for example, if the foreground activity started a dialog, which allows the previous activity to be seen behind it.
- It hosts a [Service](#) that's bound to a visible (or foreground) activity.

A visible process is considered extremely important and will not be killed unless doing so is required to keep all foreground processes running.

### 3. Service process

A process that is running a service that has been started with the [startService\(\)](#) method and does not fall into either of the two higher categories. Although service processes are not directly tied to anything the user sees, they are generally doing things that the user cares about (such as playing music in the background or downloading data on the network), so the system keeps them running unless there's not enough memory to retain them along with all foreground and visible processes.

## 4. Background process

A process holding an activity that's not currently visible to the user (the activity's [onStop\(\)](#) method has been called). These processes have no direct impact on the user experience, and the system can kill them at any time to reclaim memory for a foreground, visible, or service process. Usually there are many background processes running, so they are kept in an LRU (least recently used) list to ensure that the process with the activity that was most recently seen by the user is the last to be killed. If an activity implements its lifecycle methods correctly, and saves its current state, killing its process will not have a visible effect on the user experience, because when the user navigates back to the activity, the activity restores all of its visible state. See the [Activities](#) document for information about saving and restoring state.

## 5. Empty process

A process that doesn't hold any active application components. The only reason to keep this kind of process alive is for caching purposes, to improve startup time the next time a component needs to run in it. The system often kills these processes in order to balance overall system resources between process caches and the underlying kernel caches.

Android ranks a process at the highest level it can, based upon the importance of the components currently active in the process. For example, if a process hosts a service and a visible activity, the process is ranked as a visible process, not a service process.

In addition, a process's ranking might be increased because other processes are dependent on it—a process that is serving another process can never be ranked lower than the process it is serving. For example, if a content provider in process A is serving a client in process B, or if a service in process A is bound to a component in process B, process A is always considered at least as important as process B.

Because a process running a service is ranked higher than a process with background activities, an activity that initiates a long-running operation might do well to start a [service](#) for that operation, rather than simply create a worker thread—particularly if the operation will likely outlast the activity. For example, an activity that's uploading a picture to a web site should start a service to perform the upload so that the upload can continue in the background even if the user leaves the activity. Using a service guarantees that the operation will have at least "service process" priority, regardless of what happens to the activity. This is the same reason that broadcast receivers should employ services rather than simply put time-consuming operations in a thread.

# Threads

When an application is launched, the system creates a thread of execution for the application, called "main." This thread is very important because it is in charge of dispatching events to the appropriate user interface widgets, including drawing events. It is also the thread in which your application interacts with components from the Android UI toolkit (components from the [android.widget](#) and [android.view](#) packages). As such, the main thread is also sometimes called the UI thread.

The system does *not* create a separate thread for each instance of a component. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread. Consequently, methods that respond to system callbacks (such as [onKeyDown\(\)](#) to report user actions or a lifecycle callback method) always run in the UI thread of the process.

For instance, when the user touches a button on the screen, your app's UI thread dispatches the touch event to the widget, which in turn sets its pressed state and posts an invalidate request to the event queue. The UI thread dequeues the request and notifies the widget that it should redraw itself.

When your app performs intensive work in response to user interaction, this single thread model can yield poor performance unless you implement your application properly. Specifically, if everything is happening in the UI thread, performing long operations such as network access or database queries will block the whole UI. When the thread is blocked, no events can be dispatched, including drawing events. From the user's perspective, the application appears to hang. Even worse, if the UI thread is blocked for more than a few seconds (about 5 seconds currently) the user is presented with the infamous "[application not responding](#)" (ANR) dialog. The user might then decide to quit your application and uninstall it if they are unhappy.

Additionally, the Android UI toolkit is *not* thread-safe. So, you must not manipulate your UI from a worker thread—you must do all manipulation to your user interface from the UI thread. Thus, there are simply two rules to Android's single thread model:

1. Do not block the UI thread
2. Do not access the Android UI toolkit from outside the UI thread

## Worker threads

Because of the single thread model described above, it's vital to the responsiveness of your application's UI that you do not block the UI thread. If you have operations to perform that are not instantaneous, you should make sure to do them in separate threads ("background" or "worker" threads).

For example, below is some code for a click listener that downloads an image from a separate thread and displays it in an [ImageView](#):

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");  
            mImageView.setImageBitmap(b);  
        }  
    }).start();  
}
```

At first, this seems to work fine, because it creates a new thread to handle the network operation. However, it violates the second rule of the single-threaded model: *do not access the Android UI toolkit from outside the UI thread*—this sample modifies the [ImageView](#) from the worker thread instead of the UI thread. This can result in undefined and unexpected behavior, which can be difficult and time-consuming to track down.

To fix this problem, Android offers several ways to access the UI thread from other threads. Here is a list of methods that can help:

- [Activity.runOnUiThread\(Runnable\)](#)
- [View.post\(Runnable\)](#)
- [View.postDelayed\(Runnable, long\)](#)

For example, you can fix the above code by using the [View.post\(Runnable\)](#) method:

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            final Bitmap bitmap = loadImageFromNetwork("http://example.com/image.png");  
            mImageView.post(new Runnable() {  
                public void run() {  
                    mImageView.setImageBitmap(bitmap);  
                }  
            });  
        }  
    }).start();  
}
```

```

        }
    } );
}).start();
}
}

```

Now this implementation is thread-safe: the network operation is done from a separate thread while the [ImageView](#) is manipulated from the UI thread.

However, as the complexity of the operation grows, this kind of code can get complicated and difficult to maintain. To handle more complex interactions with a worker thread, you might consider using a [Handler](#) in your worker thread, to process messages delivered from the UI thread. Perhaps the best solution, though, is to extend the [AsyncTask](#) class, which simplifies the execution of worker thread tasks that need to interact with the UI.

## Using AsyncTask

[AsyncTask](#) allows you to perform asynchronous work on your user interface. It performs the blocking operations in a worker thread and then publishes the results on the UI thread, without requiring you to handle threads and/or handlers yourself.

To use it, you must subclass [AsyncTask](#) and implement the [doInBackground\(\)](#) callback method, which runs in a pool of background threads. To update your UI, you should implement [onPostExecute\(\)](#), which delivers the result from [doInBackground\(\)](#) and runs in the UI thread, so you can safely update your UI. You can then run the task by calling [execute\(\)](#) from the UI thread.

For example, you can implement the previous example using [AsyncTask](#) this way:

```

public void onClick(View v) {
    new DownloadImageTask().execute("http://example.com/image.png");
}

private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
    /** The system calls this to perform work in a worker thread and
     * delivers it the parameters given to AsyncTask.execute() */
    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

    /** The system calls this to perform work in the UI thread and delivers
     * the result from doInBackground() */
    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}

```

Now the UI is safe and the code is simpler, because it separates the work into the part that should be done on a worker thread and the part that should be done on the UI thread.

You should read the [AsyncTask](#) reference for a full understanding on how to use this class, but here is a quick overview of how it works:

- You can specify the type of the parameters, the progress values, and the final value of the task, using generics
- The method [doInBackground\(\)](#) executes automatically on a worker thread

- `onPreExecute()`, `onPostExecute()`, and `onProgressUpdate()` are all invoked on the UI thread
- The value returned by `doInBackground()` is sent to `onPostExecute()`
- You can call `publishProgress()` at anytime in `doInBackground()` to execute `onProgressUpdate()` on the UI thread
- You can cancel the task at any time, from any thread

**Caution:** Another problem you might encounter when using a worker thread is unexpected restarts in your activity due to a [runtime configuration change](#) (such as when the user changes the screen orientation), which may destroy your worker thread. To see how you can persist your task during one of these restarts and how to properly cancel the task when the activity is destroyed, see the source code for the [Shelves](#) sample application.

## Thread-safe methods

In some situations, the methods you implement might be called from more than one thread, and therefore must be written to be thread-safe.

This is primarily true for methods that can be called remotely—such as methods in a [bound service](#). When a call on a method implemented in an [IBinder](#) originates in the same process in which the [IBinder](#) is running, the method is executed in the caller's thread. However, when the call originates in another process, the method is executed in a thread chosen from a pool of threads that the system maintains in the same process as the [IBinder](#) (it's not executed in the UI thread of the process). For example, whereas a service's [onBind\(\)](#) method would be called from the UI thread of the service's process, methods implemented in the object that [onBind\(\)](#) returns (for example, a subclass that implements RPC methods) would be called from threads in the pool. Because a service can have more than one client, more than one pool thread can engage the same [IBinder](#) method at the same time. [IBinder](#) methods must, therefore, be implemented to be thread-safe.

Similarly, a content provider can receive data requests that originate in other processes. Although the [ContentResolver](#) and [ContentProvider](#) classes hide the details of how the interprocess communication is managed, [ContentProvider](#) methods that respond to those requests—the methods [query\(\)](#), [insert\(\)](#), [delete\(\)](#), [update\(\)](#), and [getType\(\)](#)—are called from a pool of threads in the content provider's process, not the UI thread for the process. Because these methods might be called from any number of threads at the same time, they too must be implemented to be thread-safe.

## Interprocess Communication

Android offers a mechanism for interprocess communication (IPC) using remote procedure calls (RPCs), in which a method is called by an activity or other application component, but executed remotely (in another process), with any result returned back to the caller. This entails decomposing a method call and its data to a level the operating system can understand, transmitting it from the local process and address space to the remote process and address space, then reassembling and reenacting the call there. Return values are then transmitted in the opposite direction. Android provides all the code to perform these IPC transactions, so you can focus on defining and implementing the RPC programming interface.

To perform IPC, your application must bind to a service, using [bindService\(\)](#). For more information, see the [Services](#) developer guide.

# Permissions

## In this document

1. [Security Architecture](#)
2. [Application Signing](#)
3. [User IDs and File Access](#)
4. [Using Permissions](#)
5. [Declaring and Enforcing Permissions](#)
  1. [...in AndroidManifest.xml](#)
  2. [...when Sending Broadcasts](#)
  3. [Other Permission Enforcement](#)
6. [URI Permissions](#)

This document describes how application developers can use the security features provided by Android. A more general [Android Security Overview](#) is provided in the Android Open Source Project.

Android is a privilege-separated operating system, in which each application runs with a distinct system identity (Linux user ID and group ID). Parts of the system are also separated into distinct identities. Linux thereby isolates applications from each other and from the system.

Additional finer-grained security features are provided through a "permission" mechanism that enforces restrictions on the specific operations that a particular process can perform, and per-URI permissions for granting ad-hoc access to specific pieces of data.

## Security Architecture

A central design point of the Android security architecture is that no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or e-mails), reading or writing another application's files, performing network access, keeping the device awake, etc.

Because Android sandboxes applications from each other, applications must explicitly share resources and data. They do this by declaring the *permissions* they need for additional capabilities not provided by the basic sandbox. Applications statically declare the permissions they require, and the Android system prompts the user for consent at the time the application is installed. Android has no mechanism for granting permissions dynamically (at run-time) because it complicates the user experience to the detriment of security.

The application sandbox does not depend on the technology used to build an application. In particular the Dalvik VM is not a security boundary, and any app can run native code (see [the Android NDK](#)). All types of applications — Java, native, and hybrid — are sandboxed in the same way and have the same degree of security from each other.

## Application Signing

All Android applications (.apk files) must be signed with a certificate whose private key is held by their developer. This certificate identifies the author of the application. The certificate does *not* need to be signed by a certificate authority: it is perfectly allowable, and typical, for Android applications to use self-signed certificates. The purpose of certificates in Android is to distinguish application authors. This allows the system to grant or deny applications access to [signature-level permissions](#) and to grant or deny an application's [request to be given the same Linux identity](#) as another application.

# User IDs and File Access

At install time, Android gives each package a distinct Linux user ID. The identity remains constant for the duration of the package's life on that device. On a different device, the same package may have a different UID; what matters is that each package has a distinct UID on a given device.

Because security enforcement happens at the process level, the code of any two packages can not normally run in the same process, since they need to run as different Linux users. You can use the `sharedUserId` attribute in the `AndroidManifest.xml`'s `manifest` tag of each package to have them assigned the same user ID. By doing this, for purposes of security the two packages are then treated as being the same application, with the same user ID and file permissions. Note that in order to retain security, only two applications signed with the same signature (and requesting the same `sharedUserId`) will be given the same user ID.

Any data stored by an application will be assigned that application's user ID, and not normally accessible to other packages. When creating a new file with `getSharedPreferences(String, int)`, `openFileOutput(String, int)`, or `openOrCreateDatabase(String, int, SQLiteDatabase.CursorFactory)`, you can use the `MODE_WORLD_READABLE` and/or `MODE_WORLD_WRITEABLE` flags to allow any other package to read/write the file. When setting these flags, the file is still owned by your application, but its global read and/or write permissions have been set appropriately so any other application can see it.

# Using Permissions

A basic Android application has no permissions associated with it by default, meaning it can not do anything that would adversely impact the user experience or any data on the device. To make use of protected features of the device, you must include in your `AndroidManifest.xml` one or more `<uses-permission>` tags declaring the permissions that your application needs.

For example, an application that needs to monitor incoming SMS messages would specify:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    ...
</manifest>
```

At application install time, permissions requested by the application are granted to it by the package installer, based on checks against the signatures of the applications declaring those permissions and/or interaction with the user. *No* checks with the user are done while an application is running: it either was granted a particular permission when installed, and can use that feature as desired, or the permission was not granted and any attempt to use the feature will fail without prompting the user.

Often times a permission failure will result in a `SecurityException` being thrown back to the application. However, this is not guaranteed to occur everywhere. For example, the `sendBroadcast(Intent)` method checks permissions as data is being delivered to each receiver, after the method call has returned, so you will not receive an exception if there are permission failures. In almost all cases, however, a permission failure will be printed to the system log.

The permissions provided by the Android system can be found at `Manifest.permission`. Any application may also define and enforce its own permissions, so this is not a comprehensive list of all possible permissions.

A particular permission may be enforced at a number of places during your program's operation:

- At the time of a call into the system, to prevent an application from executing certain functions.
- When starting an activity, to prevent applications from launching activities of other applications.
- Both sending and receiving broadcasts, to control who can receive your broadcast or who can send a broadcast to you.
- When accessing and operating on a content provider.
- Binding to or starting a service.

**Caution:** Over time, new restrictions may be added to the platform such that, in order to use certain APIs, your app must request a permission that it previously did not need. Because existing apps assume access to those APIs is freely available, Android may apply the new permission request to the app's manifest to avoid breaking the app on the new platform version. Android makes the decision as to whether an app might need the permission based on the value provided for the [targetSdkVersion](#) attribute. If the value is lower than the version in which the permission was added, then Android adds the permission.

For example, the [WRITE\\_EXTERNAL\\_STORAGE](#) permission was added in API level 4 to restrict access to the shared storage space. If your [targetSdkVersion](#) is 3 or lower, this permission is added to your app on newer versions of Android.

Beware that if this happens to your app, your app listing on Google Play will show these required permissions even though your app might not actually require them.

To avoid this and remove the default permissions you don't need, always update your [targetSdkVersion](#) to be as high as possible. You can see which permissions were added with each release in the [Build.VERSION\\_CODES](#) documentation.

## Declaring and Enforcing Permissions

To enforce your own permissions, you must first declare them in your `AndroidManifest.xml` using one or more [<permission>](#) tags.

For example, an application that wants to control who can start one of its activities could declare a permission for this operation as follows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.me.app.myapp" >
    <permission android:name="com.me.app.myapp.permission.DEADLY_ACTIVITY"
        android:label="@string/permlab_deadlyActivity"
        android:description="@string/permdesc_deadlyActivity"
        android:permissionGroup="android.permission-group.COST MONEY"
        android:protectionLevel="dangerous" />
    ...
</manifest>
```

The [<protectionLevel>](#) attribute is required, telling the system how the user is to be informed of applications requiring the permission, or who is allowed to hold that permission, as described in the linked documentation.

The [<permissionGroup>](#) attribute is optional, and only used to help the system display permissions to the user. You will usually want to set this to either a standard system group (listed in [an-  
droid.Manifest.permission\\_group](#)) or in more rare cases to one defined by yourself. It is preferred to use an existing group, as this simplifies the permission UI shown to the user.

Note that both a label and description should be supplied for the permission. These are string resources that can be displayed to the user when they are viewing a list of permissions ([android:label](#)) or details on a single permission ([android:description](#)). The label should be short, a few words describing the key piece of functionality the permission is protecting. The description should be a couple sentences describing what the permission allows a holder to do. Our convention for the description is two sentences, the first describing the permission, the second warning the user of what bad things can happen if an application is granted the permission.

Here is an example of a label and description for the CALL\_PHONE permission:

```
<string name="permlab_callPhone">directly call phone numbers</string>
<string name="permdesc_callPhone">Allows the application to call
    phone numbers without your intervention. Malicious applications may
    cause unexpected calls on your phone bill. Note that this does not
    allow the application to call emergency numbers.</string>
```

You can look at the permissions currently defined in the system with the Settings app and the shell command adb shell pm list permissions. To use the Settings app, go to Settings > Applications. Pick an app and scroll down to see the permissions that the app uses. For developers, the adb '-s' option displays the permissions in a form similar to how the user will see them:

```
$ adb shell pm list permissions -s
All Permissions:
```

```
Network communication: view Wi-Fi state, create Bluetooth connections, full
Internet access, view network state
```

```
Your location: access extra location provider commands, fine (GPS) location,
mock location sources for testing, coarse (network-based) location
```

```
Services that cost you money: send SMS messages, directly call phone numbers
```

```
...
```

## Enforcing Permissions in AndroidManifest.xml

High-level permissions restricting access to entire components of the system or application can be applied through your `AndroidManifest.xml`. All that this requires is including an [android:permission](#) attribute on the desired component, naming the permission that will be used to control access to it.

[\*\*Activity\*\*](#) permissions (applied to the `<activity>` tag) restrict who can start the associated activity. The permission is checked during [Context.startActivity\(\)](#) and [Activity.startActivityForResult\(\)](#); if the caller does not have the required permission then [SecurityException](#) is thrown from the call.

[\*\*Service\*\*](#) permissions (applied to the `<service>` tag) restrict who can start or bind to the associated service. The permission is checked during [Context.startService\(\)](#), [Context.stopService\(\)](#) and [Context.bindService\(\)](#); if the caller does not have the required permission then [SecurityException](#) is thrown from the call.

[\*\*BroadcastReceiver\*\*](#) permissions (applied to the `<receiver>` tag) restrict who can send broadcasts to the associated receiver. The permission is checked *after* [Context.sendBroadcast\(\)](#) returns, as the system tries to deliver the submitted broadcast to the given receiver. As a result, a permission failure will not result in an exception being thrown back to the caller; it will just not deliver the intent. In the same way, a permission

can be supplied to [Context.registerReceiver\(\)](#) to control who can broadcast to a programmatically registered receiver. Going the other way, a permission can be supplied when calling [ContentText.sendBroadcast\(\)](#) to restrict which BroadcastReceiver objects are allowed to receive the broadcast (see below).

[ContentProvider](#) permissions (applied to the [`<provider>`](#) tag) restrict who can access the data in a [ContentProvider](#). (Content providers have an important additional security facility available to them called [URI permissions](#) which is described later.) Unlike the other components, there are two separate permission attributes you can set: [android:readPermission](#) restricts who can read from the provider, and [android:writePermission](#) restricts who can write to it. Note that if a provider is protected with both a read and write permission, holding only the write permission does not mean you can read from a provider. The permissions are checked when you first retrieve a provider (if you don't have either permission, a SecurityException will be thrown), and as you perform operations on the provider. Using [ContentResolver.query\(\)](#) requires holding the read permission; using [ContentResolver.insert\(\)](#), [ContentResolver.update\(\)](#), [ContentResolver.delete\(\)](#) requires the write permission. In all of these cases, not holding the required permission results in a [SecurityException](#) being thrown from the call.

## Enforcing Permissions when Sending Broadcasts

In addition to the permission enforcing who can send Intents to a registered [BroadcastReceiver](#) (as described above), you can also specify a required permission when sending a broadcast. By calling [ContentText.sendBroadcast\(\)](#) with a permission string, you require that a receiver's application must hold that permission in order to receive your broadcast.

Note that both a receiver and a broadcaster can require a permission. When this happens, both permission checks must pass for the Intent to be delivered to the associated target.

## Other Permission Enforcement

Arbitrarily fine-grained permissions can be enforced at any call into a service. This is accomplished with the [Context.checkSelfPermission\(\)](#) method. Call with a desired permission string and it will return an integer indicating whether that permission has been granted to the current calling process. Note that this can only be used when you are executing a call coming in from another process, usually through an IDL interface published from a service or in some other way given to another process.

There are a number of other useful ways to check permissions. If you have the pid of another process, you can use the Context method [Context.checkSelfPermission\(String, int, int\)](#) to check a permission against that pid. If you have the package name of another application, you can use the direct PackageManager method [PackageManager.checkSelfPermission\(String, String\)](#) to find out whether that particular package has been granted a specific permission.

## URI Permissions

The standard permission system described so far is often not sufficient when used with content providers. A content provider may want to protect itself with read and write permissions, while its direct clients also need to hand specific URIs to other applications for them to operate on. A typical example is attachments in a mail application. Access to the mail should be protected by permissions, since this is sensitive user data. However, if a URI to an image attachment is given to an image viewer, that image viewer will not have permission to open the attachment since it has no reason to hold a permission to access all e-mail.

The solution to this problem is per-URI permissions: when starting an activity or returning a result to an activity, the caller can set [Intent.FLAG\\_GRANT\\_READ\\_URI\\_PERMISSION](#) and/or [In-](#)

`Intent.FLAG_GRANT_WRITE_URI_PERMISSION`. This grants the receiving activity permission access the specific data URI in the Intent, regardless of whether it has any permission to access data in the content provider corresponding to the Intent.

This mechanism allows a common capability-style model where user interaction (opening an attachment, selecting a contact from a list, etc) drives ad-hoc granting of fine-grained permission. This can be a key facility for reducing the permissions needed by applications to only those directly related to their behavior.

The granting of fine-grained URI permissions does, however, require some cooperation with the content provider holding those URIs. It is strongly recommended that content providers implement this facility, and declare that they support it through the `android:grantUriPermissions` attribute or `<grant-uri-permissions>` tag.

More information can be found in the `Context.grantUriPermission()`, `Context.revokeUriPermission()`, and `Context.checkUriPermission()` methods.

# App Widgets

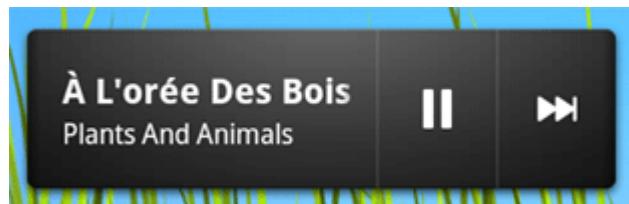
## In this document

1. [The Basics](#)
2. [Declaring an App Widget in the Manifest](#)
3. [Adding the AppWidgetProviderInfo Metadata](#)
4. [Creating the App Widget Layout](#)
5. [Using the AppWidgetProvider Class](#)
  1. [Receiving App Widget broadcast Intents](#)
6. [Creating an App Widget Configuration Activity](#)
  1. [Updating the App Widget from the Configuration Activity](#)
7. [Setting a Preview Image](#)
8. [Enabling App Widgets on the Lockscreen](#)
  1. [Sizing guidelines](#)
9. [Using App Widgets with Collections](#)
  1. [Sample application](#)
  2. [Implementing app widgets with collections](#)
  3. [Keeping Collection Data Fresh](#)

## Key classes

1. [AppWidgetProvider](#)
2. [AppWidgetProviderInfo](#)
3. [AppWidgetManager](#)

App Widgets are miniature application views that can be embedded in other applications (such as the Home screen) and receive periodic updates. These views are referred to as Widgets in the user interface, and you can publish one with an App Widget provider. An application component that is able to hold other App Widgets is called an App Widget host. The screenshot below shows the Music App Widget.



This document describes how to publish an App Widget using an App Widget provider. For a discussion of creating your own [AppWidgetHost](#) to host app widgets, see [App Widget Host](#).

## Widget Design

For information about how to design your app widget, read the [Widgets](#) design guide.

## The Basics

To create an App Widget, you need the following:

### [AppWidgetProviderInfo](#) object

Describes the metadata for an App Widget, such as the App Widget's layout, update frequency, and the AppWidgetProvider class. This should be defined in XML.

## [AppWidgetProvider](#) class implementation

Defines the basic methods that allow you to programmatically interface with the App Widget, based on broadcast events. Through it, you will receive broadcasts when the App Widget is updated, enabled, disabled and deleted.

## View layout

Defines the initial layout for the App Widget, defined in XML.

Additionally, you can implement an App Widget configuration Activity. This is an optional [Activity](#) that launches when the user adds your App Widget and allows him or her to modify App Widget settings at create-time.

The following sections describe how to set up each of these components.

## Declaring an App Widget in the Manifest

First, declare the [AppWidgetProvider](#) class in your application's `AndroidManifest.xml` file. For example:

```
<receiver android:name="ExampleAppWidgetProvider" >
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
    <meta-data android:name="android.appwidget.provider"
              android:resource="@xml/example_appwidget_info" />
</receiver>
```

The `<receiver>` element requires the `android:name` attribute, which specifies the [AppWidgetProvider](#) used by the App Widget.

The `<intent-filter>` element must include an `<action>` element with the `android:name` attribute. This attribute specifies that the [AppWidgetProvider](#) accepts the [ACTION\\_APPWIDGET\\_UPDATE](#) broadcast. This is the only broadcast that you must explicitly declare. The [AppWidgetManager](#) automatically sends all other App Widget broadcasts to the AppWidgetProvider as necessary.

The `<meta-data>` element specifies the [AppWidgetProviderInfo](#) resource and requires the following attributes:

- `android:name` - Specifies the metadata name. Use `android.appwidget.provider` to identify the data as the [AppWidgetProviderInfo](#) descriptor.
- `android:resource` - Specifies the [AppWidgetProviderInfo](#) resource location.

## Adding the AppWidgetProviderInfo Metadata

The [AppWidgetProviderInfo](#) defines the essential qualities of an App Widget, such as its minimum layout dimensions, its initial layout resource, how often to update the App Widget, and (optionally) a configuration Activity to launch at create-time. Define the AppWidgetProviderInfo object in an XML resource using a single `<appwidget-provider>` element and save it in the project's `res/xml/` folder.

For example:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="40dp"
```

```
        android:minHeight="40dp"
        android:updatePeriodMillis="86400000"
        android:previewImage="@drawable/preview"
        android:initialLayout="@layout/example_appwidget"
        android:configure="com.example.android.ExampleAppWidgetConfigure"
        android:resizeMode="horizontal|vertical"
        android:widgetCategory="home_screen|keyguard"
        android:initialKeyguardLayout="@layout/example_keyguard">
    </appwidget-provider>
```

Here's a summary of the `<appwidget-provider>` attributes:

- The values for the `minWidth` and `minHeight` attributes specify the minimum amount of space the App Widget consumes *by default*. The default Home screen positions App Widgets in its window based on a grid of cells that have a defined height and width. If the values for an App Widget's minimum width or height don't match the dimensions of the cells, then the App Widget dimensions round *up* to the nearest cell size.

See the [App Widget Design Guidelines](#) for more information on sizing your App Widgets.

**Note:** To make your app widget portable across devices, your app widget's minimum size should never be larger than 4 x 4 cells.

- The `minResizeWidth` and `minResizeHeight` attributes specify the App Widget's absolute minimum size. These values should specify the size below which the App Widget would be illegible or otherwise unusable. Using these attributes allows the user to resize the widget to a size that may be smaller than the default widget size defined by the `minWidth` and `minHeight` attributes. Introduced in Android 3.1.

See the [App Widget Design Guidelines](#) for more information on sizing your App Widgets.

- The `updatePeriodMillis` attribute defines how often the App Widget framework should request an update from the [AppWidgetProvider](#) by calling the [onUpdate\(\)](#) callback method. The actual update is not guaranteed to occur exactly on time with this value and we suggest updating as infrequently as possible—perhaps no more than once an hour to conserve the battery. You might also allow the user to adjust the frequency in a configuration—some people might want a stock ticker to update every 15 minutes, or maybe only four times a day.

**Note:** If the device is asleep when it is time for an update (as defined by `updatePeriodMillis`), then the device will wake up in order to perform the update. If you don't update more than once per hour, this probably won't cause significant problems for the battery life. If, however, you need to update more frequently and/or you do not need to update while the device is asleep, then you can instead perform updates based on an alarm that will not wake the device. To do so, set an alarm with an Intent that your `AppWidgetProvider` receives, using the [AlarmManager](#). Set the alarm type to either [ELAPSED\\_REALTIME](#) or [RTC](#), which will only deliver the alarm when the device is awake. Then set `updatePeriodMillis` to zero ("0").

- The `initialLayout` attribute points to the layout resource that defines the App Widget layout.
- The `configure` attribute defines the [Activity](#) to launch when the user adds the App Widget, in order for him or her to configure App Widget properties. This is optional (read [Creating an App Widget Configuration Activity](#) below).
- The `previewImage` attribute specifies a preview of what the app widget will look like after it's configured, which the user sees when selecting the app widget. If not supplied, the user instead sees your application's launcher icon. This field corresponds to the `android:previewImage` attribute in the

<receiver> element in the `AndroidManifest.xml` file. For more discussion of using `preViewImage`, see [Setting a Preview Image](#). Introduced in Android 3.0.

- The `autoAdvanceViewId` attribute specifies the view ID of the app widget subview that should be auto-advanced by the widget's host. Introduced in Android 3.0.
- The `resizeMode` attribute specifies the rules by which a widget can be resized. You use this attribute to make homescreen widgets resizeable—horizontally, vertically, or on both axes. Users touch-hold a widget to show its resize handles, then drag the horizontal and/or vertical handles to change the size on the layout grid. Values for the `resizeMode` attribute include "horizontal", "vertical", and "none". To declare a widget as resizeable horizontally and vertically, supply the value "horizontal|vertical". Introduced in Android 3.1.
- The `minResizeHeight` attribute specifies the minimum height (in dps) to which the widget can be resized. This field has no effect if it is greater than `minHeight` or if vertical resizing isn't enabled (see `resizeMode`). Introduced in Android 4.0.
- The `minResizeWidth` attribute specifies the minimum width (in dps) to which the widget can be resized. This field has no effect if it is greater than `minWidth` or if horizontal resizing isn't enabled (see `resizeMode`). Introduced in Android 4.0.
- The `widgetCategory` attribute declares whether your App Widget can be displayed on the home screen, the lock screen (keyguard), or both. Values for this attribute include "home\_screen" and "keyguard". A widget that is displayed on both needs to ensure that it follows the design guidelines for both widget classes. For more information, see [Enabling App Widgets on the Lockscreen](#). The default value is "home\_screen". Introduced in Android 4.2.
- The `initialKeyguardLayout` attribute points to the layout resource that defines the lock screen App Widget layout. This works the same way as the `android:initialLayout`, in that it provides a layout that can appear immediately until your app widget is initialized and able to update the layout. Introduced in Android 4.2.

See the [AppWidgetProviderInfo](#) class for more information on the attributes accepted by the `<appwidget-provider>` element.

## Creating the App Widget Layout

You must define an initial layout for your App Widget in XML and save it in the project's `res/layout/` directory. You can design your App Widget using the View objects listed below, but before you begin designing your App Widget, please read and understand the [App Widget Design Guidelines](#).

Creating the App Widget layout is simple if you're familiar with [Layouts](#). However, you must be aware that App Widget layouts are based on [RemoteViews](#), which do not support every kind of layout or view widget.

A `RemoteViews` object (and, consequently, an App Widget) can support the following layout classes:

- [FrameLayout](#)
- [LinearLayout](#)
- [RelativeLayout](#)
- [GridLayout](#)

And the following widget classes:

- [AnalogClock](#)
- [Button](#)
- [Chronometer](#)
- [ImageButton](#)
- [ImageView](#)

- [ProgressBar](#)
- [TextView](#)
- [ViewFlipper](#)
- [ListView](#)
- [GridView](#)
- [StackView](#)
- [AdapterViewFlipper](#)

Descendants of these classes are not supported.

RemoteViews also supports [ViewStub](#), which is an invisible, zero-sized View you can use to lazily inflate layout resources at runtime.

## Adding margins to App Widgets

Widgets should not generally extend to screen edges and should not visually be flush with other widgets, so you should add margins on all sides around your widget frame.

As of Android 4.0, app widgets are automatically given padding between the widget frame and the app widget's bounding box to provide better alignment with other widgets and icons on the user's home screen. To take advantage of this strongly recommended behavior, set your application's [targetSdkVersion](#) to 14 or greater.

It's easy to write a single layout that has custom margins applied for earlier versions of the platform, and has no extra margins for Android 4.0 and greater:

1. Set your application's [targetSdkVersion](#) to 14 or greater.
2. Create a layout such as the one below, that references a [dimension resource](#) for its margins:

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/widget_margin">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal"
        android:background="@drawable/my_widget_background">
        ...
    </LinearLayout>

</FrameLayout>
```

3. Create two dimensions resources, one in `res/values/` to provide the pre-Android 4.0 custom margins, and one in `res/values-v14/` to provide no extra padding for Android 4.0 widgets:

**res/values/dimens.xml:**

```
<dimen name="widget_margin">8dp</dimen>
```

**res/values-v14/dimens.xml:**

```
<dimen name="widget_margin">0dp</dimen>
```

Another option is to simply build extra margins into your [nine-patch](#) background assets by default, and provide different nine-patches with no margins for API level 14 or later.

## Using the AppWidgetProvider Class

You must declare your AppWidgetProvider class implementation as a broadcast receiver using the `<receiver>` element in the `AndroidManifest` (see [Declaring an App Widget in the Manifest](#) above).

The [AppWidgetProvider](#) class extends `BroadcastReceiver` as a convenience class to handle the App Widget broadcasts. The AppWidgetProvider receives only the event broadcasts that are relevant to the App Widget, such as when the App Widget is updated, deleted, enabled, and disabled. When these broadcast events occur, the AppWidgetProvider receives the following method calls:

### [onUpdate\(\)](#)

This is called to update the App Widget at intervals defined by the `updatePeriodMillis` attribute in the `AppWidgetProviderInfo` (see [Adding the AppWidgetProviderInfo Metadata](#) above). This method is also called when the user adds the App Widget, so it should perform the essential setup, such as define event handlers for Views and start a temporary [Service](#), if necessary. However, if you have declared a configuration Activity, **this method is not called** when the user adds the App Widget, but is called for the subsequent updates. It is the responsibility of the configuration Activity to perform the first update when configuration is done. (See [Creating an App Widget Configuration Activity](#) below.)

### [onAppWidgetOptionsChanged\(\)](#)

This is called when the widget is first placed and any time the widget is resized. You can use this callback to show or hide content based on the widget's size ranges. You get the size ranges by calling [getAppWidgetOptions\(\)](#), which returns a [Bundle](#) that includes the following:

- [OPTION\\_APPWIDGET\\_MIN\\_WIDTH](#)—Contains the lower bound on the current width, in dp units, of a widget instance.
- [OPTION\\_APPWIDGET\\_MIN\\_HEIGHT](#)—Contains the lower bound on the current height, in dp units, of a widget instance.
- [OPTION\\_APPWIDGET\\_MAX\\_WIDTH](#)—Contains the upper bound on the current width, in dp units, of a widget instance.
- [OPTION\\_APPWIDGET\\_MAX\\_HEIGHT](#)—Contains the upper bound on the current width, in dp units, of a widget instance.

This callback was introduced in API Level 16 (Android 4.1). If you implement this callback, make sure that your app doesn't depend on it since it won't be called on older devices.

### [onDeleted\(Context, int\[\]\)](#)

This is called every time an App Widget is deleted from the App Widget host.

### [onEnabled\(Context\)](#)

This is called when an instance the App Widget is created for the first time. For example, if the user adds two instances of your App Widget, this is only called the first time. If you need to open a new database or perform other setup that only needs to occur once for all App Widget instances, then this is a good place to do it.

### [onDisabled\(Context\)](#)

This is called when the last instance of your App Widget is deleted from the App Widget host. This is where you should clean up any work done in [onEnabled\(Context\)](#), such as delete a temporary database.

### [onReceive \(Context, Intent\)](#)

This is called for every broadcast and before each of the above callback methods. You normally don't need to implement this method because the default AppWidgetProvider implementation filters all App Widget broadcasts and calls the above methods as appropriate.

The most important AppWidgetProvider callback is [onUpdate \(\)](#) because it is called when each App Widget is added to a host (unless you use a configuration Activity). If your App Widget accepts any user interaction events, then you need to register the event handlers in this callback. If your App Widget doesn't create temporary files or databases, or perform other work that requires clean-up, then [onUpdate \(\)](#) may be the only callback method you need to define. For example, if you want an App Widget with a button that launches an Activity when clicked, you could use the following implementation of AppWidgetProvider:

```
public class ExampleAppWidgetProvider extends AppWidgetProvider {  
  
    public void onUpdate(Context context, AppWidgetManager appWidgetManager, int  
        final int N = appWidgetIds.length;  
  
        // Perform this loop procedure for each App Widget that belongs to this  
        for (int i=0; i<N; i++) {  
            int appWidgetId = appWidgetIds[i];  
  
            // Create an Intent to launch ExampleActivity  
            Intent intent = new Intent(context, ExampleActivity.class);  
            PendingIntent pendingIntent = PendingIntent.getActivity(context, 0,  
  
            // Get the layout for the App Widget and attach an on-click listener  
            // to the button  
            RemoteViews views = new RemoteViews(context.getPackageName(), R.layout.  
            views.setOnClickPendingIntent(R.id.button, pendingIntent);  
  
            // Tell the AppWidgetManager to perform an update on the current app  
            appWidgetManager.updateAppWidget(appWidgetId, views);  
        }  
    }  
}
```

This AppWidgetProvider defines only the [onUpdate \(\)](#) method for the purpose of defining a [PendingIntent](#) that launches an [Activity](#) and attaching it to the App Widget's button with [setOnClickPendingIntent \(int, PendingIntent\)](#). Notice that it includes a loop that iterates through each entry in `appWidgetIds`, which is an array of IDs that identify each App Widget created by this provider. In this way, if the user creates more than one instance of the App Widget, then they are all updated simultaneously. However, only one `updatePeriodMillis` schedule will be managed for all instances of the App Widget. For example, if the update schedule is defined to be every two hours, and a second instance of the App Widget is added one hour after the first one, then they will both be updated on the period defined by the first one and the second update period will be ignored (they'll both be updated every two hours, not every hour).

**Note:** Because [AppWidgetProvider](#) is an extension of [BroadcastReceiver](#), your process is not guaranteed to keep running after the callback methods return (see [BroadcastReceiver](#) for information about the broadcast lifecycle). If your App Widget setup process can take several seconds (perhaps while performing web requests) and you require that your process continues, consider starting a [Service](#) in the [onUpdate \(\)](#) method. From within the Service, you can perform your own updates to the App Widget without worrying about the AppWidgetProvider closing down due to an [Application Not Responding](#) (ANR) error. See the [Wiktionary sample's AppWidgetProvider](#) for an example of an App Widget running a [Service](#).

Also see the [ExampleAppWidgetProvider.java](#) sample class.

## Receiving App Widget broadcast Intents

[AppWidgetProvider](#) is just a convenience class. If you would like to receive the App Widget broadcasts directly, you can implement your own [BroadcastReceiver](#) or override the [onReceive\(Context, Intent\)](#) callback. The Intents you need to care about are as follows:

- [ACTION\\_APPWIDGET\\_UPDATE](#)
- [ACTION\\_APPWIDGET\\_DELETED](#)
- [ACTION\\_APPWIDGET\\_ENABLED](#)
- [ACTION\\_APPWIDGET\\_DISABLED](#)
- [ACTION\\_APPWIDGET\\_OPTIONS\\_CHANGED](#)

## Creating an App Widget Configuration Activity

If you would like the user to configure settings when he or she adds a new App Widget, you can create an App Widget configuration Activity. This [Activity](#) will be automatically launched by the App Widget host and allows the user to configure available settings for the App Widget at create-time, such as the App Widget color, size, update period or other functionality settings.

The configuration Activity should be declared as a normal Activity in the Android manifest file. However, it will be launched by the App Widget host with the [ACTION\\_APPWIDGET\\_CONFIGURE](#) action, so the Activity needs to accept this Intent. For example:

```
<activity android:name=".ExampleAppWidgetConfigure">
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_CONFIGURE"/>
    </intent-filter>
</activity>
```

Also, the Activity must be declared in the AppWidgetProviderInfo XML file, with the `android:configure` attribute (see [Adding the AppWidgetProviderInfo Metadata](#) above). For example, the configuration Activity can be declared like this:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    android:configure="com.example.android.ExampleAppWidgetConfigure"
    ...
</appwidget-provider>
```

Notice that the Activity is declared with a fully-qualified namespace, because it will be referenced from outside your package scope.

That's all you need to get started with a configuration Activity. Now all you need is the actual Activity. There are, however, two important things to remember when you implement the Activity:

- The App Widget host calls the configuration Activity and the configuration Activity should always return a result. The result should include the App Widget ID passed by the Intent that launched the Activity (saved in the Intent extras as [EXTRA\\_APPWIDGET\\_ID](#)).
- The [onUpdate\(\)](#) method **will not be called** when the App Widget is created (the system will not send the ACTION\_APPWIDGET\_UPDATE broadcast when a configuration Activity is launched). It is the responsibility of the configuration Activity to request an update from the AppWidgetManager when

the App Widget is first created. However, [onUpdate \(\)](#) will be called for subsequent updates—it is only skipped the first time.

See the code snippets in the following section for an example of how to return a result from the configuration and update the App Widget.

## Updating the App Widget from the configuration Activity

When an App Widget uses a configuration Activity, it is the responsibility of the Activity to update the App Widget when configuration is complete. You can do so by requesting an update directly from the [AppWidgetManager](#).

Here's a summary of the procedure to properly update the App Widget and close the configuration Activity:

1. First, get the App Widget ID from the Intent that launched the Activity:

```
Intent intent = getIntent();
Bundle extras = intent.getExtras();
if (extras != null) {
    mAppWidgetId = extras.getInt(
        AppWidgetManager.EXTRA_APPWIDGET_ID,
        AppWidgetManager.INVALID_APPWIDGET_ID);
}
```

2. Perform your App Widget configuration.
3. When the configuration is complete, get an instance of the AppWidgetManager by calling [getInstance \(Context\)](#):

```
AppWidgetManager appWidgetManager = AppWidgetManager.getInstance(context)
```

4. Update the App Widget with a [RemoteViews](#) layout by calling [updateAppWidget \(int, RemoteViews\)](#):

```
RemoteViews views = new RemoteViews(context.getPackageName(),
R.layout.example_appwidget);
appWidgetManager.updateAppWidget(mAppWidgetId, views);
```

5. Finally, create the return Intent, set it with the Activity result, and finish the Activity:

```
Intent resultValue = new Intent();
resultValue.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, mAppWidgetId);
setResult(RESULT_OK, resultValue);
finish();
```

**Tip:** When your configuration Activity first opens, set the Activity result to RESULT\_CANCELED. This way, if the user backs-out of the Activity before reaching the end, the App Widget host is notified that the configuration was cancelled and the App Widget will not be added.

See the [ExampleAppWidgetConfigure.java](#) sample class in ApiDemos for an example.

# Setting a Preview Image

Android 3.0 introduces the [previewImage](#) field, which specifies a preview of what the app widget looks like. This preview is shown to the user from the widget picker. If this field is not supplied, the app widget's icon is used for the preview.

This is how you specify this setting in XML:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    android:previewImage="@drawable/preview"
</appwidget-provider>
```

To help create a preview image for your app widget (to specify in the [previewImage](#) field), the Android emulator includes an application called "Widget Preview." To create a preview image, launch this application, select the app widget for your application and set it up how you'd like your preview image to appear, then save it and place it in your application's drawable resources.

# Enabling App Widgets on the Lockscreen

Android 4.2 introduces the ability for users to add widgets to the lock screen. To indicate that your app widget is available for use on the lock screen, declare the [android:widgetCategory](#) attribute in the XML file that specifies your [AppWidgetProviderInfo](#). This attribute supports two values: "home\_screen" and "keyguard". An app widget can declare support for one or both.

By default, every app widget supports placement on the Home screen, so "home\_screen" is the default value for the [android:widgetCategory](#) attribute. If you want your app widget to be available for the lock screen, add the "keyguard" value:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    android:widgetCategory="keyguard|home_screen"
</appwidget-provider>
```

If you declare a widget to be displayable on both keyguard (lockscreen) and home, it's likely that you'll want to customize the widget depending on where it is displayed. For example, you might create a separate layout file for keyguard vs. home. The next step is to detect the widget category at runtime and respond accordingly. You can detect whether your widget is on the lockscreen or home screen by calling [getAppWidgetOptions\(\)](#) to get the widget's options as a [Bundle](#). The returned bundle will include the key [OPTION\\_APPWIDGET\\_HOST\\_CATEGORY](#), whose value will be one of [WIDGET\\_CATEGORY\\_HOME\\_SCREEN](#) or [WIDGET\\_CATEGORY\\_KEYGUARD](#). This value is determined by the host into which the widget is bound. In the [AppWidgetProvider](#), you can then check the widget's category, for example:

```
AppWidgetManager appWidgetManager;
int widgetId;
Bundle myOptions = appWidgetManager.getAppWidgetOptions(widgetId);

// Get the value of OPTION_APPWIDGET_HOST_CATEGORY
int category = myOptions.getInt(AppWidgetManager.OPTION_APPWIDGET_HOST_CATEGORY)

// If the value is WIDGET_CATEGORY_KEYGUARD, it's a lockscreen widget
boolean isKeyguard = category == AppWidgetProviderInfo.WIDGET_CATEGORY_KEYGUARD
```

Once you know the widget's category, you can optionally load a different base layout, set different properties, and so on. For example:

```
int baseLayout = isKeyguard ? R.layout.keyguard_widget_layout : R.layout.widget
```

You should also specify an initial layout for your app widget when on the lock screen with the [an-droid:initialKeyguardLayout](#) attribute. This works the same way as the [an-droid:initialLayout](#), in that it provides a layout that can appear immediately until your app widget is initialized and able to update the layout.

## Sizing guidelines

When a widget is hosted on the lockscreen, the framework ignores the minWidth, minHeight, minResizeWidth, and minResizeHeight fields. If a widget is also a home screen widget, these parameters are still needed as they're still used on home, but they will be ignored for purposes of the lockscreen.

The width of a lockscreen widget always fills the provided space. For the height of a lockscreen widget, you have the following options:

- If the widget does not mark itself as vertically resizable (`android:resizeMode="vertical"`), then the widget height will always be "small":
  - On a phone in portrait mode, "small" is defined as the space remaining when an unlock UI is being displayed.
  - On tablets and landscape phones, "small" is set on a per-device basis.
- If the widget marks itself as vertically resizable, then the widget height shows up as "small" on portrait phones displaying an unlock UI. In all other cases, the widget sizes to fill the available height.

## Using App Widgets with Collections

Android 3.0 introduces app widgets with collections. These kinds of App Widgets use the [RemoteViewsService](#) to display collections that are backed by remote data, such as from a [content provider](#). The data provided by the [RemoteViewsService](#) is presented in the app widget using one of the following view types, which we'll refer to as "collection views:"

### ListView

A view that shows items in a vertically scrolling list. For an example, see the Gmail app widget.

### GridView

A view that shows items in two-dimensional scrolling grid. For an example, see the Bookmarks app widget.

### StackView

A stacked card view (kind of like a rolodex), where the user can flick the front card up/down to see the previous/next card, respectively. Examples include the YouTube and Books app widgets.

### AdapterViewFlipper

An adapter-backed simple [ViewAnimator](#) that animates between two or more views. Only one child is shown at a time.

As stated above, these collection views display collections backed by remote data. This means that they use an [Adapter](#) to bind their user interface to their data. An [Adapter](#) binds individual items from a set of data into individual [View](#) objects. Because these collection views are backed by adapters, the Android framework must include extra architecture to support their use in app widgets. In the context of an app widget, the [Adapter](#) is

replaced by a [RemoteViewsFactory](#), which is simply a thin wrapper around the [Adapter](#) interface. When requested for a specific item in the collection, the [RemoteViewsFactory](#) creates and returns the item for the collection as a [RemoteViews](#) object. In order to include a collection view in your app widget, you must implement [RemoteViewsService](#) and [RemoteViewsFactory](#).

[RemoteViewsService](#) is a service that allows a remote adapter to request [RemoteViews](#) objects. [RemoteViewsFactory](#) is an interface for an adapter between a collection view (such as [ListView](#), [GridView](#), and so on) and the underlying data for that view. From the [StackView Widget sample](#), here is an example of the boilerplate code you use to implement this service and interface:

```
public class StackWidgetService extends RemoteViewsService {  
    @Override  
    public RemoteViewsFactory onGetViewFactory(Intent intent) {  
        return new StackRemoteViewsFactory(this.getApplicationContext(), intent  
    }  
  
    class StackRemoteViewsFactory implements RemoteViewsService.RemoteViewsFactory  
    //... include adapter-like methods here. See the StackView Widget sample.  
}
```

## Sample application

The code excerpts in this section are drawn from the [StackView Widget sample](#):



This sample consists of a stack of 10 views, which display the values "0 !" through "9 !". The sample app widget has these primary behaviors:

- The user can vertically fling the top view in the app widget to display the next or previous view. This is a built-in StackView behavior.
- Without any user interaction, the app widget automatically advances through its views in sequence, like a slide show. This is due to the setting `android:autoAdvanceViewId="@+id/stack_view"` in the `res/xml/stackwidgetinfo.xml` file. This setting applies to the view ID, which in this case is the view ID of the stack view.
- If the user touches the top view, the app widget displays the [Toast](#) message "Touched view *n*," where *n* is the index (position) of the touched view. For more discussion of how this is implemented, see [Adding behavior to individual items](#).

## Implementing app widgets with collections

To implement an app widget with collections, you follow the same basic steps you would use to implement any app widget. The following sections describe the additional steps you need to perform to implement an app widget with collections.

### Manifest for app widgets with collections

In addition to the requirements listed in [Declaring an app widget in the Manifest](#), to make it possible for app widgets with collections to bind to your [RemoteViewsService](#), you must declare the service in your manifest file with the permission [BIND\\_REMOTEVIEWS](#). This prevents other applications from freely accessing your app widget's data. For example, when creating an App Widget that uses [RemoteViewsService](#) to populate a collection view, the manifest entry may look like this:

```
<service android:name="MyWidgetService"
...
    android:permission="android.permission.BIND_REMOTEVIEWS" />
```

The line `android:name="MyWidgetService"` refers to your subclass of [RemoteViewsService](#).

### Layout for app widgets with collections

The main requirement for your app widget layout XML file is that it include one of the collection views: [ListView](#), [GridView](#), [StackView](#), or [AdapterViewFlipper](#). Here is the `widget_layout.xml` for the [StackView Widget sample](#):

```
<?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">
    <StackView xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/stack_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:loopViews="true" />
    <TextView xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/empty_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:background="@drawable/widget_item_background"
        android:textColor="#ffffffff"
        android:textStyle="bold"
        android:text="@string/empty_view_text"
        android:textSize="20sp" />
</FrameLayout>
```

Note that empty views must be siblings of the collection view for which the empty view represents empty state.

In addition to the layout file for your entire app widget, you must create another layout file that defines the layout for each item in the collection (for example, a layout for each book in a collection of books). For example,

the [StackView Widget sample](#) only has one layout file, `widget_item.xml`, since all items use the same layout. But the [WeatherListWidget sample](#) has two layout files: `dark_widget_item.xml` and `light_widget_item.xml`.

## AppWidgetProvider class for app widgets with collections

As with a regular app widget, the bulk of your code in your [AppWidgetProvider](#) subclass typically goes in [onUpdate\(\)](#). The major difference in your implementation for [onUpdate\(\)](#) when creating an app widget with collections is that you must call [setRemoteAdapter\(\)](#). This tells the collection view where to get its data. The [RemoteViewsService](#) can then return your implementation of [RemoteViewsFactory](#), and the widget can serve up the appropriate data. When you call this method, you must pass an intent that points to your implementation of [RemoteViewsService](#) and the app widget ID that specifies the app widget to update.

For example, here's how the StackView Widget sample implements the [onUpdate\(\)](#) callback method to set the [RemoteViewsService](#) as the remote adapter for the app widget collection:

```
public void onUpdate(Context context, AppWidgetManager appWidgetManager,
int[] appWidgetIds) {
    // update each of the app widgets with the remote adapter
    for (int i = 0; i < appWidgetIds.length; ++i) {

        // Set up the intent that starts the StackWidgetService, which will
        // provide the views for this collection.
        Intent intent = new Intent(context, StackWidgetService.class);
        // Add the app widget ID to the intent extras.
        intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetIds[i]);
        intent.setData(Uri.parse(intent.toUri(Intent.URI_INTENT_SCHEME)));
        // Instantiate the RemoteViews object for the app widget layout.
        RemoteViews rv = new RemoteViews(context.getPackageName(), R.layout.widget_stack_view);
        // Set up the RemoteViews object to use a RemoteViews adapter.
        // This adapter connects
        // to a RemoteViewsService through the specified intent.
        // This is how you populate the data.
        rv.setRemoteAdapter(appWidgetIds[i], R.id.stack_view, intent);

        // The empty view is displayed when the collection has no items.
        // It should be in the same layout used to instantiate the RemoteViews
        // object above.
        rv.setEmptyView(R.id.stack_view, R.id.empty_view);

        //
        // Do additional processing specific to this app widget...
        //

        appWidgetManager.updateAppWidget(appWidgetIds[i], rv);
    }
    super.onUpdate(context, appWidgetManager, appWidgetIds);
}
```

## RemoteViewsService class

### Persisting data

You can't rely on a single instance of your service, or any data it contains, to persist. You should therefore not store any data in your [RemoteViewsService](#) (unless it is static). If you want your app widget's data to persist, the best approach is to use a [ContentProvider](#) whose data persists beyond the process lifecycle.

As described above, your [RemoteViewsService](#) subclass provides the [RemoteViewsFactory](#) used to populate the remote collection view.

Specifically, you need to perform these steps:

1. Subclass [RemoteViewsService](#). [RemoteViewsService](#) is the service through which a remote adapter can request [RemoteViews](#).
2. In your [RemoteViewsService](#) subclass, include a class that implements the [RemoteViewsFactory](#) interface. [RemoteViewsFactory](#) is an interface for an adapter between a remote collection view (such as [ListView](#), [GridView](#), and so on) and the underlying data for that view. Your implementation is responsible for making a [RemoteViews](#) object for each item in the data set. This interface is a thin wrapper around [Adapter](#).

The primary contents of the [RemoteViewsService](#) implementation is its [RemoteViewsFactory](#), described below.

### RemoteViewsFactory interface

Your custom class that implements the [RemoteViewsFactory](#) interface provides the app widget with the data for the items in its collection. To do this, it combines your app widget item XML layout file with a source of data. This source of data could be anything from a database to a simple array. In the [StackView Widget sample](#), the data source is an array of [WidgetItem](#)s. The [RemoteViewsFactory](#) functions as an adapter to glue the data to the remote collection view.

The two most important methods you need to implement for your [RemoteViewsFactory](#) subclass are [onCreate\(\)](#) and [getViewAt\(\)](#).

The system calls [onCreate\(\)](#) when creating your factory for the first time. This is where you set up any connections and/or cursors to your data source. For example, the [StackView Widget sample](#) uses [onCreate\(\)](#) to initialize an array of [WidgetItem](#) objects. When your app widget is active, the system accesses these objects using their index position in the array and the text they contain is displayed

Here is an excerpt from the [StackView Widget](#) sample's [RemoteViewsFactory](#) implementation that shows portions of the [onCreate\(\)](#) method:

```
class StackRemoteViewsFactory implements  
RemoteViewsService.RemoteViewsFactory {  
    private static final int mCount = 10;  
    private List<WidgetItem> mWidgetItemList = new ArrayList<WidgetItem>();  
    private Context mContext;  
    private int mAppWidgetId;  
  
    public StackRemoteViewsFactory(Context context, Intent intent) {  
        mContext = context;  
        mAppWidgetId = intent.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
```

```

        AppWidgetManager.INVALID_APPWIDGET_ID);
    }

    public void onCreate() {
        // In onCreate() you setup any connections / cursors to your data source
        // for example downloading or creating content etc, should be deferred
        // or getViewAt(). Taking more than 20 seconds in this call will result
        for (int i = 0; i < mCount; i++) {
            mWidgetItem.add(new WidgetItem(i + "!="));
        }
        ...
    }
...
}

```

The [RemoteViewsFactory](#) method [getViewAt\(\)](#) returns a [RemoteViews](#) object corresponding to the data at the specified position in the data set. Here is an excerpt from the [StackView Widget](#) sample's [RemoteViewsFactory](#) implementation:

```

public RemoteViews getViewAt(int position) {

    // Construct a remote views item based on the app widget item XML file,
    // and set the text based on the position.
    RemoteViews rv = new RemoteViews(mContext.getPackageName(), R.layout.widget);
    rv.setTextViewText(R.id.widget_item, mWidgetItem.get(position).text);

    ...
    // Return the remote views object.
    return rv;
}

```

## Adding behavior to individual items

The above sections show you how to bind your data to your app widget collection. But what if you want to add dynamic behavior to the individual items in your collection view?

As described in [Using the AppWidgetProvider Class](#), you normally use [setOnClickPendingIntent\(\)](#) to set an object's click behavior—such as to cause a button to launch an [Activity](#). But this approach is not allowed for child views in an individual collection item (to clarify, you could use [setOnClickPendingIntent\(\)](#) to set up a global button in the Gmail app widget that launches the app, for example, but not on the individual list items). Instead, to add click behavior to individual items in a collection, you use [setOnClickFillInIntent\(\)](#). This entails setting up a pending intent template for your collection view, and then setting a fill-in intent on each item in the collection via your [RemoteViewsFactory](#).

This section uses the [StackView Widget sample](#) to describe how to add behavior to individual items. In the [StackView Widget sample](#), if the user touches the top view, the app widget displays the [Toast](#) message "Touched view *n*," where *n* is the index (position) of the touched view. This is how it works:

- The StackWidgetProvider (an [AppWidgetProvider](#) subclass) creates a pending intent that has a custom action called `TOAST_ACTION`.
- When the user touches a view, the intent is fired and it broadcasts `TOAST_ACTION`.
- This broadcast is intercepted by the StackWidgetProvider's [onReceive\(\)](#) method, and the app widget displays the [Toast](#) message for the touched view. The data for the collection items is provided by the [RemoteViewsFactory](#), via the [RemoteViewsService](#).

**Note:** The [StackView Widget sample](#) uses a broadcast, but typically an app widget would simply launch an activity in a scenario like this one.

### Setting up the pending intent template

The `StackWidgetProvider` ([AppWidgetProvider](#) subclass) sets up a pending intent. Individuals items of a collection cannot set up their own pending intents. Instead, the collection as a whole sets up a pending intent template, and the individual items set a fill-in intent to create unique behavior on an item-by-item basis.

This class also receives the broadcast that is sent when the user touches a view. It processes this event in its [onReceive\(\)](#) method. If the intent's action is `TOAST_ACTION`, the app widget displays a [Toast](#) message for the current view.

```
public class StackWidgetProvider extends AppWidgetProvider {  
    public static final String TOAST_ACTION = "com.example.android.stackwidget.  
    public static final String EXTRA_ITEM = "com.example.android.stackwidget.EX  
  
    ...  
  
    // Called when the BroadcastReceiver receives an Intent broadcast.  
    // Checks to see whether the intent's action is TOAST_ACTION. If it is, the  
    // displays a Toast message for the current item.  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        AppWidgetManager mgr = AppWidgetManager.getInstance(context);  
        if (intent.getAction().equals(TOAST_ACTION)) {  
            int appWidgetId = intent.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,  
                AppWidgetManager.INVALID_APPWIDGET_ID);  
            int viewIndex = intent.getIntExtra(EXTRA_ITEM, 0);  
            Toast.makeText(context, "Touched view " + viewIndex, Toast.LENGTH_S  
        }  
        super.onReceive(context, intent);  
    }  
  
    @Override  
    public void onUpdate(Context context, AppWidgetManager appWidgetManager, int  
        // update each of the app widgets with the remote adapter  
        for (int i = 0; i < appWidgetIds.length; ++i) {  
  
            // Sets up the intent that points to the StackViewService that will  
            // provide the views for this collection.  
            Intent intent = new Intent(context, StackWidgetService.class);  
            intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetIds[i]);  
            // When intents are compared, the extras are ignored, so we need to  
            // into the data so that the extras will not be ignored.  
            intent.setData(Uri.parse(intent.toUri(Intent.URI_INTENT_SCHEME)));  
            RemoteViews rv = new RemoteViews(context.getPackageName(), R.layout.  
            rv.setRemoteAdapter(appWidgetIds[i], R.id.stack_view, intent);  
  
            // The empty view is displayed when the collection has no items. It  
            // of the collection view.  
            rv.setEmptyView(R.id.stack_view, R.id.empty_view);  
        }  
    }  
}
```

```

    // This section makes it possible for items to have individualized
    // It does this by setting up a pending intent template. Individual
    // cannot set up their own pending intents. Instead, the collection
    // up a pending intent template, and the individual items set a file
    // to create unique behavior on an item-by-item basis.
    Intent toastIntent = new Intent(context, StackWidgetProvider.class)
    // Set the action for the intent.
    // When the user touches a particular view, it will have the effect
    // broadcasting TOAST_ACTION.
    toastIntent.setAction(StackWidgetProvider.TOAST_ACTION);
    toastIntent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
    intent.setData(Uri.parse(intent.toUri(Intent.URI_INTENT_SCHEME)));
    PendingIntent toastPendingIntent = PendingIntent.getBroadcast(context,
        PendingIntent.FLAG_UPDATE_CURRENT);
    rv.setPendingIntentTemplate(R.id.stack_view, toastPendingIntent);

    appWidgetManager.updateAppWidget(appWidgetIds[i], rv);
}
super.onUpdate(context, appWidgetManager, appWidgetIds);
}
}

```

### Setting the fill-in Intent

Your [RemoteViewsFactory](#) must set a fill-in intent on each item in the collection. This makes it possible to distinguish the individual on-click action of a given item. The fill-in intent is then combined with the [PendingIntent](#) template in order to determine the final intent that will be executed when the item is clicked.

```

public class StackWidgetService extends RemoteViewsService {
    @Override
    public RemoteViewsFactory onGetViewFactory(Intent intent) {
        return new StackRemoteViewsFactory(this.getApplicationContext(), intent);
    }
}

class StackRemoteViewsFactory implements RemoteViewsService.RemoteViewsFactory {
    private static final int mCount = 10;
    private List<WidgetItem> mWidgetItemList = new ArrayList<WidgetItem>();
    private Context mContext;
    private int mAppWidgetId;

    public StackRemoteViewsFactory(Context context, Intent intent) {
        mContext = context;
        mAppWidgetId = intent.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
            AppWidgetManager.INVALID_APPWIDGET_ID);
    }

    // Initialize the data set.
    public void onCreate() {
        // In onCreate() you set up any connections / cursors to your data
        // for example downloading or creating content etc, should be defer
        // or getViewAt(). Taking more than 20 seconds in this call will re
        for (int i = 0; i < mCount; i++) {

```

```

        mWidgetItems.add(new WidgetItem(i + "!"));

    }

    ...

}

 ...



// Given the position (index) of a WidgetItem in the array, use the item
// combination with the app widget item XML file to construct a RemoteViews
public RemoteViews getViewAt(int position) {
    // position will always range from 0 to getCount() - 1.

    // Construct a RemoteViews item based on the app widget item XML file
    // text based on the position.
    RemoteViews rv = new RemoteViews(mContext.getPackageName(), R.layout.widget_item);
    rv.setTextViewText(R.id.widget_item, mWidgetItems.get(position).text);

    // Next, set a fill-intent, which will be used to fill in the pending
    // intent that is set on the collection view in StackWidgetProvider.
    Bundle extras = new Bundle();
    extras.putInt(StackWidgetProvider.EXTRA_ITEM, position);
    Intent fillInIntent = new Intent();
    fillInIntent.putExtras(extras);
    // Make it possible to distinguish the individual on-click
    // action of a given item
    rv.setOnClickFillInIntent(R.id.widget_item, fillInIntent);

    ...

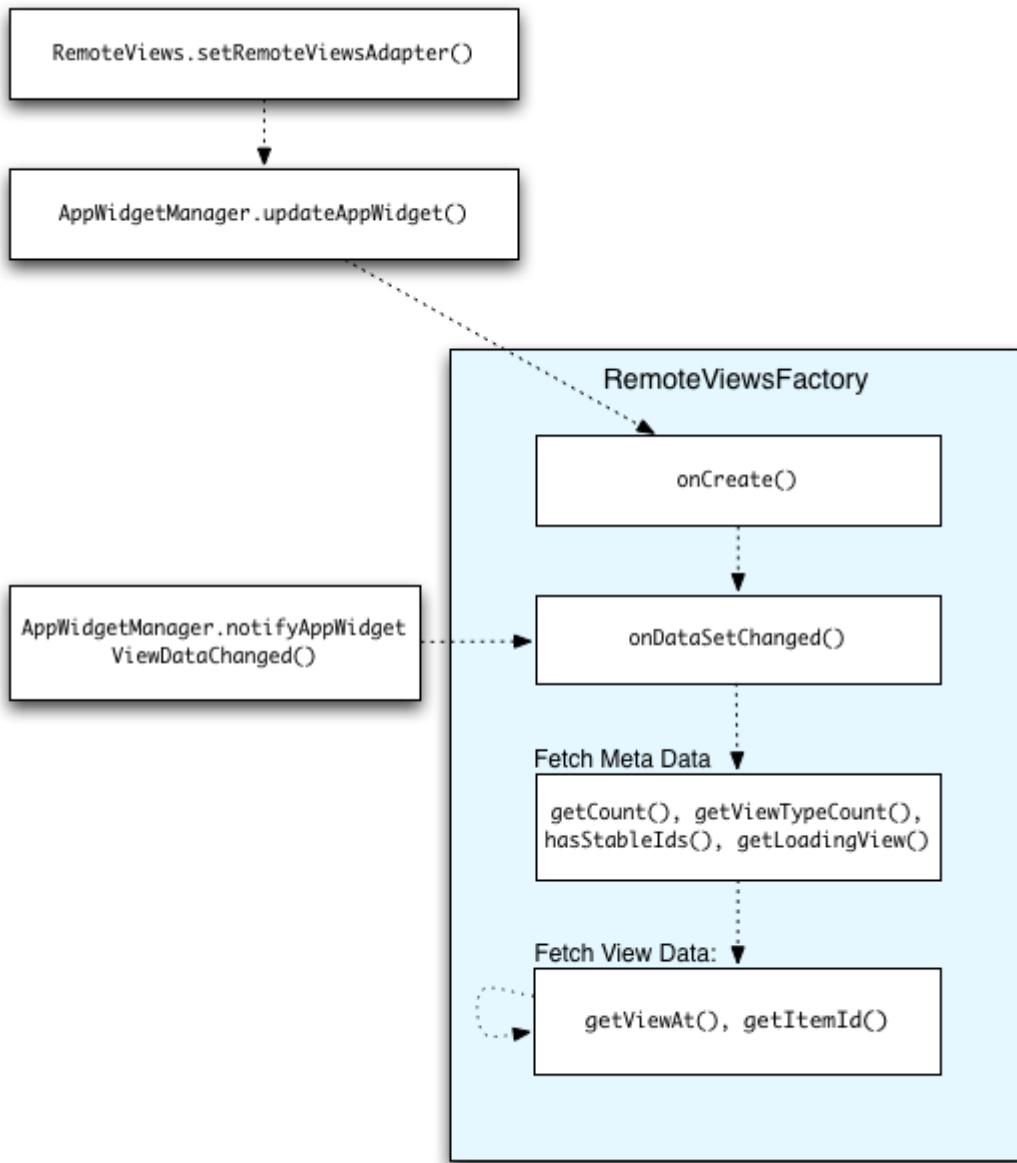
    // Return the RemoteViews object.
    return rv;
}

...
}

```

## Keeping Collection Data Fresh

The following figure illustrates the flow that occurs in an app widget that uses collections when updates occur. It shows how the app widget code interacts with the [RemoteViewsFactory](#), and how you can trigger updates:



One feature of app widgets that use collections is the ability to provide users with up-to-date content. For example, consider the Android 3.0 Gmail app widget, which provides users with a snapshot of their inbox. To make this possible, you need to be able to trigger your [RemoteViewsFactory](#) and collection view to fetch and display new data. You achieve this with the [AppWidgetManager](#) call [notifyAppWidgetViewDataChanged\(\)](#). This call results in a callback to your `RemoteViewsFactory`'s [onDataSetChanged\(\)](#) method, which gives you the opportunity to fetch any new data. Note that you can perform processing-intensive operations synchronously within the [onDataSetChanged\(\)](#) callback. You are guaranteed that this call will be completed before the metadata or view data is fetched from the [RemoteViewsFactory](#). In addition, you can perform processing-intensive operations within the [getViewAt\(\)](#) method. If this call takes a long time, the loading view (specified by the `RemoteViewsFactory`'s [getLoadingView\(\)](#) method) will be displayed in the corresponding position of the collection view until it returns.

# App Widget Host

## In this document

1. [Binding App Widgets](#)
  1. [Binding app widgets on Android 4.0 and lower](#)
  2. [Binding app widgets on Android 4.1 and higher](#)
2. [Host Responsibilities](#)
  1. [Android 3.0](#)
  2. [Android 3.1](#)
  3. [Android 4.0](#)
  4. [Android 4.1](#)
  5. [Android 4.2](#)

The Android Home screen available on most Android devices allows the user to embed [app widgets](#) for quick access to content. If you're building a Home replacement or a similar app, you can also allow the user to embed app widgets by implementing an [AppWidgetHost](#). This is not something that most apps will ever need to do, but if you are creating your own host, it's important to understand the contractual obligations a host implicitly agrees to.

This document focuses on the responsibilities involved in implementing a custom [AppWidgetHost](#). For an example of how to implement an [AppWidgetHost](#), see the source code for the Android Home screen [Launcher](#).

Here is an overview of key classes and concepts involved in implementing a custom [AppWidgetHost](#):

- **App Widget Host**— The [AppWidgetHost](#) provides the interaction with the AppWidget service for apps, like the home screen, that want to embed app widgets in their UI. An [AppWidgetHost](#) must have an ID that is unique within the host's own package. This ID remains persistent across all uses of the host. The ID is typically a hard-coded value that you assign in your application.
- **App Widget ID**— Each app widget instance is assigned a unique ID at the time of binding (see [bindAppWidgetIdIfAllowed\(\)](#), discussed in more detail in [Binding app widgets](#)). The unique ID is obtained by the host using [allocateAppWidgetId\(\)](#). This ID is persistent across the lifetime of the widget, that is, until it is deleted from the host. Any host-specific state (such as the size and location of the widget) should be persisted by the hosting package and associated with the app widget ID.
- **App Widget Host View**— [AppWidgetHostView](#) can be thought of as a frame that the widget is wrapped in whenever it needs to be displayed. An app widget is assigned to an [AppWidgetHostView](#) every time the widget is inflated by the host.
- **Options Bundle**— The [AppWidgetHost](#) uses the options bundle to communicate information to the [AppWidgetProvider](#) about how the widget is being displayed (for example, size range, and whether the widget is on a lockscreen or the home screen). This information allows the [AppWidgetProvider](#) to tailor the widget's contents and appearance based on how and where it is displayed. You use [updateAppWidgetOptions\(\)](#) and [updateAppWidgetSize\(\)](#) to modify an app widget's bundle. Both of these methods trigger a callback to the [AppWidgetProvider](#).

## Binding App Widgets

When a user adds an app widget to a host, a process called *binding* occurs. *Binding* refers to associating a particular app widget ID to a specific host and to a specific [AppWidgetProvider](#). There are different ways of achieving this, depending on what version of Android your app is running on.

## Binding app widgets on Android 4.0 and lower

On devices running Android version 4.0 and lower, users add app widgets via a system activity that allows users to select a widget. This implicitly does a permission check—that is, by adding the app widget, the user is implicitly granting permission to your app to add app widgets to the host. Here is an example that illustrates this approach, taken from the original [Launcher](#). In this snippet, an event handler invokes [startActivityForResult\(\)](#) with the request code `REQUEST_PICK_APPWIDGET` in response to a user action:

```
private static final int REQUEST_CREATE_APPWIDGET = 5;
private static final int REQUEST_PICK_APPWIDGET = 9;
...
public void onClick(DialogInterface dialog, int which) {
    switch (which) {
        ...
        case AddAdapter.ITEM_APPWIDGET: {
            ...
            int appWidgetId =
                Launcher.this.mAppWidgetHost.allocateAppWidgetId();
            Intent pickIntent =
                new Intent(AppWidgetManager.ACTION_APPWIDGET_PICK);
            pickIntent.putExtra
                (AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
            ...
            startActivityForResult(pickIntent, REQUEST_PICK_APPWIDGET);
            break;
        }
        ...
    }
}
```

When the system activity finishes, it returns a result with the user's chosen app widget to your activity. In the following example, the activity responds by calling `addAppWidget()` to add the app widget:

```
public final class Launcher extends Activity
    implements View.OnClickListener, OnLongClickListener {
    ...
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        mWaitingForResult = false;

        if (resultCode == RESULT_OK && mAddItemCellInfo != null) {
            switch (requestCode) {
                ...
                case REQUEST_PICK_APPWIDGET:
                    addAppWidget(data);
                    break;
                case REQUEST_CREATE_APPWIDGET:
                    completeAddAppWidget(data, mAddItemCellInfo, !mDesktopLocked);
                    break;
            }
        }
        ...
    }
}
```

The method `addAppWidget()` checks to see if the app widget needs to be configured before it's added:

```
void addAppWidget(Intent data) {
    int appWidgetId = data.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, -1)

    String customWidget = data.getStringExtra(EXTRA_CUSTOM_WIDGET);
    AppWidgetProviderInfo appWidget =
        mAppWidgetManager.getAppWidgetInfo(appWidgetId);

    if (appWidget.configure != null) {
        // Launch over to configure widget, if needed.
        Intent intent = new Intent(AppWidgetManager.ACTION_APPWIDGET_CONFIGURE)
        intent.setComponent(appWidget.configure);
        intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
        startActivityForResult(intent, REQUEST_CREATE_APPWIDGET);
    } else {
        // Otherwise, finish adding the widget.
    }
}
```

For more discussion of configuration, see [Creating an App Widget Configuration Activity](#).

Once the app widget is ready, the next step is to do the actual work of adding it to the workspace. The [original Launcher](#) uses a method called `completeAddAppWidget()` to do this.

## Binding app widgets on Android 4.1 and higher

Android 4.1 adds APIs for a more streamlined binding process. These APIs also make it possible for a host to provide a custom UI for binding. To use this improved process, your app must declare the [BIND\\_APPWIDGET](#) permission in its manifest:

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

But this is just the first step. At runtime the user must explicitly grant permission to your app to allow it to add app widgets to the host. To test whether your app has permission to add the widget, you use the [bindAppWidgetIdIfAllowed\(\)](#) method. If [bindAppWidgetIdIfAllowed\(\)](#) returns `false`, your app must display a dialog prompting the user to grant permission ("allow" or "always allow," to cover all future app widget additions). This snippet gives an example of how to display the dialog:

```
Intent intent = new Intent(AppWidgetManager.ACTION_APPWIDGET_BIND);
intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_PROVIDER, info.componentName);
// This is the options bundle discussed above
intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_OPTIONS, options);
startActivityForResult(intent, REQUEST_BIND_APPWIDGET);
```

The host also has to check whether the user added an app widget that needs configuration. For more discussion of this topic, see [Creating an App Widget Configuration Activity](#).

# Host Responsibilities

## What Version are You Targeting?

The approach you use in implementing your host should depend on what Android version you're targeting. Many of the features described in this section were introduced in 3.0 or later. For example:

- Android 3.0 (API Level 11) introduces auto-advance behavior for widgets.
- Android 3.1 (API Level 12) introduces the ability to resize widgets.
- Android 4.0 (API Level 15) introduces a change in padding policy that puts the responsibility on the host to manage padding.
- Android 4.1 (API Level 16) adds an API that allows the widget provider to get more detailed information about the environment in which its widget instances are being hosted.
- Android 4.2 (API Level 17) introduces the options bundle and the [bindAppWidgetIdIfAllowed\(\)](#) method. It also introduces lockscreen widgets.

If you are targeting earlier devices, refer to the original [Launcher](#) as an example.

Widget developers can specify a number of configuration settings for widgets using the [AppWidgetProviderInfo metadata](#). These configuration options, discussed in more detail below, can be retrieved by the host from the [AppWidgetProviderInfo](#) object associated with a widget provider.

Regardless of the version of Android you are targeting, all hosts have the following responsibilities:

- When adding a widget, you must allocate the widget ID as described above. You must also make sure that when a widget is removed from the host, you call [deleteAppWidgetId\(\)](#) to deallocate the widget ID.
- When adding a widget, be sure to launch its configuration activity if it exists, as described in [Updating the App Widget from the Configuration Activity](#). This is a necessary step for many app widgets before they can be properly displayed.
- Every app widget specifies a minimum width and height in dps, as defined in the [AppWidgetProviderInfo](#) metadata (using `android:minWidth` and `android:minHeight`). Make sure that the widget is laid out with at least this many dps. For example, many hosts align icons and widgets in a grid. In this scenario, by default the host should add the app widget using the minimum number of cells that satisfy the `minWidth` and `minHeight` constraints.

In addition to the requirements listed above, specific platform versions introduce features that place new responsibilities on the host. These are described in the following sections.

### Android 3.0

Android 3.0 (API Level 11) introduces the ability for a widget to specify [autoAdvanceViewId\(\)](#). This view ID should point to an instance of an [Advanceable](#), such as [StackView](#) or [AdapterViewFlipper](#). This indicates that the host should call [advance\(\)](#) on this view at an interval deemed appropriate by the host (taking into account whether it makes sense to advance the widget—for example, the host probably wouldn't want to advance a widget if it were on another page, or if the screen were turned off).

### Android 3.1

Android 3.1 (API Level 12) introduces the ability to resize widgets. A widget can specify that it is resizable using the `android:resizeMode` attribute in the [AppWidgetProviderInfo](#) metadata, and indicate

whether it supports horizontal and/or vertical resizing. Introduced in Android 4.0 (API Level 14), the widget can also specify a [android:minResizeWidth](#) and/or [android:minResizeHeight](#).

It is the host's responsibility to make it possible for the widget to be resized horizontally and/or vertically, as specified by the widget. A widget that specifies that it is resizable can be resized arbitrarily large, but should not be resized smaller than the values specified by [android:minResizeWidth](#) and [android:minResizeHeight](#). For a sample implementation, see [AppWidgetResizeFrame](#) in Launcher2.

## Android 4.0

Android 4.0 (API Level 15) introduces a change in padding policy that puts the responsibility on the host to manage padding. As of 4.0, app widgets no longer include their own padding. Instead, the system adds padding for each widget, based the characteristics of the current screen. This leads to a more uniform, consistent presentation of widgets in a grid. To assist applications that host app widgets, the platform provides the method [getDefalutPaddingForWidget\(\)](#). Applications can call this method to get the system-defined padding and account for it when computing the number of cells to allocate to the widget.

## Android 4.1

Android 4.1 (API Level 16) adds an API that allows the widget provider to get more detailed information about the environment in which its widget instances are being hosted. Specifically, the host hints to the widget provider about the size at which the widget is being displayed. It is the host's responsibility to provide this size information.

The host provides this information via [updateAppWidgetSize\(\)](#). The size is specified as a minimum and maximum width/height in dps. The reason that a range is specified (as opposed to a fixed size) is because the width and height of a widget may change with orientation. You don't want the host to have to update all of its widgets on rotation, as this could cause serious system slowdown. These values should be updated once upon the widget being placed, any time the widget is resized, and any time the launcher inflates the widget for the first time in a given boot (as the values aren't persisted across boot).

## Android 4.2

Android 4.2 (API Level 17) adds the ability for the options bundle to be specified at bind time. This is the ideal way to specify app widget options, including size, as it gives the [AppWidgetProvider](#) immediate access to the options data on the first update. This can be achieved by using the method [bindAppWidgetIdIfAllowed\(\)](#). For more discussion of this topic, see [Binding app widgets](#).

Android 4.2 also introduces lockscreen widgets. When hosting widgets on the lockscreen, the host must specify this information within the app widget options bundle (the [AppWidgetProvider](#) can use this information to style the widget appropriately). To designate a widget as a lockscreen widget, use [updateAppWidgetOptions\(\)](#) and include the field [OPTION\\_APPWIDGET\\_HOST\\_CATEGORY](#) with the value [WIDGET\\_CATEGORY\\_KEYGUARD](#). This option defaults to [WIDGET\\_CATEGORY\\_HOME\\_SCREEN](#), so it is not explicitly required to set this for a home screen host.

Make sure that your host adds only app widgets that are appropriate for your app—for example, if your host is a home screen, ensure that the [android:widgetCategory](#) attribute in the [AppWidgetProviderInfo](#) metadata includes the flag [WIDGET\\_CATEGORY\\_HOME\\_SCREEN](#). Similarly, for the lockscreen, ensure that field includes the flag [WIDGET\\_CATEGORY\\_KEYGUARD](#). For more discussion of this topic, see [Enabling App Widgets on the Lockscreen](#).

# The AndroidManifest.xml File

## In this document

1. [Structure of the Manifest File](#)
2. [File Conventions](#)
3. [File Features](#)
  1. [Intent Filters](#)
  2. [Icons and Labels](#)
  3. [Permissions](#)
  4. [Libraries](#)

Every application must have an `AndroidManifest.xml` file (with precisely that name) in its root directory. The manifest presents essential information about the application to the Android system, information the system must have before it can run any of the application's code. Among other things, the manifest does the following:

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It describes the components of the application — the activities, services, broadcast receivers, and content providers that the application is composed of. It names the classes that implement each of the components and publishes their capabilities (for example, which [Intent](#) messages they can handle). These declarations let the Android system know what the components are and under what conditions they can be launched.
- It determines which processes will host application components.
- It declares which permissions the application must have in order to access protected parts of the API and interact with other applications.
- It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the [Instrumentation](#) classes that provide profiling and other information as the application is running. These declarations are present in the manifest only while the application is being developed and tested; they're removed before the application is published.
- It declares the minimum level of the Android API that the application requires.
- It lists the libraries that the application must be linked against.

## Structure of the Manifest File

The diagram below shows the general structure of the manifest file and every element that it can contain. Each element, along with all of its attributes, is documented in full in a separate file. To view detailed information about any element, click on the element name in the diagram, in the alphabetical list of elements that follows the diagram, or on any other mention of the element name.

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

<manifest>

    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
```

```

<uses-feature />
<supports-screens />
<compatible-screens />
<supports-gl-texture />

<application>

    <activity>
        <intent-filter>
            <action />
            <category />
            <data />
        </intent-filter>
        <meta-data />
    </activity>

    <activity-alias>
        <intent-filter> . . .
        <meta-data />
    </activity-alias>

    <service>
        <intent-filter> . . .
        <meta-data/>
    </service>

    <receiver>
        <intent-filter> . . .
        <meta-data />
    </receiver>

    <provider>
        <grant-uri-permission />
        <meta-data />
        <path-permission />
    </provider>

    <uses-library />

</application>

</manifest>

```

All the elements that can appear in the manifest file are listed below in alphabetical order. These are the only legal elements; you cannot add your own elements or attributes.

```

<action>
<activity>
<activity-alias>
<application>
<category>
<data>
<grant-uri-permission>
<instrumentation>

```

```
<intent-filter>
<manifest>
<meta-data>
<permission>
<permission-group>
<permission-tree>
<provider>
<receiver>
<service>
<supports-screens>
<uses-configuration>
<uses-feature>
<uses-library>
<uses-permission>
<uses-sdk>
```

## File Conventions

Some conventions and rules apply generally to all elements and attributes in the manifest:

### Elements

Only the [<manifest>](#) and [<application>](#) elements are required, they each must be present and can occur only once. Most of the others can occur many times or not at all — although at least some of them must be present for the manifest to accomplish anything meaningful.

If an element contains anything at all, it contains other elements. All values are set through attributes, not as character data within an element.

Elements at the same level are generally not ordered. For example, [<activity>](#), [<provider>](#), and [<service>](#) elements can be intermixed in any sequence. (An [<activity-alias>](#) element is the exception to this rule: It must follow the [<activity>](#) it is an alias for.)

### Attributes

In a formal sense, all attributes are optional. However, there are some that must be specified for an element to accomplish its purpose. Use the documentation as a guide. For truly optional attributes, it mentions a default value or states what happens in the absence of a specification.

Except for some attributes of the root [<manifest>](#) element, all attribute names begin with an android: prefix — for example, android:alwaysRetainTaskState. Because the prefix is universal, the documentation generally omits it when referring to attributes by name.

### Declaring class names

Many elements correspond to Java objects, including elements for the application itself (the [<application>](#) element) and its principal components — activities ([<activity>](#)), services ([<service>](#)), broadcast receivers ([<receiver>](#)), and content providers ([<provider>](#)).

If you define a subclass, as you almost always would for the component classes ([Activity](#), [Service](#), [BroadcastReceiver](#), and [ContentProvider](#)), the subclass is declared through a name attribute. The name must include the full package designation. For example, an [Service](#) subclass might be declared as follows:

```
<manifest . . . >
    <application . . . >
        <service android:name="com.example.project.SecretService" . . . >
            . . .
        </service>
        . . .
    </application>
</manifest>
```

However, as a shorthand, if the first character of the string is a period, the string is appended to the application's package name (as specified by the [`<manifest>`](#) element's [`package`](#) attribute). The following assignment is the same as the one above:

```
<manifest package="com.example.project" . . . >
    <application . . . >
        <service android:name=".SecretService" . . . >
            . . .
        </service>
        . . .
    </application>
</manifest>
```

When starting a component, Android creates an instance of the named subclass. If a subclass isn't specified, it creates an instance of the base class.

## Multiple values

If more than one value can be specified, the element is almost always repeated, rather than listing multiple values within a single element. For example, an intent filter can list several actions:

```
<intent-filter . . . >
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.INSERT" />
    <action android:name="android.intent.action.DELETE" />
    . . .
</intent-filter>
```

## Resource values

Some attributes have values that can be displayed to users — for example, a label and an icon for an activity. The values of these attributes should be localized and therefore set from a resource or theme. Resource values are expressed in the following format,

```
@ [package:] type:name
```

where the *package* name can be omitted if the resource is in the same package as the application, *type* is a type of resource — such as "string" or "drawable" — and *name* is the name that identifies the specific resource. For example:

```
<activity android:icon="@drawable/smallPic" . . . >
```

Values from a theme are expressed in a similar manner, but with an initial '?' rather than '@':

```
? [package:] type:name
```

## String values

Where an attribute value is a string, double backslashes ('\\') must be used to escape characters — for example, '\\n' for a newline or '\\xxxxx' for a Unicode character.

# File Features

The following sections describe how some Android features are reflected in the manifest file.

## Intent Filters

The core components of an application (its activities, services, and broadcast receivers) are activated by *intents*. An intent is a bundle of information (an [Intent](#) object) describing a desired action — including the data to be acted upon, the category of component that should perform the action, and other pertinent instructions. Android locates an appropriate component to respond to the intent, launches a new instance of the component if one is needed, and passes it the Intent object.

Components advertise their capabilities — the kinds of intents they can respond to — through *intent filters*. Since the Android system must learn which intents a component can handle before it launches the component, intent filters are specified in the manifest as [`<intent-filter>`](#) elements. A component may have any number of filters, each one describing a different capability.

An intent that explicitly names a target component will activate that component; the filter doesn't play a role. But an intent that doesn't specify a target by name can activate a component only if it can pass through one of the component's filters.

For information on how Intent objects are tested against intent filters, see a separate document, [Intents and Intent Filters](#).

## Icons and Labels

A number of elements have `icon` and `label` attributes for a small icon and a text label that can be displayed to users. Some also have a `description` attribute for longer explanatory text that can also be shown on-screen. For example, the [`<permission>`](#) element has all three of these attributes, so that when the user is asked whether to grant the permission to an application that has requested it, an icon representing the permission, the name of the permission, and a description of what it entails can all be presented to the user.

In every case, the icon and label set in a containing element become the default `icon` and `label` settings for all of the container's subelements. Thus, the icon and label set in the [`<application>`](#) element are the default icon and label for each of the application's components. Similarly, the icon and label set for a component — for example, an [`<activity>`](#) element — are the default settings for each of the component's [`<intent-filter>`](#) elements. If an [`<application>`](#) element sets a label, but an activity and its intent filter do not, the application label is treated as the label for both the activity and the intent filter.

The icon and label set for an intent filter are used to represent a component whenever the component is presented to the user as fulfilling the function advertised by the filter. For example, a filter with "android.intent.action.MAIN" and "android.intent.category.LAUNCHER" settings advertises an activity as one that initiates an application — that is, as one that should be displayed in the application launcher. The icon and label set in the filter are therefore the ones displayed in the launcher.

## Permissions

A *permission* is a restriction limiting access to a part of the code or to data on the device. The limitation is imposed to protect critical data and code that could be misused to distort or damage the user experience.

Each permission is identified by a unique label. Often the label indicates the action that's restricted. For example, here are some permissions defined by Android:

```
android.permission.CALL_EMERGENCY_NUMBERS  
android.permission.READ_OWNER_DATA  
android.permission.SET_WALLPAPER  
android.permission.DEVICE_POWER
```

A feature can be protected by at most one permission.

If an application needs access to a feature protected by a permission, it must declare that it requires that permission with a [`<uses-permission>`](#) element in the manifest. Then, when the application is installed on the device, the installer determines whether or not to grant the requested permission by checking the authorities that signed the application's certificates and, in some cases, asking the user. If the permission is granted, the application is able to use the protected features. If not, its attempts to access those features will simply fail without any notification to the user.

An application can also protect its own components (activities, services, broadcast receivers, and content providers) with permissions. It can employ any of the permissions defined by Android (listed in [`an-  
droid.Manifest.permission`](#)) or declared by other applications. Or it can define its own. A new permission is declared with the [`<permission>`](#) element. For example, an activity could be protected as follows:

```
<manifest . . . >  
    <permission android:name="com.example.project.DEBIT_ACCT" . . . />  
    <uses-permission android:name="com.example.project.DEBIT_ACCT" />  
    . . .  
    <application . . . >  
        <activity android:name="com.example.project.FreneticActivity"  
            android:permission="com.example.project.DEBIT_ACCT"  
            . . . >  
            . . .  
        </activity>  
    </application>  
</manifest>
```

Note that, in this example, the DEBIT\_ACCT permission is not only declared with the [`<permission>`](#) element, its use is also requested with the [`<uses-permission>`](#) element. Its use must be requested in order for other components of the application to launch the protected activity, even though the protection is imposed by the application itself.

If, in the same example, the `permission` attribute was set to a permission declared elsewhere (such as `an-  
droid.permission.CALL_EMERGENCY_NUMBERS`, it would not have been necessary to declare it again with a [`<permission>`](#) element. However, it would still have been necessary to request its use with [`<uses-  
permission>`](#).

The [`<permission-tree>`](#) element declares a namespace for a group of permissions that will be defined in code. And [`<permission-group>`](#) defines a label for a set of permissions (both those declared in the manifest with [`<permission>`](#) elements and those declared elsewhere). It affects only how the permissions are grouped when presented to the user. The [`<permission-group>`](#) element does not specify which permissions belong to the group; it just gives the group a name. A permission is placed in the group by assigning the group name to the [`<permission>`](#) element's `permissionGroup` attribute.

## Libraries

Every application is linked against the default Android library, which includes the basic packages for building applications (with common classes such as Activity, Service, Intent, View, Button, Application, ContentProvider, and so on).

However, some packages reside in their own libraries. If your application uses code from any of these packages, it must explicitly ask to be linked against them. The manifest must contain a separate [`<uses-library>`](#) element to name each of the libraries. (The library name can be found in the documentation for the package.)

# <action>

**syntax:**

```
<action android:name="string" />
```

**contained in:**

[<intent-filter>](#)

**description:**

Adds an action to an intent filter. An [<intent-filter>](#) element must contain one or more `<action>` elements. If it doesn't contain any, no Intent objects will get through the filter. See [Intents and Intent Filters](#) for details on intent filters and the role of action specifications within a filter.

**attributes:**

**android:name**

The name of the action. Some standard actions are defined in the [Intent](#) class as `ACTION_string` constants. To assign one of these actions to this attribute, prepend "android.intent.action." to the *string* that follows ACTION\_. For example, for ACTION\_MAIN, use "android.intent.action.MAIN" and for ACTION\_WEB\_SEARCH, use "android.intent.action.WEB\_SEARCH".

For actions you define, it's best to use the package name as a prefix to ensure uniqueness. For example, a TRANSMOGRIFY action might be specified as follows:

```
<action android:name="com.example.project.TRANSMOGRIFY" />
```

**introduced in:**

API Level 1

**see also:**

[<intent-filter>](#)

# <activity>

syntax:

```
<activity android:allowTaskReparenting=["true" | "false"]
          android:alwaysRetainTaskState=["true" | "false"]
          android:clearTaskOnLaunch=["true" | "false"]
          android:configChanges=["mcc", "mnc", "locale",
                                 "touchscreen", "keyboard", "keyboardHidden",
                                 "navigation", "screenLayout", "fontScale",
                                 "orientation", "screenSize", "smallestScreen"
                                 "size", "density"]
          android:enabled=["true" | "false"]
          android:excludeFromRecents=["true" | "false"]
          android:exported=["true" | "false"]
          android:finishOnTaskLaunch=["true" | "false"]
          android:hardwareAccelerated=["true" | "false"]
          android:icon="drawable resource"
          android:label="string resource"
          android:launchMode=["multiple" | "singleTop" |
                             "singleTask" | "singleInstance"]
          android:multiprocess=["true" | "false"]
          android:name="string"
          android:noHistory=["true" | "false"]
          android:parentActivityName="string"
          android:permission="string"
          android:process="string"
          android:screenOrientation=["unspecified" | "behind" |
                                     "landscape" | "portrait" |
                                     "reverseLandscape" | "reversePortrait" |
                                     "sensorLandscape" | "sensorPortrait" |
                                     "userLandscape" | "userPortrait" |
                                     "sensor" | "fullSensor" | "nosensor" |
                                     "user" | "fullUser" | "locked"]
          android:stateNotNeeded=["true" | "false"]
          android:taskAffinity="string"
          android:theme="resource or theme"
          android:uiOptions=["none" | "splitActionBarWhenNarrow"]
          android:windowSoftInputMode=["stateUnspecified",
                                       "stateUnchanged", "stateHidden",
                                       "stateAlwaysHidden", "stateVisible",
                                       "stateAlwaysVisible", "adjustUnspecified",
                                       "adjustResize", "adjustPan"] >
    . . .
</activity>
```

contained in:

[<application>](#)

can contain:

[<intent-filter>](#)  
[<meta-data>](#)

## **description:**

Declares an activity (an [Activity](#) subclass) that implements part of the application's visual user interface. All activities must be represented by <activity> elements in the manifest file. Any that are not declared there will not be seen by the system and will never be run.

## **attributes:**

### **android:allowTaskReparenting**

Whether or not the activity can move from the task that started it to the task it has an affinity for when that task is next brought to the front — "true" if it can move, and "false" if it must remain with the task where it started.

If this attribute is not set, the value set by the corresponding [allowTaskReparenting](#) attribute of the [<application>](#) element applies to the activity. The default value is "false".

Normally when an activity is started, it's associated with the task of the activity that started it and it stays there for its entire lifetime. You can use this attribute to force it to be re-parented to the task it has an affinity for when its current task is no longer displayed. Typically, it's used to cause the activities of an application to move to the main task associated with that application.

For example, if an e-mail message contains a link to a web page, clicking the link brings up an activity that can display the page. That activity is defined by the browser application, but is launched as part of the e-mail task. If it's reparented to the browser task, it will be shown when the browser next comes to the front, and will be absent when the e-mail task again comes forward.

The affinity of an activity is defined by the [taskAffinity](#) attribute. The affinity of a task is determined by reading the affinity of its root activity. Therefore, by definition, a root activity is always in a task with the same affinity. Since activities with "singleTask" or "singleInstance" launch modes can only be at the root of a task, re-parenting is limited to the "standard" and "singleTop" modes. (See also the [launchMode](#) attribute.)

### **android:alwaysRetainTaskState**

Whether or not the state of the task that the activity is in will always be maintained by the system — "true" if it will be, and "false" if the system is allowed to reset the task to its initial state in certain situations. The default value is "false". This attribute is meaningful only for the root activity of a task; it's ignored for all other activities.

Normally, the system clears a task (removes all activities from the stack above the root activity) in certain situations when the user re-selects that task from the home screen. Typically, this is done if the user hasn't visited the task for a certain amount of time, such as 30 minutes.

However, when this attribute is "true", users will always return to the task in its last state, regardless of how they get there. This is useful, for example, in an application like the web browser where there is a lot of state (such as multiple open tabs) that users would not like to lose.

### **android:clearTaskOnLaunch**

Whether or not all activities will be removed from the task, except for the root activity, whenever it is re-launched from the home screen — "true" if the task is always stripped down to its root activity, and "false" if not. The default value is "false". This attribute is meaningful only for activities that start a new task (the root activity); it's ignored for all other activities in the task.

When the value is "true", every time users start the task again, they are brought to its root activity regardless of what they were last doing in the task and regardless of whether they used the *Back* or *Home* button to leave it. When the value is "false", the task may be cleared of activities in some situations (see the [alwaysRetainTaskState](#) attribute), but not always.

Suppose, for example, that someone launches activity P from the home screen, and from there goes to activity Q. The user next presses *Home*, and then returns to activity P. Normally, the user would see activity Q, since that is what they were last doing in P's task. However, if P set this flag to "true", all of the activities on top of it (Q in this case) were removed when the user pressed *Home* and the task went to the background. So the user sees only P when returning to the task.

If this attribute and [allowTaskReparenting](#) are both "true", any activities that can be re-parented are moved to the task they share an affinity with; the remaining activities are then dropped, as described above.

#### **android:configChanges**

Lists configuration changes that the activity will handle itself. When a configuration change occurs at runtime, the activity is shut down and restarted by default, but declaring a configuration with this attribute will prevent the activity from being restarted. Instead, the activity remains running and its [onConfigurationChanged\(\)](#) method is called.

**Note:** Using this attribute should be avoided and used only as a last resort. Please read [Handling Runtime Changes](#) for more information about how to properly handle a restart due to a configuration change.

Any or all of the following strings are valid values for this attribute. Multiple values are separated by '|' — for example, "locale|navigation|orientation".

Value	Description
"mcc"	The IMSI mobile country code (MCC) has changed — a SIM has been detected and updated the MCC.
"mnc"	The IMSI mobile network code (MNC) has changed — a SIM has been detected and updated the MNC.
"locale"	The locale has changed — the user has selected a new language that text should be displayed in.
"touchscreen"	The touchscreen has changed. (This should never normally happen.)
"keyboard"	The keyboard type has changed — for example, the user has plugged in an external keyboard.
"keyboardHidden"	The keyboard accessibility has changed — for example, the user has revealed the hardware keyboard.
"navigation"	The navigation type (trackball/dpad) has changed. (This should never normally happen.)
"screenLayout"	The screen layout has changed — this might be caused by a different display being activated.
"fontScale"	The font scaling factor has changed — the user has selected a new global font size.
"uiMode"	The user interface mode has changed — this can be caused when the user places the device into a desk/car dock or when the night mode changes. See <a href="#">UiModeManager</a> . <i>Added in API level 8</i> .
"orientation"	The screen orientation has changed — the user has rotated the device.

**Note:** If your application targets API level 13 or higher (as declared by the [minSdkVersion](#) and [targetSdkVersion](#) attributes), then you should also declare the " screenSize" configuration, because it also changes when a device switches between portrait and landscape orientations.

"screenSize"

The current available screen size has changed. This represents a change in the currently available size, relative to the current aspect ratio, so will change when the user switches between landscape and portrait. However, if your application targets API level 12 or lower, then your activity always handles this configuration change itself (this configuration change does not restart your activity, even when running on an Android 3.2 or higher device).

*Added in API level 13.*

"smallestScreenSize"

The physical screen size has changed. This represents a change in size regardless of orientation, so will only change when the actual physical screen size has changed such as switching to an external display. A change to this configuration corresponds to a change in the [smallest-Width configuration](#). However, if your application targets API level 12 or lower, then your activity always handles this configuration change itself (this configuration change does not restart your activity, even when running on an Android 3.2 or higher device).

*Added in API level 13.*

"layoutDirection"

The layout direction has changed. For example, changing from left-to-right (LTR) to right-to-left (RTL). *Added in API level 17.*

All of these configuration changes can impact the resource values seen by the application. Therefore, when [onConfigurationChanged\(\)](#) is called, it will generally be necessary to again retrieve all resources (including view layouts, drawables, and so on) to correctly handle the change.

#### **android:enabled**

Whether or not the activity can be instantiated by the system — "true" if it can be, and "false" if not. The default value is "true".

The [`<application>`](#) element has its own [enabled](#) attribute that applies to all application components, including activities. The [`<application>`](#) and [`<activity>`](#) attributes must both be "true" (as they both are by default) for the system to be able to instantiate the activity. If either is "false", it cannot be instantiated.

#### **android:excludeFromRecents**

Whether or not the task initiated by this activity should be excluded from the list of recently used applications ("recent apps"). That is, when this activity is the root activity of a new task, this attribute determines whether the task should not appear in the list of recent apps. Set "true" if the task should be *excluded* from the list; set "false" if it should be *included*. The default value is "false".

#### **android:exported**

Whether or not the activity can be launched by components of other applications — "true" if it can be, and "false" if not. If "false", the activity can be launched only by components of the same application or applications with the same user ID.

The default value depends on whether the activity contains intent filters. The absence of any filters means that the activity can be invoked only by specifying its exact class name. This implies that the activity is intended only for application-internal use (since others would not know the class name). So in this case, the default value is "false". On the other hand, the presence of at least one filter implies that the activity is intended for external use, so the default value is "true".

This attribute is not the only way to limit an activity's exposure to other applications. You can also use a permission to limit the external entities that can invoke the activity (see the [permission](#) attribute).

#### **android:finishOnTaskLaunch**

Whether or not an existing instance of the activity should be shut down (finished) whenever the user again launches its task (chooses the task on the home screen) — "true" if it should be shut down, and "false" if not. The default value is "false".

If this attribute and [allowTaskReparenting](#) are both "true", this attribute trumps the other. The affinity of the activity is ignored. The activity is not re-parented, but destroyed.

#### **android:hardwareAccelerated**

Whether or not hardware-accelerated rendering should be enabled for this Activity — "true" if it should be enabled, and "false" if not. The default value is "false".

Starting from Android 3.0, a hardware-accelerated OpenGL renderer is available to applications, to improve performance for many common 2D graphics operations. When the hardware-accelerated renderer is enabled, most operations in Canvas, Paint, Xfermode, ColorFilter, Shader, and Camera are accelerated. This results in smoother animations, smoother scrolling, and improved responsiveness overall, even for applications that do not explicitly make use of the framework's OpenGL libraries. Because of the increased resources required to enable hardware acceleration, your app will consume more RAM.

Note that not all of the OpenGL 2D operations are accelerated. If you enable the hardware-accelerated renderer, test your application to ensure that it can make use of the renderer without errors.

#### **android:icon**

An icon representing the activity. The icon is displayed to users when a representation of the activity is required on-screen. For example, icons for activities that initiate tasks are displayed in the launcher window. The icon is often accompanied by a label (see the [android:label](#) attribute).

This attribute must be set as a reference to a drawable resource containing the image definition. If it is not set, the icon specified for the application as a whole is used instead (see the [<application>](#) element's [icon](#) attribute).

The activity's icon — whether set here or by the [<application>](#) element — is also the default icon for all the activity's intent filters (see the [<intent-filter>](#) element's [icon](#) attribute).

#### **android:label**

A user-readable label for the activity. The label is displayed on-screen when the activity must be represented to the user. It's often displayed along with the activity icon.

If this attribute is not set, the label set for the application as a whole is used instead (see the [<application>](#) element's [label](#) attribute).

The activity's label — whether set here or by the [<application>](#) element — is also the default label for all the activity's intent filters (see the [<intent-filter>](#) element's [label](#) attribute).

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

## `android:launchMode`

An instruction on how the activity should be launched. There are four modes that work in conjunction with activity flags (`FLAG_ACTIVITY_*` constants) in [Intent](#) objects to determine what should happen when the activity is called upon to handle an intent. They are:

```
"standard"  
"singleTop"  
"singleTask"  
"singleInstance"
```

The default mode is "standard".

As shown in the table below, the modes fall into two main groups, with "standard" and "singleTop" activities on one side, and "singleTask" and "singleInstance" activities on the other. An activity with the "standard" or "singleTop" launch mode can be instantiated multiple times. The instances can belong to any task and can be located anywhere in the activity stack. Typically, they're launched into the task that called [startActivity\(\)](#) (unless the Intent object contains a `FLAG_ACTIVITY_NEW_TASK` instruction, in which case a different task is chosen — see the [taskAffinity](#) attribute).

In contrast, "singleTask" and "singleInstance" activities can only begin a task. They are always at the root of the activity stack. Moreover, the device can hold only one instance of the activity at a time — only one such task.

The "standard" and "singleTop" modes differ from each other in just one respect: Every time there's a new intent for a "standard" activity, a new instance of the class is created to respond to that intent. Each instance handles a single intent. Similarly, a new instance of a "singleTop" activity may also be created to handle a new intent. However, if the target task already has an existing instance of the activity at the top of its stack, that instance will receive the new intent (in an [onNewIntent\(\)](#) call); a new instance is not created. In other circumstances — for example, if an existing instance of the "singleTop" activity is in the target task, but not at the top of the stack, or if it's at the top of a stack, but not in the target task — a new instance would be created and pushed on the stack.

The "singleTask" and "singleInstance" modes also differ from each other in only one respect: A "singleTask" activity allows other activities to be part of its task. It's always at the root of its task, but other activities (necessarily "standard" and "singleTop" activities) can be launched into that task. A "singleInstance" activity, on the other hand, permits no other activities to be part of its task. It's the only activity in the task. If it starts another activity, that activity is assigned to a different task — as if `FLAG_ACTIVITY_NEW_TASK` was in the intent.

Use Cases	Launch Mode	Multiple Instances?	Comments
Normal launches for most activities	"standard"	Yes	Default. The system always creates a new instance of the activity in the target task and routes the intent to it.
	"singleTop"	Conditionally	If an instance of the activity already exists at the top of the target task, the system routes the intent to that instance through a call to its <a href="#">onNewIntent()</a> method, rather than creating a new instance of the activity.
Specialized launches	"singleTask"	No	The system creates the activity at the root of a new task and routes the intent to it. However, if an

*(not recommended for general use)*

"singleInstance" No

instance of the activity already exists, the system routes the intent to existing instance through a call to its [onNewIntent\(\)](#) method, rather than creating a new one.

Same as "singleTask", except that the system doesn't launch any other activities into the task holding the instance. The activity is always the single and only member of its task.

As shown in the table above, standard is the default mode and is appropriate for most types of activities. SingleTop is also a common and useful launch mode for many types of activities. The other modes — singleTask and singleInstance — are not appropriate for most applications, since they result in an interaction model that is likely to be unfamiliar to users and is very different from most other applications.

Regardless of the launch mode that you choose, make sure to test the usability of the activity during launch and when navigating back to it from other activities and tasks using the *Back* button.

For more information on launch modes and their interaction with Intent flags, see the [Tasks and Back Stack](#) document.

#### **android:multiprocess**

Whether an instance of the activity can be launched into the process of the component that started it — "true" if it can be, and "false" if not. The default value is "false".

Normally, a new instance of an activity is launched into the process of the application that defined it, so all instances of the activity run in the same process. However, if this flag is set to "true", instances of the activity can run in multiple processes, allowing the system to create instances wherever they are used (provided permissions allow it), something that is almost never necessary or desirable.

#### **android:name**

The name of the class that implements the activity, a subclass of [Activity](#). The attribute value should be a fully qualified class name (such as, "com.example.project.ExtracurricularActivity"). However, as a shorthand, if the first character of the name is a period (for example, ".ExtracurricularActivity"), it is appended to the package name specified in the [<manifest>](#) element.

Once you publish your application, you [should not change this name](#) (unless you've set [android:exported="false"](#)).

There is no default. The name must be specified.

#### **android:noHistory**

Whether or not the activity should be removed from the activity stack and finished (its [finish\(\)](#) method called) when the user navigates away from it and it's no longer visible on screen — "true" if it should be finished, and "false" if not. The default value is "false".

A value of "true" means that the activity will not leave a historical trace. It will not remain in the activity stack for the task, so the user will not be able to return to it.

This attribute was introduced in API Level 3.

## **android:parentActivityName**

The class name of the logical parent of the activity. The name here must match the class name given to the corresponding `<activity>` element's [android:name](#) attribute.

The system reads this attribute to determine which activity should be started when the user presses the Up button in the action bar. The system can also use this information to synthesize a back stack of activities with [TaskStackBuilder](#).

To support API levels 4 - 16, you can also declare the parent activity with a `<meta-data>` element that specifies a value for "android.support.PARENT\_ACTIVITY". For example:

```
<activity
    android:name="com.example.app.ChildActivity"
    android:label="@string/title_child_activity"
    android:parentActivityName="com.example.myfirstapp.MainActivity" >
    <!-- Parent activity meta-data to support API level 4+ -->
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value="com.example.app.MainActivity" />
</activity>
```

For more information about declaring the parent activity to support Up navigation, read [Providing Up Navigation](#).

This attribute was introduced in API Level 16.

## **android:permission**

The name of a permission that clients must have to launch the activity or otherwise get it to respond to an intent. If a caller of [startActivity\(\)](#) or [startActivityForResult\(\)](#) has not been granted the specified permission, its intent will not be delivered to the activity.

If this attribute is not set, the permission set by the `<application>` element's [permission](#) attribute applies to the activity. If neither attribute is set, the activity is not protected by a permission.

For more information on permissions, see the [Permissions](#) section in the introduction and another document, [Security and Permissions](#).

## **android:process**

The name of the process in which the activity should run. Normally, all components of an application run in a default process name created for the application and you do not need to use this attribute. But if necessary, you can override the default process name with this attribute, allowing you to spread your app components across multiple processes.

If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the activity runs in that process. If the process name begins with a lowercase character, the activity will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage.

The `<application>` element's [process](#) attribute can set a different default process name for all components.

## **android:screenOrientation**

The orientation of the activity's display on the device.

The value can be any one of the following strings:

"unspecified"	The default value. The system chooses the orientation. The policy it uses, and therefore the choices made in specific contexts, may differ from device to device.
"behind"	The same orientation as the activity that's immediately beneath it in the activity stack.
"landscape"	Landscape orientation (the display is wider than it is tall).
"portrait"	Portrait orientation (the display is taller than it is wide).
"reverseLandscape"	Landscape orientation in the opposite direction from normal landscape. <i>Added in API level 9.</i>
"reversePortrait"	Portrait orientation in the opposite direction from normal portrait. <i>Added in API level 9.</i>
"sensorLandscape"	Landscape orientation, but can be either normal or reverse landscape based on the device sensor. <i>Added in API level 9.</i>
"sensorPortrait"	Portrait orientation, but can be either normal or reverse portrait based on the device sensor. <i>Added in API level 9.</i>
"userLandscape"	Landscape orientation, but can be either normal or reverse landscape based on the device sensor and the user's sensor preference. If the user has locked sensor-based rotation, this behaves the same as <code>landscape</code> , otherwise it behaves the same as <code>sensorLandscape</code> . <i>Added in API level 18.</i>
"userPortrait"	Portrait orientation, but can be either normal or reverse portrait based on the device sensor and the user's sensor preference. If the user has locked sensor-based rotation, this behaves the same as <code>portrait</code> , otherwise it behaves the same as <code>sensorPortrait</code> . <i>Added in API level 18.</i>
"sensor"	The orientation is determined by the device orientation sensor. The orientation of the display depends on how the user is holding the device; it changes when the user rotates the device. Some devices, though, will not rotate to all four possible orientations, by default. To allow all four orientations, use " <code>fullSensor</code> ".
"fullSensor"	The orientation is determined by the device orientation sensor for any of the 4 orientations. This is similar to " <code>sensor</code> " except this allows any of the 4 possible screen orientations, regardless of what the device will normally do (for example, some devices won't normally use reverse portrait or reverse landscape, but this enables those). <i>Added in API level 9.</i>
"nosensor"	The orientation is determined without reference to a physical orientation sensor. The sensor is ignored, so the display will not rotate based on how the user moves the device. Except for this distinction, the system chooses the orientation using the same policy as for the " <code>unspecified</code> " setting.
"user"	The user's current preferred orientation.
"fullUser"	If the user has locked sensor-based rotation, this behaves the same as <code>user</code> , otherwise it behaves the same as <code>fullSensor</code> and allows any of the 4 possible screen orientations. <i>Added in API level 18.</i>
"locked"	Locks the orientation to its current rotation, whatever that is. <i>Added in API level 18.</i>

**Note:** When you declare one of the landscape or portrait values, it is considered a hard requirement for the orientation in which the activity runs. As such, the value you declare enables filtering by services such as Google Play so your application is available only to devices that support the orientation required by your activities. For example, if you declare either "`landscape`", "`reverseLandscape`", or "`sensorLandscape`", then your application will be available only to devices that support landscape orientation. However, you should also explicitly declare that your application re-

quires either portrait or landscape orientation with the `<uses-feature>` element. For example, `<uses-feature android:name="android.hardware.screen.portrait"/>`. This is purely a filtering behavior provided by Google Play (and other services that support it) and the platform itself does not control whether your app can be installed when a device supports only certain orientations.

#### **android:stateNotNeeded**

Whether or not the activity can be killed and successfully restarted without having saved its state — "true" if it can be restarted without reference to its previous state, and "false" if its previous state is required. The default value is "false".

Normally, before an activity is temporarily shut down to save resources, its [onSaveInstanceState\(\)](#) method is called. This method stores the current state of the activity in a [Bundle](#) object, which is then passed to [onCreate\(\)](#) when the activity is restarted. If this attribute is set to "true", `onSaveInstanceState()` may not be called and `onCreate()` will be passed `null` instead of the [Bundle](#) — just as it was when the activity started for the first time.

A "true" setting ensures that the activity can be restarted in the absence of retained state. For example, the activity that displays the home screen uses this setting to make sure that it does not get removed if it crashes for some reason.

#### **android:taskAffinity**

The task that the activity has an affinity for. Activities with the same affinity conceptually belong to the same task (to the same "application" from the user's perspective). The affinity of a task is determined by the affinity of its root activity.

The affinity determines two things — the task that the activity is re-parented to (see the [allowTaskReparenting](#) attribute) and the task that will house the activity when it is launched with the [FLAG\\_ACTIVITY\\_NEW\\_TASK](#) flag.

By default, all activities in an application have the same affinity. You can set this attribute to group them differently, and even place activities defined in different applications within the same task. To specify that the activity does not have an affinity for any task, set it to an empty string.

If this attribute is not set, the activity inherits the affinity set for the application (see the [<application>](#) element's [taskAffinity](#) attribute). The name of the default affinity for an application is the package name set by the [<manifest>](#) element.

#### **android:theme**

A reference to a style resource defining an overall theme for the activity. This automatically sets the activity's context to use this theme (see [setTheme\(\)](#), and may also cause "starting" animations prior to the activity being launched (to better match what the activity actually looks like).

If this attribute is not set, the activity inherits the theme set for the application as a whole — from the [<application>](#) element's [theme](#) attribute. If that attribute is also not set, the default system theme is used. For more information, see the [Styles and Themes](#) developer guide.

#### **android:uiOptions**

Extra options for an activity's UI.

Must be one of the following values.

Value	Description
-------	-------------

"none"	No extra UI options. This is the default.
"splitActionBarWhenNarrow"	Add a bar at the bottom of the screen to display action items in the <a href="#">ActionBar</a> , when constrained for horizontal space (such as when in portrait mode on a handset). Instead of a small number of action items appearing in the action bar at the top of the screen, the action bar is split into the top navigation section and the bottom bar for action items. This ensures a reasonable amount of space is made available not only for the action items, but also for navigation and title elements at the top. Menu items are not split across the two bars; they always appear together.

For more information about the action bar, see the [Action Bar](#) developer guide.

This attribute was added in API level 14.

#### **android:windowSoftInputMode**

How the main window of the activity interacts with the window containing the on-screen soft keyboard. The setting for this attribute affects two things:

- The state of the soft keyboard — whether it is hidden or visible — when the activity becomes the focus of user attention.
- The adjustment made to the activity's main window — whether it is resized smaller to make room for the soft keyboard or whether its contents pan to make the current focus visible when part of the window is covered by the soft keyboard.

The setting must be one of the values listed in the following table, or a combination of one "state..." value plus one "adjust..." value. Setting multiple values in either group — multiple "state..." values, for example — has undefined results. Individual values are separated by a vertical bar (|). For example:

```
<activity android:windowSoftInputMode="stateVisible|adjustResize" . . . >
```

Values set here (other than "stateUnspecified" and "adjustUnspecified") override values set in the theme.

<b>Value</b>	<b>Description</b>
"stateUnspecified"	The state of the soft keyboard (whether it is hidden or visible) is not specified. The system will choose an appropriate state or rely on the setting in the theme.
	This is the default setting for the behavior of the soft keyboard.
"stateUnchanged"	The soft keyboard is kept in whatever state it was last in, whether visible or hidden, when the activity comes to the fore.
"stateHidden"	The soft keyboard is hidden when the user chooses the activity — that is, when the user affirmatively navigates forward to the activity, rather than backs into it because of leaving another activity.
"stateAlwaysHidden"	The soft keyboard is always hidden when the activity's main window has input focus.
"stateVisible"	The soft keyboard is visible when that's normally appropriate (when the user is navigating forward to the activity's main window).

The soft keyboard is made visible when the user chooses the activity — "stateAlwaysVisible" — that is, when the user affirmatively navigates forward to the activity, rather than backs into it because of leaving another activity.

It is unspecified whether the activity's main window resizes to make room for the soft keyboard, or whether the contents of the window pan to make the current focus visible on-screen. The system will automatically select one of these modes depending on whether the content of the window has any layout views that can scroll their contents. If there is such a view, the window will be resized, on the assumption that scrolling can make all of the window's contents visible within a smaller area.

This is the default setting for the behavior of the main window.

"adjustUnspecified" The activity's main window is always resized to make room for the soft keyboard on screen.

"adjustResize" The activity's main window is not resized to make room for the soft keyboard. Rather, the contents of the window are automatically panned so that the current focus is never obscured by the keyboard and users can always see what they are typing. This is generally less desirable than resizing, because the user may need to close the soft keyboard to get at and interact with obscured parts of the window.

This attribute was introduced in API Level 3.

#### introduced in:

API Level 1 for all attributes except for [noHistory](#) and [windowSoftInputMode](#), which were added in API Level 3.

#### see also:

[<application>](#)  
[<activity-alias>](#)

# <activity-alias>

**syntax:**

```
<activity-alias android:enabled=["true" | "false"]  
    android:exported=["true" | "false"]  
    android:icon="drawable resource"  
    android:label="string resource"  
    android:name="string"  
    android:permission="string"  
    android:targetActivity="string" >  
    . . .  
</activity-alias>
```

**contained in:**

[<application>](#)

**can contain:**

[<intent-filter>](#)  
[<meta-data>](#)

**description:**

An alias for an activity, named by the `targetActivity` attribute. The target must be in the same application as the alias and it must be declared before the alias in the manifest.

The alias presents the target activity as an independent entity. It can have its own set of intent filters, and they, rather than the intent filters on the target activity itself, determine which intents can activate the target through the alias and how the system treats the alias. For example, the intent filters on the alias may specify the "[android.intent.action.MAIN](#)" and "[android.intent.category.LAUNCHER](#)" flags, causing it to be represented in the application launcher, even though none of the filters on the target activity itself set these flags.

With the exception of `targetActivity`, `<activity-alias>` attributes are a subset of [<activity>](#) attributes. For attributes in the subset, none of the values set for the target carry over to the alias. However, for attributes not in the subset, the values set for the target activity also apply to the alias.

**attributes:**

**android:enabled**

Whether or not the target activity can be instantiated by the system through this alias — "true" if it can be, and "false" if not. The default value is "true".

The [<application>](#) element has its own `enabled` attribute that applies to all application components, including activity aliases. The [<application>](#) and `<activity-alias>` attributes must both be "true" for the system to be able to instantiate the target activity through the alias. If either is "false", the alias does not work.

**android:exported**

Whether or not components of other applications can launch the target activity through this alias — "true" if they can, and "false" if not. If "false", the target activity can be launched through the alias only by components of the same application as the alias or applications with the same user ID.

The default value depends on whether the alias contains intent filters. The absence of any filters means that the activity can be invoked through the alias only by specifying the exact name of the alias. This

implies that the alias is intended only for application-internal use (since others would not know its name) — so the default value is "false". On the other hand, the presence of at least one filter implies that the alias is intended for external use — so the default value is "true".

#### **android:icon**

An icon for the target activity when presented to users through the alias. See the [`<activity>`](#) element's [`icon`](#) attribute for more information.

#### **android:label**

A user-readable label for the alias when presented to users through the alias. See the [`<activity>`](#) element's [`label`](#) attribute for more information.

#### **android:name**

A unique name for the alias. The name should resemble a fully qualified class name. But, unlike the name of the target activity, the alias name is arbitrary; it does not refer to an actual class.

#### **android:permission**

The name of a permission that clients must have to launch the target activity or get it to do something via the alias. If a caller of [`startActivity\(\)`](#) or [`startActivityForResult\(\)`](#) has not been granted the specified permission, the target activity will not be activated.

This attribute supplants any permission set for the target activity itself. If it is not set, a permission is not needed to activate the target through the alias.

For more information on permissions, see the [Permissions](#) section in the introduction.

#### **android:targetActivity**

The name of the activity that can be activated through the alias. This name must match the name attribute of an [`<activity>`](#) element that precedes the alias in the manifest.

#### **introduced in:**

API Level 1

#### **see also:**

[`<activity>`](#)

# <application>

**syntax:**

```
<application android:allowTaskReparenting=["true" | "false"]
    android:allowBackup=["true" | "false"]
    android:backupAgent="string"
    android:debuggable=["true" | "false"]
    android:description="string resource"
    android:enabled=["true" | "false"]
    android:hasCode=["true" | "false"]
    android:hardwareAccelerated=["true" | "false"]
    android:icon="drawable resource"
    android:killAfterRestore=["true" | "false"]
    android:largeHeap=["true" | "false"]
    android:label="string resource"
    android:logo="drawable resource"
    android:manageSpaceActivity="string"
    android:name="string"
    android:permission="string"
    android:persistent=["true" | "false"]
    android:process="string"
    android:restoreAnyVersion=["true" | "false"]
    android:requiredAccountType="string"
    android:restrictedAccountType="string"
    android:supportsRtl=["true" | "false"]
    android:taskAffinity="string"
    android:testOnly=["true" | "false"]
    android:theme="resource or theme"
    android:uiOptions=["none" | "splitActionBarWhenNarrow"]
    android:vmSafeMode=["true" | "false"] >
    . . .
</application>
```

**contained in:**

[<manifest>](#)

**can contain:**

[<activity>](#)  
[<activity-alias>](#)  
[<service>](#)  
[<receiver>](#)  
[<provider>](#)  
[<uses-library>](#)

**description:**

The declaration of the application. This element contains subelements that declare each of the application's components and has attributes that can affect all the components. Many of these attributes (such as icon, label, permission, process, taskAffinity, and allowTaskReparenting) set default values for corresponding attributes of the component elements. Others (such as debuggable, enabled, description, and allowClearUserData) set values for the application as a whole and cannot be overridden by the components.

## attributes

### **android:allowTaskReparenting**

Whether or not activities that the application defines can move from the task that started them to the task they have an affinity for when that task is next brought to the front — "true" if they can move, and "false" if they must remain with the task where they started. The default value is "false".

The [`<activity>`](#) element has its own [`allowTaskReparenting`](#) attribute that can override the value set here. See that attribute for more information.

### **android:allowBackup**

Whether to allow the application to participate in the backup and restore infrastructure. If this attribute is set to false, no backup or restore of the application will ever be performed, even by a full-system backup that would otherwise cause all application data to be saved via adb. The default value of this attribute is true.

### **android:backupAgent**

The name of the class that implement's the application's backup agent, a subclass of [`BackupAgent`](#). The attribute value should be a fully qualified class name (such as, "`com.example.project.MyBackupAgent`"). However, as a shorthand, if the first character of the name is a period (for example, "`.MyBackupAgent`"), it is appended to the package name specified in the [`<manifest>`](#) element.

There is no default. The name must be specified.

### **android:debuggable**

Whether or not the application can be debugged, even when running on a device in user mode — "true" if it can be, and "false" if not. The default value is "false".

### **android:description**

User-readable text about the application, longer and more descriptive than the application label. The value must be set as a reference to a string resource. Unlike the label, it cannot be a raw string. There is no default value.

### **android:enabled**

Whether or not the Android system can instantiate components of the application — "true" if it can, and "false" if not. If the value is "true", each component's `enabled` attribute determines whether that component is enabled or not. If the value is "false", it overrides the component-specific values; all components are disabled.

The default value is "true".

### **android:hasCode**

Whether or not the application contains any code — "true" if it does, and "false" if not. When the value is "false", the system does not try to load any application code when launching components. The default value is "true".

An application would not have any code of its own only if it's using nothing but built-in component classes, such as an activity that uses the [`AliasActivity`](#) class, a rare occurrence.

### **android:hardwareAccelerated**

Whether or not hardware-accelerated rendering should be enabled for all activities and views in this application — "true" if it should be enabled, and "false" if not. The default value is "true" if

you've set either `minSdkVersion` or `targetSdkVersion` to "14" or higher; otherwise, it's "false".

Starting from Android 3.0 (API level 11), a hardware-accelerated OpenGL renderer is available to applications, to improve performance for many common 2D graphics operations. When the hardware-accelerated renderer is enabled, most operations in Canvas, Paint, Xfermode, ColorFilter, Shader, and Camera are accelerated. This results in smoother animations, smoother scrolling, and improved responsiveness overall, even for applications that do not explicitly make use of the framework's OpenGL libraries.

Note that not all of the OpenGL 2D operations are accelerated. If you enable the hardware-accelerated renderer, test your application to ensure that it can make use of the renderer without errors.

For more information, read the [Hardware Acceleration](#) guide.

#### **android:icon**

An icon for the application as whole, and the default icon for each of the application's components. See the individual `icon` attributes for [`<activity>`](#), [`<activity-alias>`](#), [`<service>`](#), [`<receiver>`](#), and [`<provider>`](#) elements.

This attribute must be set as a reference to a drawable resource containing the image (for example "`@drawable/icon`"). There is no default icon.

#### **android:killAfterRestore**

Whether the application in question should be terminated after its settings have been restored during a full-system restore operation. Single-package restore operations will never cause the application to be shut down. Full-system restore operations typically only occur once, when the phone is first set up. Third-party applications will not normally need to use this attribute.

The default is `true`, which means that after the application has finished processing its data during a full-system restore, it will be terminated.

#### **android:largeHeap**

Whether your application's processes should be created with a large Dalvik heap. This applies to all processes created for the application. It only applies to the first application loaded into a process; if you're using a shared user ID to allow multiple applications to use a process, they all must use this option consistently or they will have unpredictable results.

Most apps should not need this and should instead focus on reducing their overall memory usage for improved performance. Enabling this also does not guarantee a fixed increase in available memory, because some devices are constrained by their total available memory.

To query the available memory size at runtime, use the methods [`getMemoryClass\(\)`](#) or [`getLargeMemoryClass\(\)`](#).

#### **android:label**

A user-readable label for the application as a whole, and a default label for each of the application's components. See the individual `label` attributes for [`<activity>`](#), [`<activity-alias>`](#), [`<service>`](#), [`<receiver>`](#), and [`<provider>`](#) elements.

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

## **android:logo**

A logo for the application as whole, and the default logo for activities.

This attribute must be set as a reference to a drawable resource containing the image (for example "@drawable/logo"). There is no default logo.

## **android:manageSpaceActivity**

The fully qualified name of an Activity subclass that the system can launch to let users manage the memory occupied by the application on the device. The activity should also be declared with an [`<activity>`](#) element.

## **android:name**

The fully qualified name of an [Application](#) subclass implemented for the application. When the application process is started, this class is instantiated before any of the application's components.

The subclass is optional; most applications won't need one. In the absence of a subclass, Android uses an instance of the base Application class.

## **android:permission**

The name of a permission that clients must have in order to interact with the application. This attribute is a convenient way to set a permission that applies to all of the application's components. It can be overwritten by setting the `permission` attributes of individual components.

For more information on permissions, see the [Permissions](#) section in the introduction and another document, [Security and Permissions](#).

## **android:persistent**

Whether or not the application should remain running at all times — "true" if it should, and "false" if not. The default value is "false". Applications should not normally set this flag; persistence mode is intended only for certain system applications.

## **android:process**

The name of a process where all components of the application should run. Each component can override this default by setting its own `process` attribute.

By default, Android creates a process for an application when the first of its components needs to run. All components then run in that process. The name of the default process matches the package name set by the [`<manifest>`](#) element.

By setting this attribute to a process name that's shared with another application, you can arrange for components of both applications to run in the same process — but only if the two applications also share a user ID and be signed with the same certificate.

If the name assigned to this attribute begins with a colon (:), a new process, private to the application, is created when it's needed. If the process name begins with a lowercase character, a global process of that name is created. A global process can be shared with other applications, reducing resource usage.

## **android:restoreAnyVersion**

Indicates that the application is prepared to attempt a restore of any backed-up data set, even if the backup was stored by a newer version of the application than is currently installed on the device. Setting this attribute to `true` will permit the Backup Manager to attempt restore even when a version mismatch suggests that the data are incompatible. *Use with caution!*

The default value of this attribute is `false`.

### **android:requiredAccountType**

Specifies the account type required by the application in order to function. If your app requires an [Account](#), the value for this attribute must correspond to the account authenticator type used by your app (as defined by [AuthenticatorDescription](#)), such as "com.google".

The default value is null and indicates that the application can work *without* any accounts.

Because restricted profiles currently cannot add accounts, specifying this attribute **makes your app unavailable from a restricted profile** unless you also declare [an-android:restrictedAccountType](#) with the same value.

**Caution:** If the account data may reveal personally identifiable information, it's important that you declare this attribute and leave [android:restrictedAccountType](#) null, so that restricted profiles cannot use your app to access personal information that belongs to the owner user.

This attribute was added in API level 18.

### **android:restrictedAccountType**

Specifies the account type required by this application and indicates that restricted profiles are allowed to access such accounts that belong to the owner user. If your app requires an [Account](#) and restricted profiles **are allowed to access** the primary user's accounts, the value for this attribute must correspond to the account authenticator type used by your app (as defined by [AuthenticatorDescription](#)), such as "com.google".

The default value is null and indicates that the application can work *without* any accounts.

**Caution:** Specifying this attribute allows restricted profiles to use your app with accounts that belong to the owner user, which may reveal personally identifiable information. If the account may reveal personal details, you **should not** use this attribute and you should instead declare the [an-android:requiredAccountType](#) attribute to make your app unavailable to restricted profiles.

This attribute was added in API level 18.

### **android:supportsRtl**

Declares whether your application is willing to support right-to-left (RTL) layouts.

If set to `true` and [targetSdkVersion](#) is set to 17 or higher, various RTL APIs will be activated and used by the system so your app can display RTL layouts. If set to `false` or if [targetSdkVersion](#) is set to 16 or lower, the RTL APIs will be ignored or will have no effect and your app will behave the same regardless of the layout direction associated to the user's Locale choice (your layouts will always be left-to-right).

The default value of this attribute is `false`.

This attribute was added in API level 17.

### **android:taskAffinity**

An affinity name that applies to all activities within the application, except for those that set a different affinity with their own [taskAffinity](#) attributes. See that attribute for more information.

By default, all activities within an application share the same affinity. The name of that affinity is the same as the package name set by the [<manifest>](#) element.

**android:testOnly**

Indicates whether this application is only for testing purposes. For example, it may expose functionality or data outside of itself that would cause a security hole, but is useful for testing. This kind of application can be installed only through adb.

**android:theme**

A reference to a style resource defining a default theme for all activities in the application. Individual activities can override the default by setting their own [theme](#) attributes. For more information, see the [Styles and Themes](#) developer guide.

**android:uiOptions**

Extra options for an activity's UI.

Must be one of the following values.

Value	Description
"none"	No extra UI options. This is the default.
"splitActionBarWhenNarrow"	Add a bar at the bottom of the screen to display action items in the <a href="#">ActionBar</a> , when constrained for horizontal space (such as when in portrait mode on a handset). Instead of a small number of action items appearing in the action bar at the top of the screen, the action bar is split into the top navigation section and the bottom bar for action items. This ensures a reasonable amount of space is made available not only for the action items, but also for navigation and title elements at the top. Menu items are not split across the two bars; they always appear together.

For more information about the action bar, see the [Action Bar](#) developer guide.

This attribute was added in API level 14.

**android:vmSafeMode**

Indicates whether the app would like the virtual machine (VM) to operate in safe mode. The default value is "false".

**introduced in:**

API Level 1

**see also:**

[`<activity>`](#)  
[`<service>`](#)  
[`<receiver>`](#)  
[`<provider>`](#)

# <category>

**syntax:**

```
<category android:name="string" />
```

**contained in:**

[<intent-filter>](#)

**description:**

Adds a category name to an intent filter. See [Intents and Intent Filters](#) for details on intent filters and the role of category specifications within a filter.

**attributes:**

**android:name**

The name of the category. Standard categories are defined in the [Intent](#) class as CATEGORY\_*name* constants. The name assigned here can be derived from those constants by prefixing "android.intent.category." to the *name* that follows CATEGORY\_. For example, the string value for CATEGORY\_LAUNCHER is "android.intent.category.LAUNCHER".

Custom categories should use the package name as a prefix, to ensure that they are unique.

**introduced in:**

API Level 1

**see also:**

[<action>](#)

[<data>](#)

# <compatible-screens>

**syntax:**

```
<compatible-screens>
    <screen android:screenSize=["small" | "normal" | "large" | "xlarge"]
            android:screenDensity=["ldpi" | "mdpi" | "hdpi" | "xhdpi"] />
    ...
</compatible-screens>
```

**contained in:**

[<manifest>](#)

**description:**

Specifies each screen configuration with which the application is compatible. Only one instance of the `<compatible-screens>` element is allowed in the manifest, but it can contain multiple `<screen>` elements. Each `<screen>` element specifies a specific screen size-density combination with which the application is compatible.

The Android system *does not* read the `<compatible-screens>` manifest element (neither at install-time nor at runtime). This element is informational only and may be used by external services (such as Google Play) to better understand the application's compatibility with specific screen configurations and enable filtering for users. Any screen configuration that is *not* declared in this element is a screen with which the application is *not* compatible. Thus, external services (such as Google Play) should not provide the application to devices with such screens.

**Caution:** Normally, **you should not use this manifest element**. Using this element can dramatically reduce the potential user base for your application, by not allowing users to install your application if they have a device with a screen configuration that you have not listed. You should use it only as a last resort, when the application absolutely does not work with all screen configurations. Instead of using this element, you should follow the guide to [Supporting Multiple Screens](#), in order to provide complete support for multiple screens, by adding alternative resources for different screen sizes and densities.

If you want to set only a minimum screen *size* for your application, then you should use the `<supports-screens>` element. For example, if you want your application to be available only for *large* and *xlarge* screen devices, the `<supports-screens>` element allows you to declare that your application does not support *small* and *normal* screen sizes. External services (such as Google Play) will filter your application accordingly. You can also use the `<supports-screens>` element to declare whether the system should resize your application for different screen sizes.

Also see the [Filters on Google Play](#) document for more information about how Google Play filters applications using this and other manifest elements.

**child elements:**

[<screen>](#)

Specifies a single screen configuration with which the application is compatible.

At least one instance of this element must be placed inside the `<compatible-screens>` element. This element *must include both* the `android:screenSize` and `android:screenDensity` attributes (if you do not declare both attributes, then the element is ignored).

**attributes:**

**android:screenSize**

**Required.** Specifies the screen size for this screen configuration.

Accepted values:

- small
- normal
- large
- xlarge

For information about the different screen sizes, see [Supporting Multiple Screens](#).

**android:screenDensity**

**Required.** Specifies the screen density for this screen configuration.

Accepted values:

- ldpi
- mdpi
- hdpi
- xhdpi

For information about the different screen densities, see [Supporting Multiple Screens](#).

**example**

If your application is compatible with only small and normal screens, regardless of screen density, then you must specify eight different <screen> elements, because each screen size has four different density configurations. You must declare each one of these; any combination of size and density that you do *not* specify is considered a screen configuration with which your application is *not* compatible. Here's what the manifest entry looks like if your application is compatible with only small and normal screens:

```
<manifest ... >
    ...
    <compatible-screens>
        <!-- all small size screens -->
        <screen android:screenSize="small" android:screenDensity="ldpi" />
        <screen android:screenSize="small" android:screenDensity="mdpi" />
        <screen android:screenSize="small" android:screenDensity="hdpi" />
        <screen android:screenSize="small" android:screenDensity="xhdpi" />
        <!-- all normal size screens -->
        <screen android:screenSize="normal" android:screenDensity="ldpi" />
        <screen android:screenSize="normal" android:screenDensity="mdpi" />
        <screen android:screenSize="normal" android:screenDensity="hdpi" />
        <screen android:screenSize="normal" android:screenDensity="xhdpi" />
    </compatible-screens>
    <application ... >
        ...
        <application>
    </manifest>
```

**introduced in:**

API Level 9

**see also:**

[Supporting Multiple Screens](#)

[Filters on Google Play](#)

# <data>

**syntax:**

```
<data android:host="string"
      android:mimeType="string"
      android:path="string"
      android:pathPattern="string"
      android:pathPrefix="string"
      android:port="string"
      android:scheme="string" />
```

**contained in:**

[<intent-filter>](#)

**description:**

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

`scheme://host:port/path or pathPrefix or pathPattern`

These attributes are optional, but also mutually dependent: If a [scheme](#) is not specified for the intent filter, all the other URI attributes are ignored. If a [host](#) is not specified for the filter, the [port](#) attribute and all the path attributes are ignored.

All the <data> elements contained within the same [<intent-filter>](#) element contribute to the same filter. So, for example, the following filter specification,

```
<intent-filter . . .
    <data android:scheme="something" android:host="project.example.com" />
    .
    .
</intent-filter>
```

is equivalent to this one:

```
<intent-filter . . .
    <data android:scheme="something" />
    <data android:host="project.example.com" />
    .
    .
</intent-filter>
```

You can place any number of <data> elements inside an [<intent-filter>](#) to give it multiple data options. None of its attributes have default values.

Information on how intent filters work, including the rules for how Intent objects are matched against filters, can be found in another document, [Intents and Intent Filters](#). See also the [Intent Filters](#) section in the introduction.

**attributes:**

**android:host**

The host part of a URI authority. This attribute is meaningless unless a [scheme](#) attribute is also specified for the filter.

Note: host name matching in the Android framework is case-sensitive, unlike the formal RFC. As a result, you should always specify host names using lowercase letters.

#### **android:mimeType**

A MIME media type, such as `image/jpeg` or `audio/mpeg4-generic`. The subtype can be the asterisk wildcard (\*) to indicate that any subtype matches.

It's common for an intent filter to declare a `<data>` that includes only the `android:mimeType` attribute.

Note: MIME type matching in the Android framework is case-sensitive, unlike formal RFC MIME types. As a result, you should always specify MIME types using lowercase letters.

#### **android:path**

#### **android:pathPrefix**

#### **android:pathPattern**

The path part of a URI. The `path` attribute specifies a complete path that is matched against the complete path in an Intent object. The `pathPrefix` attribute specifies a partial path that is matched against only the initial part of the path in the Intent object. The `pathPattern` attribute specifies a complete path that is matched against the complete path in the Intent object, but it can contain the following wildcards:

- An asterisk ('\*') matches a sequence of 0 to many occurrences of the immediately preceding character.
- A period followed by an asterisk (".\*") matches any sequence of 0 to many characters.

Because '\' is used as an escape character when the string is read from XML (before it is parsed as a pattern), you will need to double-escape: For example, a literal '\*' would be written as "\\\\*" and a literal '\' would be written as "\\\\\". This is basically the same as what you would need to write if constructing the string in Java code.

For more information on these three types of patterns, see the descriptions of [PATTERN\\_LITERAL](#), [PATTERN\\_PREFIX](#), and [PATTERN\\_SIMPLE\\_GLOB](#) in the [PatternMatcher](#) class.

These attributes are meaningful only if the `scheme` and `host` attributes are also specified for the filter.

#### **android:port**

The port part of a URI authority. This attribute is meaningful only if the `scheme` and `host` attributes are also specified for the filter.

#### **android:scheme**

The scheme part of a URI. This is the minimal essential attribute for specifying a URI; at least one `scheme` attribute must be set for the filter, or none of the other URI attributes are meaningful.

A scheme is specified without the trailing colon (for example, `http`, rather than `http:`).

If the filter has a data type set (the `mimeType` attribute) but no scheme, the `content:` and `file:` schemes are assumed.

Note: scheme matching in the Android framework is case-sensitive, unlike the RFC. As a result, you should always specify schemes using lowercase letters.

#### **introduced in:**

API Level 1

**see also:**

[<action>](#)

[<category>](#)

# <grant-uri-permission>

**syntax:**

```
<grant-uri-permission android:path="string"  
                     android:pathPattern="string"  
                     android:pathPrefix="string" />
```

**contained in:**

[<provider>](#)

**description:**

Specifies which data subsets of the parent content provider permission can be granted for. Data subsets are indicated by the path part of a `content` : URI. (The authority part of the URI identifies the content provider.) Granting permission is a way of enabling clients of the provider that don't normally have permission to access its data to overcome that restriction on a one-time basis.

If a content provider's [grantUriPermissions](#) attribute is "true", permission can be granted for any the data under the provider's purview. However, if that attribute is "false", permission can be granted only to data subsets that are specified by this element. A provider can contain any number of `<grant-uri-permission>` elements. Each one can specify only one path (only one of the three possible attributes).

For information on how permission is granted, see the [<intent-filter>](#) element's [grantUriPermissions](#) attribute.

**attributes:**

**android:path**  
**android:pathPrefix**  
**android:pathPattern**

A path identifying the data subset or subsets that permission can be granted for. The `path` attribute specifies a complete path; permission can be granted only to the particular data subset identified by that path. The `pathPrefix` attribute specifies the initial part of a path; permission can be granted to all data subsets with paths that share that initial part. The `pathPattern` attribute specifies a complete path, but one that can contain the following wildcards:

- An asterisk ('\*') matches a sequence of 0 to many occurrences of the immediately preceding character.
- A period followed by an asterisk (".\*") matches any sequence of 0 to many characters.

Because '\' is used as an escape character when the string is read from XML (before it is parsed as a pattern), you will need to double-escape: For example, a literal '\*' would be written as "\\\\" and a literal '\' would be written as "\\\\\\". This is basically the same as what you would need to write if constructing the string in Java code.

For more information on these types of patterns, see the descriptions of [PATTERN\\_LITERAL](#), [PATTERN\\_PREFIX](#), and [PATTERN\\_SIMPLE\\_GLOB](#) in the [PatternMatcher](#) class.

**introduced in:**

API Level 1

**see also:**

the [grantUriPermissions](#) attribute of the [<provider>](#) element

# <instrumentation>

**syntax:**

```
<instrumentation android:functionalTest=["true" | "false"]  
    android:handleProfiling=["true" | "false"]  
    android:icon="drawable resource"  
    android:label="string resource"  
    android:name="string"  
    android:targetPackage="string" />
```

**contained in:**

[<manifest>](#)

**description:**

Declares an [Instrumentation](#) class that enables you to monitor an application's interaction with the system. The Instrumentation object is instantiated before any of the application's components.

**attributes:**

**android:functionalTest**

Whether or not the Instrumentation class should run as a functional test — "true" if it should, and "false" if not. The default value is "false".

**android:handleProfiling**

Whether or not the Instrumentation object will turn profiling on and off — "true" if it determines when profiling starts and stops, and "false" if profiling continues the entire time it is running. A value of "true" enables the object to target profiling at a specific set of operations. The default value is "false".

**android:icon**

An icon that represents the Instrumentation class. This attribute must be set as a reference to a drawable resource.

**android:label**

A user-readable label for the Instrumentation class. The label can be set as a raw string or a reference to a string resource.

**android:name**

The name of the [Instrumentation](#) subclass. This should be a fully qualified class name (such as, "com.example.project.StringInstrumentation"). However, as a shorthand, if the first character of the name is a period, it is appended to the package name specified in the [<manifest>](#) element.

There is no default. The name must be specified.

**android:targetPackage**

The application that the Instrumentation object will run against. An application is identified by the package name assigned in its manifest file by the [<manifest>](#) element.

**introduced in:**

API Level 1

# <intent-filter>

## syntax:

```
<intent-filter android:icon="drawable resource"  
            android:label="string resource"  
            android:priority="integer" >  
    . . .  
</intent-filter>
```

## contained in:

[<activity>](#)  
[<activity-alias>](#)  
[<service>](#)  
[<receiver>](#)

## must contain:

[<action>](#)

## can contain:

[<category>](#)  
[<data>](#)

## description:

Specifies the types of intents that an activity, service, or broadcast receiver can respond to. An intent filter declares the capabilities of its parent component — what an activity or service can do and what types of broadcasts a receiver can handle. It opens the component to receiving intents of the advertised type, while filtering out those that are not meaningful for the component.

Most of the contents of the filter are described by its [<action>](#), [<category>](#), and [<data>](#) subelements.

For a more detailed discussion of filters, see the separate [Intents and Intent Filters](#) document, as well as the [Intents Filters](#) section in the introduction.

## attributes:

### **android:icon**

An icon that represents the parent activity, service, or broadcast receiver when that component is presented to the user as having the capability described by the filter.

This attribute must be set as a reference to a drawable resource containing the image definition. The default value is the icon set by the parent component's `icon` attribute. If the parent does not specify an icon, the default is the icon set by the [<application>](#) element.

For more on intent filter icons, see [Icons and Labels](#) in the introduction.

### **android:label**

A user-readable label for the parent component. This label, rather than the one set by the parent component, is used when the component is presented to the user as having the capability described by the filter.

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

The default value is the label set by the parent component. If the parent does not specify a label, the default is the label set by the [`<application>`](#) element's [`label`](#) attribute.

For more on intent filter labels, see [Icons and Labels](#) in the introduction.

#### **android:priority**

The priority that should be given to the parent component with regard to handling intents of the type described by the filter. This attribute has meaning for both activities and broadcast receivers:

- It provides information about how able an activity is to respond to an intent that matches the filter, relative to other activities that could also respond to the intent. When an intent could be handled by multiple activities with different priorities, Android will consider only those with higher priority values as potential targets for the intent.
- It controls the order in which broadcast receivers are executed to receive broadcast messages. Those with higher priority values are called before those with lower values. (The order applies only to synchronous messages; it's ignored for asynchronous messages.)

Use this attribute only if you really need to impose a specific order in which the broadcasts are received, or want to force Android to prefer one activity over others.

The value must be an integer, such as "100". Higher numbers have a higher priority. The default value is 0. The value must be greater than -1000 and less than 1000.

Also see [setPriority\(\)](#).

#### **introduced in:**

API Level 1

#### **see also:**

[`<action>`](#)  
[`<category>`](#)  
[`<data>`](#)

# <manifest>

**syntax:**

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="string"
    android:sharedUserId="string"
    android:sharedUserLabel="string resource"
    android:versionCode="integer"
    android:versionName="string"
    android:installLocation=["auto" | "internalOnly" | "preferExternal"
    . . .
</manifest>
```

**contained in:**

*none*

**must contain:**

[<application>](#)

**can contain:**

[<instrumentation>](#)  
[<permission>](#)  
[<permission-group>](#)  
[<permission-tree>](#)  
[<uses-configuration>](#)  
[<uses-permission>](#)

[<uses-sdk>](#)

**description:**

The root element of the AndroidManifest.xml file. It must contain an [<application>](#) element and specify `xmlns:android` and `package` attributes.

**attributes:**

**`xmlns:android`**

Defines the Android namespace. This attribute should always be set to `"http://schemas.android.com/apk/res/android"`.

**`package`**

A full Java-language-style package name for the application. The name should be unique. The name may contain uppercase or lowercase letters ('A' through 'Z'), numbers, and underscores ('\_'). However, individual package name parts may only start with letters.

To avoid conflicts with other developers, you should use Internet domain ownership as the basis for your package names (in reverse). For example, applications published by Google start with `com.google`. You should also never use the `com.example` namespace when publishing your applications.

The package name serves as a unique identifier for the application. It's also the default name for the application process (see the [<application>](#) element's `process` attribute) and the default task affinity of an activity (see the [<activity>](#) element's `taskAffinity` attribute).

**Caution:** Once you publish your application, you **cannot change the package name**. The package name defines your application's identity, so if you change it, then it is considered to be a different application and users of the previous version cannot update to the new version.

#### **android:sharedUserId**

The name of a Linux user ID that will be shared with other applications. By default, Android assigns each application its own unique user ID. However, if this attribute is set to the same value for two or more applications, they will all share the same ID — provided that they are also signed by the same certificate. Application with the same user ID can access each other's data and, if desired, run in the same process.

#### **android:sharedUserLabel**

A user-readable label for the shared user ID. The label must be set as a reference to a string resource; it cannot be a raw string.

This attribute was introduced in API Level 3. It is meaningful only if the [sharedUserId](#) attribute is also set.

#### **android:versionCode**

An internal version number. This number is used only to determine whether one version is more recent than another, with higher numbers indicating more recent versions. This is not the version number shown to users; that number is set by the `versionName` attribute.

The value must be set as an integer, such as "100". You can define it however you want, as long as each successive version has a higher number. For example, it could be a build number. Or you could translate a version number in "x.y" format to an integer by encoding the "x" and "y" separately in the lower and upper 16 bits. Or you could simply increase the number by one each time a new version is released.

#### **android:versionName**

The version number shown to users. This attribute can be set as a raw string or as a reference to a string resource. The string has no other purpose than to be displayed to users. The `versionCode` attribute holds the significant version number used internally.

#### **android:installLocation**

The default install location for the application.

The following keyword strings are accepted:

<b>Value</b>	<b>Description</b>
"internalOnly"	The application must be installed on the internal device storage only. If this is set, the application will never be installed on the external storage. If the internal storage is full, then the system will not install the application. This is also the default behavior if you do not define <code>android:installLocation</code> .
"auto"	The application may be installed on the external storage, but the system will install the application on the internal storage by default. If the internal storage is full, then the system will install it on the external storage. Once installed, the user can move the application to either internal or external storage through the system settings.
"preferExternal"	The application prefers to be installed on the external storage (SD card). There is no guarantee that the system will honor this request. The application might be installed on internal storage if the external media is unavailable or full. Once installed, the user can move the application to either internal or external storage through the system settings.

**Note:** By default, your application will be installed on the internal storage and cannot be installed on the external storage unless you define this attribute to be either "auto" or "preferExternal".

When an application is installed on the external storage:

- The .apk file is saved to the external storage, but any application data (such as databases) is still saved on the internal device memory.
- The container in which the .apk file is saved is encrypted with a key that allows the application to operate only on the device that installed it. (A user cannot transfer the SD card to another device and use applications installed on the card.) Though, multiple SD cards can be used with the same device.
- At the user's request, the application can be moved to the internal storage.

The user may also request to move an application from the internal storage to the external storage. However, the system will not allow the user to move the application to external storage if this attribute is set to `internalOnly`, which is the default setting.

Read [App Install Location](#) for more information about using this attribute (including how to maintain backward compatibility).

Introduced in: API Level 8.

**introduced in:**

API Level 1 for all attributes, unless noted otherwise in the attribute description.

**see also:**

[`<application>`](#)

# <meta-data>

**syntax:**

```
<meta-data android:name="string"  
          android:resource="resource specification"  
          android:value="string" />
```

**contained in:**

[<activity>](#)  
[<activity-alias>](#)  
[<service>](#)  
[<receiver>](#)

**description:**

A name-value pair for an item of additional, arbitrary data that can be supplied to the parent component. A component element can contain any number of <meta-data> subelements. The values from all of them are collected in a single [Bundle](#) object and made available to the component as the [PackageItemInfo.metaData](#) field.

Ordinary values are specified through the [value](#) attribute. However, to assign a resource ID as the value, use the [resource](#) attribute instead. For example, the following code assigns whatever value is stored in the @string/kangaroo resource to the "zoo" name:

```
<meta-data android:name="zoo" android:value="@string/kangaroo" />
```

On the other hand, using the [resource](#) attribute would assign "zoo" the numeric ID of the resource, not the value stored in the resource:

```
<meta-data android:name="zoo" android:resource="@string/kangaroo" />
```

It is highly recommended that you avoid supplying related data as multiple separate <meta-data> entries. Instead, if you have complex data to associate with a component, store it as a resource and use the [resource](#) attribute to inform the component of its ID.

**attributes:**

## **android:name**

A unique name for the item. To ensure that the name is unique, use a Java-style naming convention — for example, "com.example.project.activity.fred".

## **android:resource**

A reference to a resource. The ID of the resource is the value assigned to the item. The ID can be retrieved from the meta-data Bundle by the [Bundle.getInt\(\)](#) method.

## **android:value**

The value assigned to the item. The data types that can be assigned as values and the Bundle methods that components use to retrieve those values are listed in the following table:

Type	Bundle method
String value, using double backslashes (\\\\") to escape characters — such as "\\n" and "\\uxxxx" for a Unicode character.	<a href="#">getString()</a>
Integer value, such as "100"	<a href="#">getInt()</a>
Boolean value, either "true" or "false"	<a href="#">getBoolean()</a>

Color value, in the form "#rgb", "#argb", "#rrggbb", or "#aarrggbb"  
Float value, such as "1.23"

[getInt\(\)](#)  
[getFloat\(\)](#)

**introduced in:**

API Level 1

# <path-permission>

syntax:

```
<path-permission android:path="string"
                 android:pathPrefix="string"
                 android:pathPattern="string"
                 android:permission="string"
                 android:readPermission="string"
                 android:writePermission="string" />
```

contained in:

[<provider>](#)

description:

Defines the path and required permissions for a specific subset of data within a content provider. This element can be specified multiple times to supply multiple paths.

attributes:

**android:path**

A complete URI path for a subset of content provider data. Permission can be granted only to the particular data identified by this path. When used to provide search suggestion content, it must be appended with "/search\_suggest\_query".

**android:pathPrefix**

The initial part of a URI path for a subset of content provider data. Permission can be granted to all data subsets with paths that share this initial part.

**android:pathPattern**

A complete URI path for a subset of content provider data, but one that can use the following wildcards:

- An asterisk ('\*'). This matches a sequence of 0 to many occurrences of the immediately preceding character.
- A period followed by an asterisk (".\*"). This matches any sequence of 0 or more characters.

Because '\' is used as an escape character when the string is read from XML (before it is parsed as a pattern), you will need to double-escape. For example, a literal '\*' would be written as "\\\*" and a literal '\\' would be written as "\\\\". This is basically the same as what you would need to write if constructing the string in Java code.

For more information on these types of patterns, see the descriptions of [PATTERN\\_LITERAL](#), [PATTERN\\_PREFIX](#), and [PATTERN\\_SIMPLE\\_GLOB](#) in the [PatternMatcher](#) class.

**android:permission**

The name of a permission that clients must have in order to read or write the content provider's data. This attribute is a convenient way of setting a single permission for both reading and writing. However, the `readPermission` and `writePermission` attributes take precedence over this one.

**android:readPermission**

A permission that clients must have in order to query the content provider.

**`android:writePermission`**

A permission that clients must have in order to make changes to the data controlled by the content provider.

**introduced in:**

API Level 4

**see also:**

[SearchManager](#)

[Manifest.permission](#)

[Security and Permissions](#)

# <permission>

syntax:

```
<permission android:description="string resource"  
           android:icon="drawable resource"  
           android:label="string resource"  
           android:name="string"  
           android:permissionGroup="string"  
           android:protectionLevel=[ "normal" | "dangerous" |  
                                "signature" | "signatureOrSystem" ] />
```

contained in:

[<manifest>](#)

description:

Declares a security permission that can be used to limit access to specific components or features of this or other applications. See the [Permissions](#) section in the introduction, and the [Security and Permissions](#) document for more information on how permissions work.

attributes:

**android:description**

A user-readable description of the permission, longer and more informative than the label. It may be displayed to explain the permission to the user — for example, when the user is asked whether to grant the permission to another application.

This attribute must be set as a reference to a string resource; unlike the `label` attribute, it cannot be a raw string.

**android:icon**

A reference to a drawable resource for an icon that represents the permission.

**android:label**

A name for the permission, one that can be displayed to users.

As a convenience, the label can be directly set as a raw string while you're developing the application. However, when the application is ready to be published, it should be set as a reference to a string resource, so that it can be localized like other strings in the user interface.

**android:name**

The name of the permission. This is the name that will be used in code to refer to the permission — for example, in a [<uses-permission>](#) element and the `permission` attributes of application components.

The name must be unique, so it should use Java-style scoping — for example, "com.example.project.PERMITTED\_ACTION".

**android:permissionGroup**

Assigns this permission to a group. The value of this attribute is the name of the group, which must be declared with the [<permission-group>](#) element in this or another application. If this attribute is not set, the permission does not belong to a group.

## **android:protectionLevel**

Characterizes the potential risk implied in the permission and indicates the procedure the system should follow when determining whether or not to grant the permission to an application requesting it. The value can be set to one of the following strings:

<b>Value</b>	<b>Meaning</b>
"normal"	The default value. A lower-risk permission that gives requesting applications access to isolated application-level features, with minimal risk to other applications, the system, or the user. The system automatically grants this type of permission to a requesting application at installation, without asking for the user's explicit approval (though the user always has the option to review these permissions before installing).
"dangerous"	A higher-risk permission that would give a requesting application access to private user data or control over the device that can negatively impact the user. Because this type of permission introduces potential risk, the system may not automatically grant it to the requesting application. For example, any dangerous permissions requested by an application may be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities.
"signature"	A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.
"signatureOrSystem"	A permission that the system grants only to applications that are in the Android system image <i>or</i> that are signed with the same certificate as the application that declared the permission. Please avoid using this option, as the <code>signature</code> protection level should be sufficient for most needs and "signatureOrSystem" works regardless of exactly where applications are installed. The "signatureOrSystem" permission is used for certain special situations where multiple vendors have applications built into a system image and need to share specific features explicitly because they are being built together.

### **introduced in:**

API Level 1

### **see also:**

[<uses-permission>](#)  
[<permission-tree>](#)  
[<permission-group>](#)

# <permission-group>

**syntax:**

```
<permission-group android:description="string resource"  
                  android:icon="drawable resource"  
                  android:label="string resource"  
                  android:name="string" />
```

**contained in:**

[<manifest>](#)

**description:**

Declares a name for a logical grouping of related permissions. Individual permission join the group through the `permissionGroup` attribute of the [<permission>](#) element. Members of a group are presented together in the user interface.

Note that this element does not declare a permission itself, only a category in which permissions can be placed. See the [<permission>](#) element for element for information on declaring permissions and assigning them to groups.

**attributes:**

**android:description**

User-readable text that describes the group. The text should be longer and more explanatory than the label. This attribute must be set as a reference to a string resource. Unlike the `label` attribute, it cannot be a raw string.

**android:icon**

An icon representing the permission. This attribute must be set as a reference to a drawable resource containing the image definition.

**android:label**

A user-readable name for the group. As a convenience, the label can be directly set as a raw string while you're developing the application. However, when the application is ready to be published, it should be set as a reference to a string resource, so that it can be localized like other strings in the user interface.

**android:name**

The name of the group. This is the name that can be assigned to a [<permission>](#) element's [<permissionGroup>](#) attribute.

**introduced in:**

API Level 1

**see also:**

[<permission>](#)  
[<permission-tree>](#)  
[<uses-permission>](#)

# <permission-tree>

**syntax:**

```
<permission-tree android:icon="drawable resource"
                 android:label="string resource" ]
                 android:name="string" />
```

**contained in:**

[<manifest>](#)

**description:**

Declares the base name for a tree of permissions. The application takes ownership of all names within the tree. It can dynamically add new permissions to the tree by calling [PackageManager.addPermission\(\)](#). Names within the tree are separated by periods ('.'). For example, if the base name is com.example.project.taxes, permissions like the following might be added:

```
com.example.project.taxes.CALCULATE
com.example.project.taxes.deductions.MAKE_SOME_UP
com.example.project.taxes.deductions.EXAGGERATE
```

Note that this element does not declare a permission itself, only a namespace in which further permissions can be placed. See the [<permission>](#) element for information on declaring permissions.

**attributes:**

**android:icon**

An icon representing all the permissions in the tree. This attribute must be set as a reference to a drawable resource containing the image definition.

**android:label**

A user-readable name for the group. As a convenience, the label can be directly set as a raw string for quick and dirty programming. However, when the application is ready to be published, it should be set as a reference to a string resource, so that it can be localized like other strings in the user interface.

**android:name**

The name that's at the base of the permission tree. It serves as a prefix to all permission names in the tree. Java-style scoping should be used to ensure that the name is unique. The name must have more than two period-separated segments in its path — for example, com.example.base is OK, but com.example is not.

**introduced in:**

API Level 1

**see also:**

[<permission>](#)

[<permission-group>](#)

[<uses-permission>](#)

# <provider>

syntax:

```
<provider android:authorities="list"
          android:enabled=["true" | "false"]
          android:exported=["true" | "false"]
          android:grantUriPermissions=["true" | "false"]
          android:icon="drawable resource"
          android:initOrder=integer"
          android:label=string resource"
          android:multiprocess=["true" | "false"]
          android:name=string"
          android:permission=string"
          android:process=string"
          android:readPermission=string"
          android:syncable=["true" | "false"]
          android:writePermission=string" >
        .
        .
        .
</provider>
```

contained in:

[application](#)

can contain:

[meta-data](#)  
[grant-uri-permission](#)  
[path-permission](#)

description:

Declares a content provider component. A content provider is a subclass of [ContentProvider](#) that supplies structured access to data managed by the application. All content providers in your application must be defined in a <provider> element in the manifest file; otherwise, the system is unaware of them and doesn't run them.

You only declare content providers that are part of your application. Content providers in other applications that you use in your application should not be declared.

The Android system stores references to content providers according to an **authority** string, part of the provider's **content URI**. For example, suppose you want to access a content provider that stores information about health care professionals. To do this, you call the method [ContentResolver.query\(\)](#), which among other arguments takes a URI that identifies the provider:

```
content://com.example.project.healthcareprovider/nurses/rn
```

The **content : scheme** identifies the URI as a content URI pointing to an Android content provider. The **authority** `com.example.project.healthcareprovider` identifies the provider itself; the Android system looks up the authority in its list of known providers and their authorities. The substring `nurses/rn` is a **path**, which the content provider can use to identify subsets of the provider data.

Notice that when you define your provider in the <provider> element, you don't include the scheme or the path in the `android:name` argument, only the authority.

For information on using and developing content providers, see the API Guide, [Content Providers](#).

## attributes:

### **android:authorities**

A list of one or more URI authorities that identify data offered by the content provider. Multiple authorities are listed by separating their names with a semicolon. To avoid conflicts, authority names should use a Java-style naming convention (such as `com.example.provider.cartoonprovider`). Typically, it's the name of the [Content-Provider](#) subclass that implements the provider

There is no default. At least one authority must be specified.

### **android:enabled**

Whether or not the content provider can be instantiated by the system — "true" if it can be, and "false" if not. The default value is "true".

The [`<application>`](#) element has its own [enabled](#) attribute that applies to all application components, including content providers. The [`<application>`](#) and [`<provider>`](#) attributes must both be "true" (as they both are by default) for the content provider to be enabled. If either is "false", the provider is disabled; it cannot be instantiated.

### **android:exported**

Whether the content provider is available for other applications to use:

- true: The provider is available to other applications. Any application can use the provider's content URI to access it, subject to the permissions specified for the provider.
- false: The provider is not available to other applications. Set `android:exported="false"` to limit access to the provider to your applications. Only applications that have the same user ID (UID) as the provider will have access to it.

The default value is "true" for applications that set either [android:minSdkVersion](#) or [android:targetSdkVersion](#) to "16" or lower. For applications that set either of these attributes to "17" or higher, the default is "false".

You can set `android:exported="false"` and still limit access to your provider by setting permissions with the [permission](#) attribute.

### **android:grantUriPermissions**

Whether or not those who ordinarily would not have permission to access the content provider's data can be granted permission to do so, temporarily overcoming the restriction imposed by the [readPermission](#), [writePermission](#), and [permission](#) attributes — "true" if permission can be granted, and "false" if not. If "true", permission can be granted to any of the content provider's data. If "false", permission can be granted only to the data subsets listed in [`<grant-uri-permission>`](#) subelements, if any. The default value is "false".

Granting permission is a way of giving an application component one-time access to data protected by a permission. For example, when an e-mail message contains an attachment, the mail application may call upon the appropriate viewer to open it, even though the viewer doesn't have general permission to look at all the content provider's data.

In such cases, permission is granted by [FLAG\\_GRANT\\_READ\\_URI\\_PERMISSION](#) and [FLAG\\_GRANT\\_WRITE\\_URI\\_PERMISSION](#) flags in the Intent object that activates the component. For example, the mail application might put `FLAG_GRANT_READ_URI_PERMISSION` in the Intent passed to `Context.startActivity()`. The permission is specific to the URI in the Intent.

If you enable this feature, either by setting this attribute to "true" or by defining [`<grant-uri-permission>`](#) subelements, you must call [`Context.revokeUriPermission\(\)`](#) when a covered URI is deleted from the provider.

See also the [`<grant-uri-permission>`](#) element.

#### **android:icon**

An icon representing the content provider. This attribute must be set as a reference to a drawable resource containing the image definition. If it is not set, the icon specified for the application as a whole is used instead (see the [`<application>`](#) element's [`icon`](#) attribute).

#### **android:initOrder**

The order in which the content provider should be instantiated, relative to other content providers hosted by the same process. When there are dependencies among content providers, setting this attribute for each of them ensures that they are created in the order required by those dependencies. The value is a simple integer, with higher numbers being initialized first.

#### **android:label**

A user-readable label for the content provided. If this attribute is not set, the label set for the application as a whole is used instead (see the [`<application>`](#) element's [`label`](#) attribute).

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

#### **android:multiprocess**

Whether or not an instance of the content provider can be created in every client process — "true" if instances can run in multiple processes, and "false" if not. The default value is "false".

Normally, a content provider is instantiated in the process of the application that defined it. However, if this flag is set to "true", the system can create an instance in every process where there's a client that wants to interact with it, thus avoiding the overhead of interprocess communication.

#### **android:name**

The name of the class that implements the content provider, a subclass of [`ContentProvider`](#). This should be a fully qualified class name (such as, "com.example.project.TransportationProvider"). However, as a shorthand, if the first character of the name is a period, it is appended to the package name specified in the [`<manifest>`](#) element.

There is no default. The name must be specified.

#### **android:permission**

The name of a permission that clients must have to read or write the content provider's data. This attribute is a convenient way of setting a single permission for both reading and writing. However, the [`readPermission`](#) and [`writePermission`](#) attributes take precedence over this one. If the [`readPermission`](#) attribute is also set, it controls access for querying the content provider. And if the [`writePermission`](#) attribute is set, it controls access for modifying the provider's data.

For more information on permissions, see the [`Permissions`](#) section in the introduction and a separate document, [`Security and Permissions`](#).

## **android:process**

The name of the process in which the content provider should run. Normally, all components of an application run in the default process created for the application. It has the same name as the application package. The [`<application>`](#) element's `process` attribute can set a different default for all components. But each component can override the default with its own `process` attribute, allowing you to spread your application across multiple processes.

If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the activity runs in that process. If the process name begins with a lowercase character, the activity will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage.

## **android:readPermission**

A permission that clients must have to query the content provider. See also the [`permission`](#) and [`writePermission`](#) attributes.

## **android:syncable**

Whether or not the data under the content provider's control is to be synchronized with data on a server—"true" if it is to be synchronized, and "false" if not.

## **android:writePermission**

A permission that clients must have to make changes to the data controlled by the content provider. See also the [`permission`](#) and [`readPermission`](#) attributes.

### **introduced in:**

API Level 1

### **see also:**

[Content Providers](#)

# <receiver>

**syntax:**

```
<receiver android:enabled=["true" | "false"]
          android:exported=["true" | "false"]
          android:icon="drawable resource"
          android:label="string resource"
          android:name="string"
          android:permission="string"
          android:process="string" >
    ...
</receiver>
```

**contained in:**

[<application>](#)

**can contain:**

[<intent-filter>](#)  
[<meta-data>](#)

**description:**

Declares a broadcast receiver (a [BroadcastReceiver](#) subclass) as one of the application's components. Broadcast receivers enable applications to receive intents that are broadcast by the system or by other applications, even when other components of the application are not running.

There are two ways to make a broadcast receiver known to the system: One is declare it in the manifest file with this element. The other is to create the receiver dynamically in code and register it with the [Context.registerReceiver\(\)](#) method. See the [BroadcastReceiver](#) class description for more on dynamically created receivers.

**attributes:**

**android:enabled**

Whether or not the broadcast receiver can be instantiated by the system — "true" if it can be, and "false" if not. The default value is "true".

The [<application>](#) element has its own [enabled](#) attribute that applies to all application components, including broadcast receivers. The [<application>](#) and [<receiver>](#) attributes must both be "true" for the broadcast receiver to be enabled. If either is "false", it is disabled; it cannot be instantiated.

**android:exported**

Whether or not the broadcast receiver can receive messages from sources outside its application — "true" if it can, and "false" if not. If "false", the only messages the broadcast receiver can receive are those sent by components of the same application or applications with the same user ID.

The default value depends on whether the broadcast receiver contains intent filters. The absence of any filters means that it can be invoked only by Intent objects that specify its exact class name. This implies that the receiver is intended only for application-internal use (since others would not normally know the class name). So in this case, the default value is "false". On the other hand, the presence of at least one filter implies that the broadcast receiver is intended to receive intents broadcast by the system or other applications, so the default value is "true".

This attribute is not the only way to limit a broadcast receiver's external exposure. You can also use a permission to limit the external entities that can send it messages (see the [permission](#) attribute).

#### **android:icon**

An icon representing the broadcast receiver. This attribute must be set as a reference to a drawable resource containing the image definition. If it is not set, the icon specified for the application as a whole is used instead (see the [application](#) element's [icon](#) attribute).

The broadcast receiver's icon — whether set here or by the [application](#) element — is also the default icon for all the receiver's intent filters (see the [intent-filter](#) element's [icon](#) attribute).

#### **android:label**

A user-readable label for the broadcast receiver. If this attribute is not set, the label set for the application as a whole is used instead (see the [application](#) element's [label](#) attribute).

The broadcast receiver's label — whether set here or by the [application](#) element — is also the default label for all the receiver's intent filters (see the [intent-filter](#) element's [label](#) attribute).

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

#### **android:name**

The name of the class that implements the broadcast receiver, a subclass of [BroadcastReceiver](#). This should be a fully qualified class name (such as, "com.example.project.ReportReceiver"). However, as a shorthand, if the first character of the name is a period (for example, ". ReportReceiver"), it is appended to the package name specified in the [manifest](#) element.

Once you publish your application, you [should not change this name](#) (unless you've set [android:exported="false"](#)).

There is no default. The name must be specified.

#### **android:permission**

The name of a permission that broadcasters must have to send a message to the broadcast receiver. If this attribute is not set, the permission set by the [application](#) element's [permission](#) attribute applies to the broadcast receiver. If neither attribute is set, the receiver is not protected by a permission.

For more information on permissions, see the [Permissions](#) section in the introduction and a separate document, [Security and Permissions](#).

#### **android:process**

The name of the process in which the broadcast receiver should run. Normally, all components of an application run in the default process created for the application. It has the same name as the application package. The [application](#) element's [process](#) attribute can set a different default for all components. But each component can override the default with its own [process](#) attribute, allowing you to spread your application across multiple processes.

If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the broadcast receiver runs in that process. If the process name begins

with a lowercase character, the receiver will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage.

**introduced in:**

API Level 1

# <service>

**syntax:**

```
<service android:enabled=["true" | "false"]
         android:exported=["true" | "false"]
         android:icon="drawable resource"
         android:isolatedProcess=["true" | "false"]
         android:label="string resource"
         android:name="string"
         android:permission="string"
         android:process="string" >
    . . .
</service>
```

**contained in:**

[<application>](#)

**can contain:**

[<intent-filter>](#)  
[<meta-data>](#)

**description:**

Declares a service (a [Service](#) subclass) as one of the application's components. Unlike activities, services lack a visual user interface. They're used to implement long-running background operations or a rich communications API that can be called by other applications.

All services must be represented by `<service>` elements in the manifest file. Any that are not declared there will not be seen by the system and will never be run.

**attributes:**

**android:enabled**

Whether or not the service can be instantiated by the system — "true" if it can be, and "false" if not. The default value is "true".

The [<application>](#) element has its own `enabled` attribute that applies to all application components, including services. The [<application>](#) and `<service>` attributes must both be "true" (as they both are by default) for the service to be enabled. If either is "false", the service is disabled; it cannot be instantiated.

**android:exported**

Whether or not components of other applications can invoke the service or interact with it — "true" if they can, and "false" if not. When the value is "false", only components of the same application or applications with the same user ID can start the service or bind to it.

The default value depends on whether the service contains intent filters. The absence of any filters means that it can be invoked only by specifying its exact class name. This implies that the service is intended only for application-internal use (since others would not know the class name). So in this case, the default value is "false". On the other hand, the presence of at least one filter implies that the service is intended for external use, so the default value is "true".

This attribute is not the only way to limit the exposure of a service to other applications. You can also use a permission to limit the external entities that can interact with the service (see the [permission](#) attribute).

#### **android:icon**

An icon representing the service. This attribute must be set as a reference to a drawable resource containing the image definition. If it is not set, the icon specified for the application as a whole is used instead (see the [application](#) element's [icon](#) attribute).

The service's icon — whether set here or by the [application](#) element — is also the default icon for all the service's intent filters (see the [intent-filter](#) element's [icon](#) attribute).

#### **android:isolatedProcess**

If set to true, this service will run under a special process that is isolated from the rest of the system and has no permissions of its own. The only communication with it is through the Service API (binding and starting).

#### **android:label**

A name for the service that can be displayed to users. If this attribute is not set, the label set for the application as a whole is used instead (see the [application](#) element's [label](#) attribute).

The service's label — whether set here or by the [application](#) element — is also the default label for all the service's intent filters (see the [intent-filter](#) element's [label](#) attribute).

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

#### **android:name**

The name of the [Service](#) subclass that implements the service. This should be a fully qualified class name (such as, "com.example.project.RoomService"). However, as a shorthand, if the first character of the name is a period (for example, ".RoomService"), it is appended to the package name specified in the [manifest](#) element.

Once you publish your application, you [should not change this name](#) (unless you've set [android:exported="false"](#)).

There is no default. The name must be specified.

#### **android:permission**

The name of a permission that that an entity must have in order to launch the service or bind to it. If a caller of [startService\(\)](#), [bindService\(\)](#), or [stopService\(\)](#), has not been granted this permission, the method will not work and the Intent object will not be delivered to the service.

If this attribute is not set, the permission set by the [application](#) element's [permission](#) attribute applies to the service. If neither attribute is set, the service is not protected by a permission.

For more information on permissions, see the [Permissions](#) section in the introduction and a separate document, [Security and Permissions](#).

#### **android:process**

The name of the process where the service is to run. Normally, all components of an application run in the default process created for the application. It has the same name as the application package. The [application](#) element's [process](#) attribute can set a different default for all components. But

component can override the default with its own `process` attribute, allowing you to spread your application across multiple processes.

If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the service runs in that process. If the process name begins with a lowercase character, the service will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage.

**see also:**

[<application>](#)  
[<activity>](#)

**introduced in:**

API Level 1

# <supports-gl-texture>



## Google Play Filtering

Google Play filters applications according to the texture compression formats that they support, to ensure that they can be installed only on devices that can handle their textures properly. You can use texture compression filtering as a way of targeting specific device types, based on GPU platform.

For important information about how Google Play uses <supports-gl-texture> elements as the basis for filtering, please read [Google Play and texture compression filtering](#), below.

## syntax:

```
<supports-gl-texture  
    android:name="string" />
```

## contained in:

[<manifest>](#)

## description:

Declares a single GL texture compression format that is supported by the application.

An application "supports" a GL texture compression format if it is capable of providing texture assets that are compressed in that format, once the application is installed on a device. The application can provide the compressed assets locally, from inside the .apk, or it can download them from a server at runtime.

Each <supports-gl-texture> element declares exactly one supported texture compression format, specified as the value of a android:name attribute. If your application supports multiple texture compression formats, you can declare multiple <supports-gl-texture> elements. For example:

```
<supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />  
<supports-gl-texture android:name="GL_OES_compressed_paletted_texture" />
```

Declared <supports-gl-texture> elements are informational, meaning that the Android system itself does not examine the elements at install time to ensure matching support on the device. However, other services (such as Google Play) or applications can check your application's <supports-gl-texture> declarations as part of handling or interacting with your application. For this reason, it's very important that you declare all of the texture compression formats (from the list below) that your application is capable of supporting.

Applications and devices typically declare their supported GL texture compression formats using the same set of well-known strings, as listed below. The set of format strings may grow over time, as needed, and since the values are strings, applications are free to declare other formats as needed.

Assuming that the application is built with SDK Platform Tools r3 or higher, filtering based on the <supports-gl-texture> element is activated for all API levels.

## attributes:

### **android:name**

Specifies a single GL texture compression format supported by the application, as a descriptor string. Common descriptor values are listed in the table below.

#### Texture Compression Format Descriptor

#### Comments

GL_OES_compressed_ETC1_RGB8_texture	Ericsson texture compression. Specified in OpenGL ES 2.0 and available in all Android-powered devices that support OpenGL ES 2.0.
GL_OES_compressed_paletted_texture	Generic paletted texture compression.
GL_AMD_compressed_3DC_texture	ATI 3Dc texture compression.
GL_AMD_compressed_ATC_texture	ATI texture compression. Available on devices running Adreno GPU, including HTC Nexus One, Droid Incredible, EVO, and others. For widest compatibility, devices may also declare a <supports-gl-texture> element with the descriptor
GL_EXT_texture_compression_latc	GL_ATI_texture_compression_atitc. Luminance alpha texture compression.
GL_EXT_texture_compression_dxt1	S3 DXT1 texture compression. Supported on devices running Nvidia Tegra2 platform, including Motorola Xoom, Motorola Atrix, Droid Bionic, and others.
GL_EXT_texture_compression_s3tc	S3 texture compression, nonspecific to DXT variant. Supported on devices running Nvidia Tegra2 platform, including Motorola Xoom, Motorola Atrix, Droid Bionic, and others. If your application requires a specific DXT variant, declare that descriptor instead of this one.
GL_IMG_texture_compression_pvrtc	PowerVR texture compression. Available in devices running PowerVR SGX530/540 GPU, such as Motorola DROID series; Samsung Galaxy S, Nexus S, and Galaxy Tab; and others.

## see also:

- [Filters on Google Play](#)

## Google Play and texture compression filtering

Google Play filters the applications that are visible to users, so that users can see and download only those applications that are compatible with their devices. One of the ways it filters applications is by texture compression compatibility, giving you control over the availability of your application to various devices, based on the capabilities of their GPUs.

To determine an application's texture compression compatibility with a given user's device, Google Play compares:

- Texture compression formats that are supported by the application — an application declares its supported texture compression formats in <supports-gl-texture> elements in its manifest with...
- Texture compression formats that are supported by the GPU on the device — a device reports the formats it supports as read-only system properties.

Each time you upload an application to the Google Play Developer Console, Google Play scans the application's manifest file and looks for any <supports-gl-texture> elements. It extracts the format descriptors from the elements and stores them internally as metadata associated with the application .apk and the application version.

When a user searches or browses for applications on Google Play, the service compares the texture compression formats supported by the application with those supported by the user's device. The comparison is based on the format descriptor strings and a match must be exact.

If *any* of an application's supported texture compression formats is also supported by the device, Google Play allows the user to see the application and potentially download it. Otherwise, if none of the application's formats is supported by the device, Google Play filters the application so that it is not available for download.

If an application does not declare any <supports-gl-texture> elements, Google Play does not apply any filtering based on GL texture compression format.

# <supports-screens>

## syntax:

```
<supports-screens android:resizeable=["true" | "false"]  
                  android:smallScreens=["true" | "false"]  
                  android:normalScreens=["true" | "false"]  
                  android:largeScreens=["true" | "false"]  
                  android:xlargeScreens=["true" | "false"]  
                  android:anyDensity=["true" | "false"]  
                  android:requiresSmallestWidthDp="integer"  
                  android:compatibleWidthLimitDp="integer"  
                  android:largestWidthLimitDp="integer"/>
```

## contained in:

[<manifest>](#)

## description:

Lets you specify the screen sizes your application supports and enable [screen compatibility mode](#) for screens larger than what your application supports. It's important that you always use this element in your application to specify the screen sizes your application supports.

An application "supports" a given screen size if it resizes properly to fill the entire screen. Normal resizing applied by the system works well for most applications and you don't have to do any extra work to make your application work on screens larger than a handset device. However, it's often important that you optimize your application's UI for different screen sizes by providing [alternative layout resources](#). For instance, you might want to modify the layout of an activity when it is on a tablet compared to when running on a handset device.

However, if your application does not work well when resized to fit different screen sizes, you can use the attributes of the <supports-screens> element to control whether your application should be distributed to smaller screens or have its UI scaled up ("zoomed") to fit larger screens using the system's [screen compatibility mode](#). When you have not designed for larger screen sizes and the normal resizing does not achieve the appropriate results, screen compatibility mode will scale your UI by emulating a *normal* size screen and medium density, then zooming in so that it fills the entire screen. Beware that this causes pixelation and blurring of your UI, so it's better if you optimize your UI for large screens.

**Note:** Android 3.2 introduces new attributes: `android:requiresSmallestWidthDp`, `android:compatibleWidthLimitDp`, and `android:largestWidthLimitDp`. If you're developing your application for Android 3.2 and higher, you should use these attributes to declare your screen size support, instead of the attributes based on generalized screen sizes.

For more information about how to properly support different screen sizes so that you can avoid using screen compatibility mode with your application, read [Supporting Multiple Screens](#).

## attributes:

### `android:resizeable`

Indicates whether the application is resizeable for different screen sizes. This attribute is true, by default. If set false, the system will run your application in [screen compatibility mode](#) on large screens.

**This attribute is deprecated.** It was introduced to help applications transition from Android 1.5 to 1.6, when support for multiple screens was first introduced. You should not use it.

#### **android:smallScreens**

Indicates whether the application supports smaller screen form-factors. A small screen is defined as one with a smaller aspect ratio than the "normal" (traditional HVGA) screen. An application that does not support small screens *will not be available* for small screen devices from external services (such as Google Play), because there is little the platform can do to make such an application work on a smaller screen. This is "true" by default.

#### **android:normalScreens**

Indicates whether an application supports the "normal" screen form-factors. Traditionally this is an HVGA medium density screen, but WQVGA low density and WVGA high density are also considered to be normal. This attribute is "true" by default.

#### **android:largeScreens**

Indicates whether the application supports larger screen form-factors. A large screen is defined as a screen that is significantly larger than a "normal" handset screen, and thus might require some special care on the application's part to make good use of it, though it may rely on resizing by the system to fill the screen.

The default value for this actually varies between some versions, so it's better if you explicitly declare this attribute at all times. Beware that setting it "false" will generally enable [screen compatibility mode](#).

#### **android:xlargeScreens**

Indicates whether the application supports extra large screen form-factors. An xlarge screen is defined as a screen that is significantly larger than a "large" screen, such as a tablet (or something larger) and may require special care on the application's part to make good use of it, though it may rely on resizing by the system to fill the screen.

The default value for this actually varies between some versions, so it's better if you explicitly declare this attribute at all times. Beware that setting it "false" will generally enable [screen compatibility mode](#).

This attribute was introduced in API level 9.

#### **android:anyDensity**

Indicates whether the application includes resources to accommodate any screen density.

For applications that support Android 1.6 (API level 4) and higher, this is "true" by default and **you should not set it "false"** unless you're absolutely certain that it's necessary for your application to work. The only time it might be necessary to disable this is if your app directly manipulates bitmaps (see the [Supporting Multiple Screens](#) document for more information).

#### **android:requiresSmallestWidthDp**

Specifies the minimum smallestWidth required. The smallestWidth is the shortest dimension of the screen space (in dp units) that must be available to your application UI—that is, the shortest of the available screen's two dimensions. So, in order for a device to be considered compatible with your application, the device's smallestWidth must be equal to or greater than this value. (Usually, the value you supply for this is the "smallest width" that your layout supports, regardless of the screen's current orientation.)

For example, a typical handset screen has a smallestWidth of 320dp, a 7" tablet has a smallestWidth of 600dp, and a 10" tablet has a smallestWidth of 720dp. These values are generally the smallestWidth because they are the shortest dimension of the screen's available space.

The size against which your value is compared takes into account screen decorations and system UI. For example, if the device has some persistent UI elements on the display, the system declares the device's smallestWidth as one that is smaller than the actual screen size, accounting for these UI elements because those are screen pixels not available for your UI. Thus, the value you use should be the minimum width required by your layout, regardless of the screen's current orientation.

If your application properly resizes for smaller screen sizes (down to the *small* size or a minimum width of 320dp), you do not need to use this attribute. Otherwise, you should use a value for this attribute that matches the smallest value used by your application for the [smallest screen width qualifier](#) (`sw<N>dp`).

**Caution:** The Android system does not pay attention to this attribute, so it does not affect how your application behaves at runtime. Instead, it is used to enable filtering for your application on services such as Google Play. However, **Google Play currently does not support this attribute for filtering** (on Android 3.2), so you should continue using the other size attributes if your application does not support small screens.

This attribute was introduced in API level 13.

#### **`android:compatibleWidthLimitDp`**

This attribute allows you to enable [screen compatibility mode](#) as a user-optimal feature by specifying the maximum "smallest screen width" for which your application is designed. If the smallest side of a device's available screen is greater than your value here, users can still install your application, but are offered to run it in screen compatibility mode. By default, screen compatibility mode is disabled and your layout is resized to fit the screen as usual, but a button is available in the system bar that allows the user to toggle screen compatibility mode on and off.

If your application is compatible with all screen sizes and its layout properly resizes, you do not need to use this attribute.

**Note:** Currently, screen compatibility mode emulates only handset screens with a 320dp width, so screen compatibility mode is not applied if your value for `android:compatibleWidthLimitDp` is larger than 320.

This attribute was introduced in API level 13.

#### **`android:largestWidthLimitDp`**

This attribute allows you to force-enable [screen compatibility mode](#) by specifying the maximum "smallest screen width" for which your application is designed. If the smallest side of a device's available screen is greater than your value here, the application runs in screen compatibility mode with no way for the user to disable it.

If your application is compatible with all screen sizes and its layout properly resizes, you do not need to use this attribute. Otherwise, you should first consider using the [`an-droid:compatibleWidthLimitDp`](#) attribute. You should use the `an-droid:largestWidthLimitDp` attribute only when your application is functionally broken when resized for larger screens and screen compatibility mode is the only way that users should use your application.

**Note:** Currently, screen compatibility mode emulates only handset screens with a 320dp width, so screen compatibility mode is not applied if your value for `android:largestWidthLimitDp` is larger than 320.

This attribute was introduced in API level 13.

**introduced in:**

API Level 4

**see also:**

- [Supporting Multiple Screens](#)
- [DisplayMetrics](#)

# <uses-configuration>

syntax:

```
<uses-configuration
    android:reqFiveWayNav=["true" | "false"]
    android:reqHardKeyboard=["true" | "false"]
    android:reqKeyboardType=["undefined" | "nokeys" | "qwerty" | "twelvekey"]
    android:reqNavigation=["undefined" | "nonav" | "dpad" | "trackball" | "wheel"]
    android:reqTouchScreen=["undefined" | "notouch" | "stylus" | "finger"] />
```

contained in:

[manifest](#)

description:

Indicates what hardware and software features the application requires. For example, an application might specify that it requires a physical keyboard or a particular navigation device, like a trackball. The specification is used to avoid installing the application on devices where it will not work.

If an application can work with different device configurations, it should include separate <uses-configuration> declarations for each one. Each declaration must be complete. For example, if an application requires a five-way navigation control, a touch screen that can be operated with a finger, and either a standard QWERTY keyboard or a numeric 12-key keypad like those found on most phones, it would specify these requirements with two <uses-configuration> elements as follows:

```
<uses-configuration android:reqFiveWayNav="true" android:reqTouchScreen="finger"
                    android:reqKeyboardType="qwerty" />
<uses-configuration android:reqFiveWayNav="true" android:reqTouchScreen="finger"
                    android:reqKeyboardType="twelvekey" />
```

attributes:

**android:reqFiveWayNav**

Whether or not the application requires a five-way navigation control — "true" if it does, and "false" if not. A five-way control is one that can move the selection up, down, right, or left, and also provides a way of invoking the current selection. It could be a D-pad (directional pad), trackball, or other device.

If an application requires a directional control, but not a control of a particular type, it can set this attribute to "true" and ignore the [reqNavigation](#) attribute. However, if it requires a particular type of directional control, it can ignore this attribute and set [reqNavigation](#) instead.

**android:reqHardKeyboard**

Whether or not the application requires a hardware keyboard — "true" if it does, and "false" if not.

**android:reqKeyboardType**

The type of keyboard the application requires, if any at all. This attribute does not distinguish between hardware and software keyboards. If a hardware keyboard of a certain type is required, specify the type here and also set the [reqHardKeyboard](#) attribute to "true".

The value must be one of the following strings:

**Value**

**Description**

" <code>undefined</code> "	The application does not require a keyboard. (A keyboard requirement is not defined.) This is the default value.
" <code>nokeys</code> "	The application does not require a keyboard.
" <code>qwerty</code> "	The application requires a standard QWERTY keyboard.
" <code>twelvekey</code> "	The application requires a twelve-key keypad, like those on most phones — with keys for the digits from 0 through 9 plus star (*) and pound (#) keys.

#### **android:reqNavigation**

The navigation device required by the application, if any. The value must be one of the following strings:

<b>Value</b>	<b>Description</b>
" <code>undefined</code> "	The application does not require any type of navigation control. (The navigation requirement is not defined.) This is the default value.
" <code>nonav</code> "	The application does not require a navigation control.
" <code>dpad</code> "	The application requires a D-pad (directional pad) for navigation.
" <code>trackball</code> "	The application requires a trackball for navigation.
" <code>wheel</code> "	The application requires a navigation wheel.

If an application requires a navigational control, but the exact type of control doesn't matter, it can set the [reqFiveWayNav](#) attribute to "`true`" rather than set this one.

#### **android:reqTouchScreen**

The type of touch screen the application requires, if any at all. The value must be one of the following strings:

<b>Value</b>	<b>Description</b>
" <code>undefined</code> "	The application doesn't require a touch screen. (The touch screen requirement is undefined.) This is the default value.
" <code>notouch</code> "	The application doesn't require a touch screen.
" <code>stylus</code> "	The application requires a touch screen that's operated with a stylus.
" <code>finger</code> "	The application requires a touch screen that can be operated with a finger.

#### **introduced in:**

API Level 3

#### **see also:**

- [configChanges](#) attribute of the [<activity>](#) element

- [ConfigurationInfo](#)

# <uses-feature>

## In this document

1. [Google Play and Feature-Based Filtering](#)
  1. [Filtering based on explicitly declared features](#)
  2. [Filtering based on implicit features](#)
  3. [Special handling for Bluetooth feature](#)
  4. [Testing the features required by your application](#)
2. [Features Reference](#)
  1. [Hardware features](#)
  2. [Software features](#)
  3. [Permissions that Imply Feature Requirements](#)



### Google Play Filtering

Google Play uses the <uses-feature> elements declared in your app manifest to filter your app from devices that do not meet its hardware and software feature requirements.

By specifying the features that your application requires, you enable Google Play to present your application only to users whose devices meet the application's feature requirements, rather than presenting it to all users.

For important information about how Google Play uses features as the basis for filtering, please read [Google Play and Feature-Based Filtering](#), below.

#### syntax:

```
<uses-feature  
    android:name="string"  
    android:required=["true" | "false"]  
    android:glEsVersion="integer" />
```

#### contained in:

[<manifest>](#)

#### description:

Declares a single hardware or software feature that is used by the application.

The purpose of a <uses-feature> declaration is to inform any external entity of the set of hardware and software features on which your application depends. The element offers a `required` attribute that lets you specify whether your application requires and cannot function without the declared feature, or whether it prefers to have the feature but can function without it. Because feature support can vary across Android devices, the <uses-feature> element serves an important role in letting an application describe the device-variable features that it uses.

The set of available features that your application declares corresponds to the set of feature constants made available by the Android [PackageManager](#), which are listed for convenience in the [Features Reference](#) tables at the bottom of this document.

You must specify each feature in a separate `<uses-feature>` element, so if your application requires multiple features, it would declare multiple `<uses-feature>` elements. For example, an application that requires both Bluetooth and camera features in the device would declare these two elements:

```
<uses-feature android:name="android.hardware.bluetooth" />
<uses-feature android:name="android.hardware.camera" />
```

In general, you should always make sure to declare `<uses-feature>` elements for all of the features that your application requires.

Declared `<uses-feature>` elements are informational only, meaning that the Android system itself does not check for matching feature support on the device before installing an application. However, other services (such as Google Play) or applications may check your application's `<uses-feature>` declarations as part of handling or interacting with your application. For this reason, it's very important that you declare all of the features (from the list below) that your application uses.

For some features, there may exist a specific attribute that allows you to define a version of the feature, such as the version of Open GL used (declared with [glEsVersion](#)). Other features that either do or do not exist for a device, such as a camera, are declared using the [name](#) attribute.

Although the `<uses-feature>` element is only activated for devices running API Level 4 or higher, it is recommended to include these elements for all applications, even if the [minSdkVersion](#) is "3" or lower. Devices running older versions of the platform will simply ignore the element.

**Note:** When declaring a feature, remember that you must also request permissions as appropriate. For example, you must still request the [CAMERA](#) permission before your application can access the camera API. Requesting the permission grants your application access to the appropriate hardware and software, while declaring the features used by your application ensures proper device compatibility.

## attributes:

### **android:name**

Specifies a single hardware or software feature used by the application, as a descriptor string. Valid descriptor values are listed in the [Hardware features](#) and [Software features](#) tables, below.

### **android:required**

Boolean value that indicates whether the application requires the feature specified in `android:name`.

- When you declare `"android:required=true"` for a feature, you are specifying that the application *cannot function, or is not designed to function*, when the specified feature is not present on the device.
- When you declare `"android:required=false"` for a feature, it means that the application *prefers to use the feature* if present on the device, but that it *is designed to function without the specified feature*, if necessary.

The default value for `android:required` if not declared is `"true"`.

### **android:glEsVersion**

The OpenGL ES version required by the application. The higher 16 bits represent the major number and the lower 16 bits represent the minor number. For example, to specify OpenGL ES version 2.0, you would set the value as `"0x00020000"`. To specify OpenGL ES 2.1, if/when such a version were made available, you would set the value as `"0x00020001"`.

An application should specify at most one `android:glEsVersion` attribute in its manifest. If it specifies more than one, the `android:glEsVersion` with the numerically highest value is used and any other values are ignored.

If an application does not specify an `android:glEsVersion` attribute, then it is assumed that the application requires only OpenGL ES 1.0, which is supported by all Android-powered devices.

An application can assume that if a platform supports a given OpenGL ES version, it also supports all numerically lower OpenGL ES versions. Therefore, an application that requires both OpenGL ES 1.0 and OpenGL ES 2.0 must specify that it requires OpenGL ES 2.0.

An application that can work with any of several OpenGL ES versions should only specify the numerically lowest version of OpenGL ES that it requires. (It can check at run-time whether a higher level of OpenGL ES is available.)

**introduced in:**

API Level 4

**see also:**

- [PackageManager](#)
- [FeatureInfo](#)
- [ConfigurationInfo](#)
- [<uses-permission>](#)
- [Filters on Google Play](#)

## Google Play and Feature-Based Filtering

Google Play filters the applications that are visible to users, so that users can see and download only those applications that are compatible with their devices. One of the ways it filters applications is by feature compatibility.

To determine an application's feature compatibility with a given user's device, Google Play compares:

- Features required by the application — an application declares features in `<uses-feature>` elements in its manifest with...
- Features available on the device, in hardware or software — a device reports the features it supports as read-only system properties.

To ensure an accurate comparison of features, the Android Package Manager provides a shared set of feature constants that both applications and devices use to declare feature requirements and support. The available feature constants are listed in the [Features Reference](#) tables at the bottom of this document, and in the class documentation for [PackageManager](#).

When the user launches Google Play, the application queries the Package Manager for the list of features available on the device by calling `getSystemAvailableFeatures()`. The Store application then passes the features list up to Google Play when establishing the session for the user.

Each time you upload an application to the Google Play Developer Console, Google Play scans the application's manifest file. It looks for `<uses-feature>` elements and evaluates them in combination with other elements, in some cases, such as `<uses-sdk>` and `<uses-permission>` elements. After establishing the application's set of required features, it stores that list internally as metadata associated with the application .apk and the application version.

When a user searches or browses for applications using the Google Play application, the service compares the features needed by each application with the features available on the user's device. If all of an application's required features are present on the device, Google Play allows the user to see the application and potentially download it. If any required feature is not supported by the device, Google Play filters the application so that it is not visible to the user and not available for download.

Because the features you declare in `<uses-feature>` elements directly affect how Google Play filters your application, it's important to understand how Google Play evaluates the application's manifest and establishes the set of required features. The sections below provide more information.

## Filtering based on explicitly declared features

An explicitly declared feature is one that your application declares in a `<uses-feature>` element. The feature declaration can include an `android:required=["true" | "false"]` attribute (if you are compiling against API level 5 or higher), which lets you specify whether the application absolutely requires the feature and cannot function properly without it ("`true`"), or whether the application prefers to use the feature if available, but is designed to run without it ("`false`").

Google Play handles explicitly declared features in this way:

- If a feature is explicitly declared as being required, Google Play adds the feature to the list of required features for the application. It then filters the application from users on devices that do not provide that feature. For example:

```
<uses-feature android:name="android.hardware.camera" android:required="true"
```

- If a feature is explicitly declared as *not* being required, Google Play *does not* add the feature to the list of required features. For that reason, an explicitly declared non-required feature is never considered when filtering the application. Even if the device does not provide the declared feature, Google Play will still consider the application compatible with the device and will show it to the user, unless other filtering rules apply. For example:

```
<uses-feature android:name="android.hardware.camera" android:required="false"
```

- If a feature is explicitly declared, but without an `android:required` attribute, Google Play assumes that the feature is required and sets up filtering on it.

In general, if your application is designed to run on Android 1.6 and earlier versions, the `android:required` attribute is not available in the API and Google Play assumes that any and all `<uses-feature>` declarations are required.

**Note:** By declaring a feature explicitly and including an `android:required="false"` attribute, you can effectively disable all filtering on Google Play for the specified feature.

## Filtering based on implicit features

An *implicit* feature is one that an application requires in order to function properly, but which is *not* declared in a `<uses-feature>` element in the manifest file. Strictly speaking, every application should *always* declare all features that it uses or requires, so the absence of a declaration for a feature used by an application should be considered an error. However, as a safeguard for users and developers, Google Play looks for implicit features in each application and sets up filters for those features, just as it would do for an explicitly declared feature.

An application might require a feature but not declare it because:

- The application was compiled against an older version of the Android library (Android 1.5 or earlier) and the `<uses-feature>` element was not available.
- The developer incorrectly assumed that the feature would be present on all devices and a declaration was unnecessary.
- The developer omitted the feature declaration accidentally.
- The developer declared the feature explicitly, but the declaration was not valid. For example, a spelling error in the `<uses-feature>` element name or an unrecognized string value for the `android:name` attribute would invalidate the feature declaration.

To account for the cases above, Google Play attempts to discover an application's implied feature requirements by examining *other elements* declared in the manifest file, specifically, `<uses-permission>` elements.

If an application requests hardware-related permissions, Google Play *assumes that the application uses the underlying hardware features and therefore requires those features*, even though there might be no corresponding to `<uses-feature>` declarations. For such permissions, Google Play adds the underlying hardware features to the metadata that it stores for the application and sets up filters for them.

For example, if an application requests the `CAMERA` permission but does not declare a `<uses-feature>` element for `android.hardware.camera`, Google Play considers that the application requires a camera and should not be shown to users whose devices do not offer a camera.

If you don't want Google Play to filter based on a specific implied feature, you can disable that behavior. To do so, declare the feature explicitly in a `<uses-feature>` element and include an `android:required="false"` attribute. For example, to disable filtering derived from the `CAMERA` permission, you would declare the feature as shown below.

```
<uses-feature android:name="android.hardware.camera" android:required="false" /
```

It's important to understand that the permissions that you request in `<uses-permission>` elements can directly affect how Google Play filters your application. The reference section [Permissions that Imply Feature Requirements](#), below, lists the full set of permissions that imply feature requirements and therefore trigger filtering.

## Special handling for Bluetooth feature

Google Play applies slightly different rules than described above, when determining filtering for Bluetooth.

If an application declares a Bluetooth permission in a `<uses-permission>` element, but does not explicitly declare the Bluetooth feature in a `<uses-feature>` element, Google Play checks the version(s) of the Android platform on which the application is designed to run, as specified in the `<uses-sdk>` element.

As shown in the table below, Google Play enables filtering for the Bluetooth feature only if the application declares its lowest or targeted platform as Android 2.0 (API level 5) or higher. However, note that Google Play applies the normal rules for filtering when the application explicitly declares the Bluetooth feature in a `<uses-feature>` element.

**Table 1.** How Google Play determines the Bluetooth feature requirement for an application that requests a Bluetooth permission but does not declare the Bluetooth feature in a `<uses-feature>` element.

If <code>minSdkVersion</code> or <code>targetSdkVersion</code> is ...	<code>Result</code>
... 5 or higher	Google Play filters for the Bluetooth feature.

<code>&lt;=4 (or uses-sdk is not declared)</code>	<code>&lt;=4</code>	Google Play <i>will not</i> filter the application from any devices based on their reported support for the <code>android.hardware.bluetooth</code> feature.
<code>&lt;=4</code>	<code>&gt;=5</code>	Google Play filters the application from any devices that do not support the <code>android.hardware.bluetooth</code> feature (including older releases).
<code>&gt;=5</code>	<code>&gt;=5</code>	

The examples below illustrate the different filtering effects, based on how Google Play handles the Bluetooth feature.

**In first example, an application that is designed to run on older API levels declares a Bluetooth permission, but does not declare the Bluetooth feature in a `<uses-feature>` element.**

*Result:* Google Play does not filter the application from any device.

```
<manifest ...>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-sdk android:minSdkVersion="3" />
    ...
</manifest>
```

**In the second example, below, the same application also declares a target API level of "5".**

*Result:* Google Play now assumes that the feature is required and will filter the application from all devices that do not report Bluetooth support, including devices running older versions of the platform.

```
<manifest ...>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="5" />
    ...
</manifest>
```

**Here the same application now specifically declares the Bluetooth feature.**

*Result:* Identical to the previous example (filtering is applied).

```
<manifest ...>
    <uses-feature android:name="android.hardware.bluetooth" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="5" />
    ...
</manifest>
```

**Finally, in the case below, the same application adds an `android:required="false"` attribute.**

*Result:* Google Play disables filtering based on Bluetooth feature support, for all devices.

```
<manifest ...>
    <uses-feature android:name="android.hardware.bluetooth" android:required="false" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="5" />
    ...
</manifest>
```

## Testing the features required by your application

You can use the `aapt` tool, included in the Android SDK, to determine how Google Play will filter your application, based on its declared features and permissions. To do so, run `aapt` with the `dump badging` com-

mand. This causes `aapt` to parse your application's manifest and apply the same rules as used by Google Play to determine the features that your application requires.

To use the tool, follow these steps:

1. First, build and export your application as an unsigned `.apk`. If you are developing in Eclipse with ADT, right-click the project and select **Android Tools > Export Unsigned Application Package**. Select a destination filename and path and click **OK**.
2. Next, locate the `aapt` tool, if it is not already in your PATH. If you are using SDK Tools r8 or higher, you can find `aapt` in the `<SDK>/platform-tools/` directory.

**Note:** You must use the version of `aapt` that is provided for the latest Platform-Tools component available. If you do not have the latest Platform-Tools component, download it using the [Android SDK Manager](#).

3. Run `aapt` using this syntax:

```
$ aapt dump badging <path_to_exported_.apk>
```

Here's an example of the command output for the second Bluetooth example, above:

```
$ ./aapt dump badging BTExample.apk
package: name='com.example.android.btexample' versionCode='1' versionName='1'
uses-permission:'android.permission.BLUETOOTH_ADMIN'
uses-feature:'android.hardware.bluetooth'
sdkVersion:'3'
targetSdkVersion:'5'
application: label='BT Example' icon='res/drawable/app_bt_ex.png'
launchable activity name='com.example.android.btexample.MyActivity' label='MyActivity'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'small' 'normal' 'large'
locales: '--_--'
densities: '160'
```

## Features Reference

The tables below provide reference information about hardware and software features and the permissions that can imply them on Google Play.

### Hardware features

The table below describes the hardware feature descriptors supported by the most current platform release. To signal that your application uses or requires a hardware feature, declare each value in a `android:name` attribute in a separate `<uses-feature>` element.

Feature Type	Feature Descriptor	Description
Audio	<code>android.hardware.audio.low_latency</code>	The application uses a low-latency audio pipeline on the device and is sensitive to delays or lag in sound input or output.

Bluetooth	android.hardware.bluetooth	The application uses Bluetooth radio features in the device.
	android.hardware.camera	The application uses the device's camera. If the device supports multiple cameras, the application uses the camera that facing away from the screen.
	android.hardware.camera.autofocus	Subfeature. The application uses the device camera's autofocus capability.
Camera	android.hardware.camera.flash	Subfeature. The application uses the device camera's flash.
	android.hardware.camera.front	Subfeature. The application uses front-facing camera on the device.
	android.hardware.camera.any	The application uses at least one camera facing in any direction. Use this in preference to android.hardware.camera if a back-facing camera is not required.
Location	android.hardware.location	The application uses one or more features on the device for determining location, such as GPS location, network location, or cell location.
	android.hardware.location.network	Subfeature. The application uses coarse location coordinates obtained from a network-based geolocation system supported on the device.
	android.hardware.location.gps	Subfeature. The application uses precise location coordinates obtained from a Global Positioning System receiver on the device.
Microphone	android.hardware.microphone	The application uses a microphone on the device.
NFC	android.hardware.nfc	The application uses Near Field Communications radio features in the device.
Sensors	android.hardware.sensor.accelerometer	The application uses motion readings from an accelerometer on the device.
	android.hardware.sensor.barometer	The application uses the device's barometer.
	android.hardware.sensor.compass	The application uses directional readings from a magnetometer (compass) on the device.
	android.hardware.sensor.gyroscope	The application uses the device's gyroscope sensor.
	android.hardware.sensor.light	The application uses the device's light sensor.

	android.hardware.sensor.proximity	The application uses the device's proximity sensor.
	android.hardware.screen.landscape	The application requires landscape orientation.
Screen	android.hardware.screen.portrait	The application requires portrait orientation.
Telephony	android.hardware.telephony	The application uses telephony features on the device, such as telephony radio with data communication services. Subfeature. The application uses CDMA telephony radio features on the device. Subfeature. The application uses GSM telephony radio features on the device.
Television	android.hardware.type.television	The application is designed for a television user experience.
Touchscreen	android.hardware.faketouch	The application uses basic touch interaction events, such as "click down", "click up", and drag.

`android.hardware.faketouch.multitouch.distinct`

The application performs distinct tracking of two or more "fingers" on a fake touch interface. This is a superset of the faketouch feature.

`android.hardware.faketouch.multitouch.jazzhand`

The application performs distinct tracking of five or more "fingers" on a fake touch interface. This is a superset of the faketouch feature.

`android.hardware.touchscreen`

The application uses touchscreen capabilities for gestures that are more interactive than basic touch events, such as a fling. This is a superset of the basic faketouch feature.

	android.hardware.touchscreen.multitouch	The application uses basic two-point multitouch capabilities on the device screen, such as for pinch gestures, but does not need to track touches independently. This is a superset of touchscreen feature.
	android.hardware.touchscreen.multitouch.distinct	Subfeature. The application uses advanced multipoint multitouch capabilities on the device screen, such as for tracking two or more points fully independently. This is a superset of multitouch feature.
	android.hardware.touchscreen.multitouch.jazzhand	The application uses advanced multipoint multitouch capabilities on the device screen, for tracking up to five points fully independently. This is a superset of distinct multitouch feature.
USB	android.hardware.usb.host	The application uses USB host mode features (behaves as the host and connects to USB devices).
	android.hardware.usb.accessory	The application uses USB accessory features (behaves as the US device and connects to USB hosts).
Wifi	android.hardware.wifi	The application uses 802.11 networking (wifi) features on the device.

## Software features

The table below describes the software feature descriptors supported by the most current platform release. To signal that your application uses or requires a software feature, declare each value in a `android:name` attribute in a separate `<uses-feature>` element.

Feature	Attribute Value	Description	Comments
Live Wallpaper	android.software.live_wallpaper	The application uses or provides Live Wallpapers.	

android.software.sip	The application uses SIP service on the device.
SIP/VOIP android.software.sip.voip	Subfeature. The application uses SIP-based VOIP service on the device. This subfeature implicitly declares the android.software.sip parent feature, unless declared with android:required="false".

## Permissions that Imply Feature Requirements

Some feature constants listed in the tables above were made available to applications *after* the corresponding API; for example, the android.hardware.bluetooth feature was added in Android 2.2 (API level 8), but the bluetooth API that it refers to was added in Android 2.0 (API level 5). Because of this, some apps were able to use the API before they had the ability to declare that they require the API via the <uses-feature> system.

To prevent those apps from being made available unintentionally, Google Play assumes that certain hardware-related permissions indicate that the underlying hardware features are required by default. For instance, applications that use Bluetooth must request the BLUETOOTH permission in a <uses-permission> element — for legacy apps, Google Play assumes that the permission declaration means that the underlying android.hardware.bluetooth feature is required by the application and sets up filtering based on that feature.

The table below lists permissions that imply feature requirements equivalent to those declared in <uses-feature> elements. Note that <uses-feature> declarations, including any declared android:required attribute, always take precedence over features implied by the permissions below.

For any of the permissions below, you can disable filtering based on the implied feature by explicitly declaring the implied feature explicitly, in a <uses-feature> element, with an android:required="false" attribute. For example, to disable any filtering based on the CAMERA permission, you would add this <uses-feature> declaration to the manifest file:

```
<uses-feature android:name="android.hardware.camera" android:required="false" />
```

Category	This Permission...	Implies This Feature Requirement
Bluetooth	BLUETOOTH	android.hardware.bluetooth  (See <a href="#">Special handling for Bluetooth feature</a> for details.)
Camera	BLUETOOTH_ADMIN CAMERA ACCESS_MOCK_LOCATION ACCESS_LOCATION_EXTRA_COMMANDS INSTALL_LOCATION_PROVIDER	android.hardware.bluetooth android.hardware.camera <i>and</i> android.hardware.camera.autofocus android.hardware.location android.hardware.location android.hardware.location.network <i>and</i> android.hardware.location android.hardware.location.gps <i>and</i> android.hardware.location
Location	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION	

Microphone	RECORD_AUDIO	android.hardware.microphone
	CALL_PHONE	android.hardware.telephony
	CALL_PRIVILEGED	android.hardware.telephony
	MODIFY_PHONE_STATE	android.hardware.telephony
	PROCESS_OUTGOING_CALLS	android.hardware.telephony
	READ_SMS	android.hardware.telephony
Telephony	RECEIVE_SMS	android.hardware.telephony
	RECEIVE_MMS	android.hardware.telephony
	RECEIVE_WAP_PUSH	android.hardware.telephony
	SEND_SMS	android.hardware.telephony
	WRITE_APN_SETTINGS	android.hardware.telephony
	WRITE_SMS	android.hardware.telephony
	ACCESS_WIFI_STATE	android.hardware.wifi
Wifi	CHANGE_WIFI_STATE	android.hardware.wifi
	CHANGE_WIFI_MULTICAST_STATE	android.hardware.wifi

# <uses-library>



## Google Play Filtering

Google Play uses the <uses-library> elements declared in your app manifest to filter your app from devices that do not meet its library requirements. For more information about filtering, see the topic [Google Play filters](#).

### syntax:

```
<uses-library  
    android:name="string"  
    android:required=["true" | "false"] />
```

### contained in:

[<application>](#)

### description:

Specifies a shared library that the application must be linked against. This element tells the system to include the library's code in the class loader for the package.

All of the android packages (such as [android.app](#), [android.content](#), [android.view](#), and [android.widget](#)) are in the default library that all applications are automatically linked against. However, some packages (such as maps) are in separate libraries that are not automatically linked. Consult the documentation for the packages you're using to determine which library contains the package code.

This element also affects the installation of the application on a particular device and the availability of the application on Google Play:

### *Installation*

If this element is present and its android:required attribute is set to true, the [PackageManager](#) framework won't let the user install the application unless the library is present on the user's device.

The android:required attribute is described in detail in the following section.

### attributes:

#### **android:name**

The name of the library. The name is provided by the documentation for the package you are using. An example of this is "android.test.runner", a package that contains Android test classes.

#### **android:required**

Boolean value that indicates whether the application requires the library specified by android:name:

- "true": The application does not function without this library. The system will not allow the application on a device that does not have the library.
- "false": The application can use the library if present, but is designed to function without it if necessary. The system will allow the application to be installed, even if the library is not present. If you use "false", you are responsible for checking at runtime that the library is available.

To check for a library, you can use reflection to determine if a particular class is available.

The default is "true".

Introduced in: API Level 7.

**introduced in:**

API Level 1

**see also:**

- [PackageManager](#)

# <uses-permission>



## Google Play Filtering

In some cases, the permissions that you request through <uses-permission> can affect how your application is filtered by Google Play.

If you request a hardware-related permission — CAMERA, for example — Google Play assumes that your application requires the underlying hardware feature and filters the application from devices that do not offer it.

To control filtering, always explicitly declare hardware features in <uses-feature> elements, rather than relying on Google Play to "discover" the requirements in <uses-permission> elements. Then, if you want to disable filtering for a particular feature, you can add a `android:required="false"` attribute to the <uses-feature> declaration.

For a list of permissions that imply hardware features, see the documentation for the [`<uses-feature>`](#) element.

### syntax:

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

### contained in:

[`<manifest>`](#)

### description:

Requests a permission that the application must be granted in order for it to operate correctly. Permissions are granted by the user when the application is installed, not while it's running.

For more information on permissions, see the [Permissions](#) section in the introduction and the separate [Security and Permissions](#) document. A list of permissions defined by the base platform can be found at [`android.Manifest.permission`](#).

### attributes:

#### `android:name`

The name of the permission. It can be a permission defined by the application with the [`<permission>`](#) element, a permission defined by another application, or one of the standard system permissions, such as "android.permission.CAMERA" or "android.permission.READ\_CONTACTS". As these examples show, a permission name typically includes the package name as a prefix.

### introduced in:

API Level 1

### see also:

- [`<permission>`](#)
- [`<uses-feature>`](#)

# In this document

1. [What is API Level?](#)
2. [Uses of API Level in Android](#)
3. [Development Considerations](#)
  1. [Application forward compatibility](#)
  2. [Application backward compatibility](#)
  3. [Selecting a platform version and API Level](#)
  4. [Declaring a minimum API Level](#)
  5. [Testing against higher API Levels](#)
4. [Using a Provisional API Level](#)
5. [Filtering the Reference Documentation by API Level](#)



## Google Play Filtering

Google Play uses the `<uses-sdk>` attributes declared in your app manifest to filter your app from devices that do not meet its platform version requirements. Before setting these attributes, make sure that you understand [Google Play filters](#).

### syntax:

```
<uses-sdk android:minSdkVersion="integer"  
        android:targetSdkVersion="integer"  
        android:maxSdkVersion="integer" />
```

### contained in:

[`<manifest>`](#)

### description:

Lets you express an application's compatibility with one or more versions of the Android platform, by means of an API Level integer. The API Level expressed by an application will be compared to the API Level of a given Android system, which may vary among different Android devices.

Despite its name, this element is used to specify the API Level, *not* the version number of the SDK (software development kit) or Android platform. The API Level is always a single integer. You cannot derive the API Level from its associated Android version number (for example, it is not the same as the major version or the sum of the major and minor versions).

Also read the document about [Versioning Your Applications](#).

### attributes:

`android:minSdkVersion`

An integer designating the minimum API Level required for the application to run. The Android system will prevent the user from installing the application if the system's API Level is lower than the value specified in this attribute. You should always declare this attribute.

**Caution:** If you do not declare this attribute, the system assumes a default value of "1", which indicates that your application is compatible with all versions of Android. If your application is *not* compatible with all versions (for instance, it uses APIs introduced in API Level 3) and you have not declared the proper `minSdkVersion`, then when installed on a system with an API Level less than 3,

the application will crash during runtime when attempting to access the unavailable APIs. For this reason, be certain to declare the appropriate API Level in the `minSdkVersion` attribute.

#### **android:targetSdkVersion**

An integer designating the API Level that the application targets. If not set, the default value equals that given to `minSdkVersion`.

This attribute informs the system that you have tested against the target version and the system should not enable any compatibility behaviors to maintain your app's forward-compatibility with the target version. The application is still able to run on older versions (down to `minSdkVersion`).

As Android evolves with each new version, some behaviors and even appearances might change. However, if the API level of the platform is higher than the version declared by your app's `targetSdkVersion`, the system may enable compatibility behaviors to ensure that your app continues to work the way you expect. You can disable such compatibility behaviors by specifying `targetSdkVersion` to match the API level of the platform on which it's running. For example, setting this value to "11" or higher allows the system to apply a new default theme (Holo) to your app when running on Android 3.0 or higher and also disables [screen compatibility mode](#) when running on larger screens (because support for API level 11 implicitly supports larger screens).

There are many compatibility behaviors that the system may enable based on the value you set for this attribute. Several of these behaviors are described by the corresponding platform versions in the [Build.VERSION\\_CODES](#) reference.

To maintain your application along with each Android release, you should increase the value of this attribute to match the latest API level, then thoroughly test your application on the corresponding platform version.

Introduced in: API Level 4

#### **android:maxSdkVersion**

An integer designating the maximum API Level on which the application is designed to run.

In Android 1.5, 1.6, 2.0, and 2.0.1, the system checks the value of this attribute when installing an application and when re-validating the application after a system update. In either case, if the application's `maxSdkVersion` attribute is lower than the API Level used by the system itself, then the system will not allow the application to be installed. In the case of re-validation after system update, this effectively removes your application from the device.

To illustrate how this attribute can affect your application after system updates, consider the following example:

An application declaring `maxSdkVersion="5"` in its manifest is published on Google Play. A user whose device is running Android 1.6 (API Level 4) downloads and installs the app. After a few weeks, the user receives an over-the-air system update to Android 2.0 (API Level 5). After the update is installed, the system checks the application's `maxSdkVersion` and successfully re-validates it. The application functions as normal. However, some time later, the device receives another system update, this time to Android 2.0.1 (API Level 6). After the update, the system can no longer re-validate the application because the system's own API Level (6) is now higher than the maximum supported by the application (5). The system prevents the application from being visible to the user, in effect removing it from the device.

**Warning:** Declaring this attribute is not recommended. First, there is no need to set the attribute as means of blocking deployment of your application onto new versions of the Android platform as they

are released. By design, new versions of the platform are fully backward-compatible. Your application should work properly on new versions, provided it uses only standard APIs and follows development best practices. Second, note that in some cases, declaring the attribute can **result in your application being removed from users' devices after a system update** to a higher API Level. Most devices on which your application is likely to be installed will receive periodic system updates over the air, so you should consider their effect on your application before setting this attribute.

Introduced in: API Level 4

Future versions of Android (beyond Android 2.0.1) will no longer check or enforce the `maxSdkVersion` attribute during installation or re-validation. Google Play will continue to use the attribute as a filter, however, when presenting users with applications available for download.

**introduced in:**

API Level 1

## What is API Level?

API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.

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

- A core set of packages and classes
- A set of XML elements and attributes for declaring a manifest file
- A set of XML elements and attributes for declaring and accessing resources
- A set of Intents
- A set of permissions that applications can request, as well as permission enforcements included in the system

Each successive version of the Android platform can include updates to the Android application framework API that it delivers.

Updates to the framework API are designed so that the new API remains compatible with earlier versions of the API. That is, most changes in the API are additive and introduce new or replacement functionality. As parts of the API are upgraded, the older replaced parts are deprecated but are not removed, so that existing applications can still use them. In a very small number of cases, parts of the API may be modified or removed, although typically such changes are only needed to ensure API robustness and application or system security. All other API parts from earlier revisions are carried forward without modification.

The framework API that an Android platform delivers is specified using an integer identifier called "API Level". Each Android platform version supports exactly one API Level, although support is implicit for all earlier API Levels (down to API Level 1). The initial release of the Android platform provided API Level 1 and subsequent releases have incremented the API Level.

The table below specifies the API Level supported by each version of the Android platform. For information about the relative numbers of devices that are running each version, see the [Platform Versions dashboards page](#).

Platform Version	API Level	VERSION_CODE	Notes
<a href="#">Android 4.3</a>	<a href="#">18</a>	<a href="#">JELLY_BEAN_MR2</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.2, 4.2.2</a>	<a href="#">17</a>	<a href="#">JELLY_BEAN_MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.1, 4.1.1</a>	<a href="#">16</a>	<a href="#">JELLY_BEAN</a>	<a href="#">Platform Highlights</a>

<a href="#">Android 4.0.3, 4.0.4</a>	<a href="#">15</a>	<a href="#">ICE CREAM SANDWICH MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.0, 4.0.1, 4.0.2</a>	<a href="#">14</a>	<a href="#">ICE CREAM SANDWICH</a>	
<a href="#">Android 3.2</a>	<a href="#">13</a>	<a href="#">HONEYCOMB MR2</a>	
<a href="#">Android 3.1.x</a>	<a href="#">12</a>	<a href="#">HONEYCOMB MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 3.0.x</a>	<a href="#">11</a>	<a href="#">HONEYCOMB</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.4</a>	<a href="#">10</a>	<a href="#">GINGERBREAD MR1</a>	
<a href="#">Android 2.3.3</a>			
<a href="#">Android 2.3.2</a>			<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.1</a>	<a href="#">9</a>	<a href="#">GINGERBREAD</a>	
<a href="#">Android 2.3</a>			
<a href="#">Android 2.2.x</a>	<a href="#">8</a>	<a href="#">FROYO</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.1.x</a>	<a href="#">7</a>	<a href="#">ECLAIR_MR1</a>	
<a href="#">Android 2.0.1</a>	<a href="#">6</a>	<a href="#">ECLAIR_0_1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.0</a>	<a href="#">5</a>	<a href="#">ECLAIR</a>	
<a href="#">Android 1.6</a>	<a href="#">4</a>	<a href="#">DONUT</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.5</a>	<a href="#">3</a>	<a href="#">CUPCAKE</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.1</a>	<a href="#">2</a>	<a href="#">BASE_1_1</a>	
Android 1.0	<a href="#">1</a>	<a href="#">BASE</a>	

## Uses of API Level in Android

The API Level identifier serves a key role in ensuring the best possible experience for users and application developers:

- It lets the Android platform describe the maximum framework API revision that it supports
- It lets applications describe the framework API revision that they require
- It lets the system negotiate the installation of applications on the user's device, such that version-incompatible applications are not installed.

Each Android platform version stores its API Level identifier internally, in the Android system itself.

Applications can use a manifest element provided by the framework API — `<uses-sdk>` — to describe the minimum and maximum API Levels under which they are able to run, as well as the preferred API Level that they are designed to support. The element offers three key attributes:

- `android:minSdkVersion` — Specifies the minimum API Level on which the application is able to run. The default value is "1".
- `android:targetSdkVersion` — Specifies the API Level on which the application is designed to run. In some cases, this allows the application to use manifest elements or behaviors defined in the target API Level, rather than being restricted to using only those defined for the minimum API Level.
- `android:maxSdkVersion` — Specifies the maximum API Level on which the application is able to run. **Important:** Please read the [`<uses-sdk>`](#) documentation before using this attribute.

For example, to specify the minimum system API Level that an application requires in order to run, the application would include in its manifest a `<uses-sdk>` element with a `android:minSdkVersion` attribute. The value of `android:minSdkVersion` would be the integer corresponding to the API Level of the earliest version of the Android platform under which the application can run.

When the user attempts to install an application, or when revalidating an application after a system update, the Android system first checks the `<uses-sdk>` attributes in the application's manifest and compares the values against its own internal API Level. The system allows the installation to begin only if these conditions are met:

- If a `android:minSdkVersion` attribute is declared, its value must be less than or equal to the system's API Level integer. If not declared, the system assumes that the application requires API Level 1.
- If a `android:maxSdkVersion` attribute is declared, its value must be equal to or greater than the system's API Level integer. If not declared, the system assumes that the application has no maximum API Level. Please read the [`<uses-sdk>`](#) documentation for more information about how the system handles this attribute.

When declared in an application's manifest, a `<uses-sdk>` element might look like this:

```
<manifest>
  <uses-sdk android:minSdkVersion="5" />
  ...
</manifest>
```

The principal reason that an application would declare an API Level in `android:minSdkVersion` is to tell the Android system that it is using APIs that were *introduced* in the API Level specified. If the application were to be somehow installed on a platform with a lower API Level, then it would crash at run-time when it tried to access APIs that don't exist. The system prevents such an outcome by not allowing the application to be installed if the lowest API Level it requires is higher than that of the platform version on the target device.

For example, the [`android.appwidget`](#) package was introduced with API Level 3. If an application uses that API, it must declare a `android:minSdkVersion` attribute with a value of "3". The application will then be installable on platforms such as Android 1.5 (API Level 3) and Android 1.6 (API Level 4), but not on the Android 1.1 (API Level 2) and Android 1.0 platforms (API Level 1).

For more information about how to specify an application's API Level requirements, see the [`<uses-sdk>`](#) section of the manifest file documentation.

## Development Considerations

The sections below provide information related to API level that you should consider when developing your application.

### Application forward compatibility

Android applications are generally forward-compatible with new versions of the Android platform.

Because almost all changes to the framework API are additive, an Android application developed using any given version of the API (as specified by its API Level) is forward-compatible with later versions of the Android platform and higher API levels. The application should be able to run on all later versions of the Android platform, except in isolated cases where the application uses a part of the API that is later removed for some reason.

Forward compatibility is important because many Android-powered devices receive over-the-air (OTA) system updates. The user may install your application and use it successfully, then later receive an OTA update to a new version of the Android platform. Once the update is installed, your application will run in a new run-time version of the environment, but one that has the API and system capabilities that your application depends on.

In some cases, changes *below* the API, such those in the underlying system itself, may affect your application when it is run in the new environment. For that reason it's important for you, as the application developer, to understand how the application will look and behave in each system environment. To help you test your application on various versions of the Android platform, the Android SDK includes multiple platforms that you can

download. Each platform includes a compatible system image that you can run in an AVD, to test your application.

## Application backward compatibility

Android applications are not necessarily backward compatible with versions of the Android platform older than the version against which they were compiled.

Each new version of the Android platform can include new framework APIs, such as those that give applications access to new platform capabilities or replace existing API parts. The new APIs are accessible to applications when running on the new platform and, as mentioned above, also when running on later versions of the platform, as specified by API Level. Conversely, because earlier versions of the platform do not include the new APIs, applications that use the new APIs are unable to run on those platforms.

Although it's unlikely that an Android-powered device would be downgraded to a previous version of the platform, it's important to realize that there are likely to be many devices in the field that run earlier versions of the platform. Even among devices that receive OTA updates, some might lag and might not receive an update for a significant amount of time.

## Selecting a platform version and API Level

When you are developing your application, you will need to choose the platform version against which you will compile the application. In general, you should compile your application against the lowest possible version of the platform that your application can support.

You can determine the lowest possible platform version by compiling the application against successively lower build targets. After you determine the lowest version, you should create an AVD using the corresponding platform version (and API Level) and fully test your application. Make sure to declare a `android:minSdkVersion` attribute in the application's manifest and set its value to the API Level of the platform version.

## Declaring a minimum API Level

If you build an application that uses APIs or system features introduced in the latest platform version, you should set the `android:minSdkVersion` attribute to the API Level of the latest platform version. This ensures that users will only be able to install your application if their devices are running a compatible version of the Android platform. In turn, this ensures that your application can function properly on their devices.

If your application uses APIs introduced in the latest platform version but does *not* declare a `android:minSdkVersion` attribute, then it will run properly on devices running the latest version of the platform, but *not* on devices running earlier versions of the platform. In the latter case, the application will crash at runtime when it tries to use APIs that don't exist on the earlier versions.

## Testing against higher API Levels

After compiling your application, you should make sure to test it on the platform specified in the application's `android:minSdkVersion` attribute. To do so, create an AVD that uses the platform version required by your application. Additionally, to ensure forward-compatibility, you should run and test the application on all platforms that use a higher API Level than that used by your application.

The Android SDK includes multiple platform versions that you can use, including the latest version, and provides an updater tool that you can use to download other platform versions as necessary.

To access the updater, use the `android` command-line tool, located in the `<sdk>/tools` directory. You can launch the SDK updater by executing `android sdk`. You can also simply double-click the `android.bat` (Windows) or `android` (OS X/Linux) file. In ADT, you can also access the updater by selecting **Window > Android SDK Manager**.

To run your application against different platform versions in the emulator, create an AVD for each platform version that you want to test. For more information about AVDs, see [Creating and Managing Virtual Devices](#). If you are using a physical device for testing, ensure that you know the API Level of the Android platform it runs. See the table at the top of this document for a list of platform versions and their API Levels.

## Using a Provisional API Level

In some cases, an "Early Look" Android SDK platform may be available. To let you begin developing on the platform although the APIs may not be final, the platform's API Level integer will not be specified. You must instead use the platform's *provisional API Level* in your application manifest, in order to build applications against the platform. A provisional API Level is not an integer, but a string matching the codename of the unreleased platform version. The provisional API Level will be specified in the release notes for the Early Look SDK release notes and is case-sensitive.

The use of a provisional API Level is designed to protect developers and device users from inadvertently publishing or installing applications based on the Early Look framework API, which may not run properly on actual devices running the final system image.

The provisional API Level will only be valid while using the Early Look SDK and can only be used to run applications in the emulator. An application using the provisional API Level can never be installed on an Android device. At the final release of the platform, you must replace any instances of the provisional API Level in your application manifest with the final platform's actual API Level integer.

## Filtering the Reference Documentation by API Level

Reference documentation pages on the Android Developers site offer a "Filter by API Level" control in the top-right area of each page. You can use the control to show documentation only for parts of the API that are actually accessible to your application, based on the API Level that it specifies in the `android:minSdkVersion` attribute of its manifest file.

To use filtering, select the checkbox to enable filtering, just below the page search box. Then set the "Filter by API Level" control to the same API Level as specified by your application. Notice that APIs introduced in a later API Level are then grayed out and their content is masked, since they would not be accessible to your application.

Filtering by API Level in the documentation does not provide a view of what is new or introduced in each API Level — it simply provides a way to view the entire API associated with a given API Level, while excluding API elements introduced in later API Levels.

If you decide that you don't want to filter the API documentation, just disable the feature using the checkbox. By default, API Level filtering is disabled, so that you can view the full framework API, regardless of API Level.

Also note that the reference documentation for individual API elements specifies the API Level at which each element was introduced. The API Level for packages and classes is specified as "Since <api level>" at the top-right corner of the content area on each documentation page. The API Level for class members is specified in their detailed description headers, at the right margin.



# User Interface

Your app's user interface is everything that the user can see and interact with. Android provides a variety of pre-build UI components such as structured layout objects and UI controls that allow you to build the graphical user interface for your app. Android also provides other UI modules for special interfaces such as dialogs, notifications, and menus.

## Blog Articles

### Say Goodbye to the Menu Button

As Ice Cream Sandwich rolls out to more devices, it's important that you begin to migrate your designs to the action bar in order to promote a consistent Android user experience.

### New Layout Widgets: Space and GridLayout

Ice Cream Sandwich (ICS) sports two new widgets that have been designed to support the richer user interfaces made possible by larger displays: Space and GridLayout.

### Customizing the Action Bar

By using the Action Bar in your Honeycomb-targeted apps, you'll give your users a familiar way to interact with your application.

### Horizontal View Swiping with ViewPager

Whether you have just started out in Android app development or are a veteran of the craft, it probably won't be too long before you'll need to implement horizontally scrolling sets of views.

# **Training**

## **Implementing Effective Navigation**

This class shows you how to plan out the high-level screen hierarchy for your application and then choose appropriate forms of navigation to allow users to effectively and intuitively traverse your content.

## **Designing for Multiple Screens**

Android powers hundreds of device types with several different screen sizes, ranging from small phones to large TV sets. This class shows you how to implement a user interface that's optimized for several screen configurations.

## **Improving Layout Performance**

Layouts are a key part of Android applications that directly affect the user experience. If implemented poorly, your layout can lead to a memory hungry application with slow UIs. This class shows you how to avoid such problems.

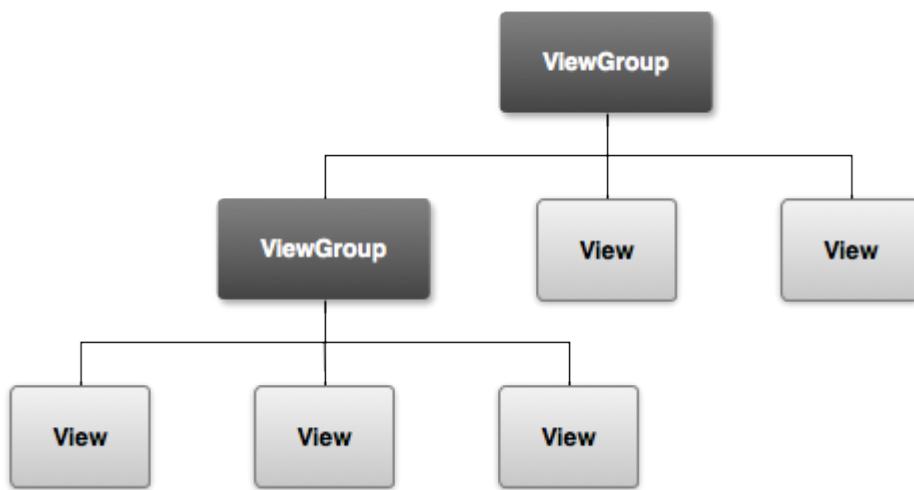
# UI Overview

All user interface elements in an Android app are built using [View](#) and [ViewGroup](#) objects. A [View](#) is an object that draws something on the screen that the user can interact with. A [ViewGroup](#) is an object that holds other [View](#) (and [ViewGroup](#)) objects in order to define the layout of the interface.

Android provides a collection of both [View](#) and [ViewGroup](#) subclasses that offer you common input controls (such as buttons and text fields) and various layout models (such as a linear or relative layout).

## User Interface Layout

The user interface for each component of your app is defined using a hierarchy of [View](#) and [ViewGroup](#) objects, as shown in figure 1. Each view group is an invisible container that organizes child views, while the child views may be input controls or other widgets that draw some part of the UI. This hierarchy tree can be as simple or complex as you need it to be (but simplicity is best for performance).



**Figure 1.** Illustration of a view hierarchy, which defines a UI layout.

To declare your layout, you can instantiate [View](#) objects in code and start building a tree, but the easiest and most effective way to define your layout is with an XML file. XML offers a human-readable structure for the layout, similar to HTML.

The name of an XML element for a view is respective to the Android class it represents. So a `<TextView>` element creates a [TextView](#) widget in your UI, and a `<LinearLayout>` element creates a [LinearLayout](#) view group.

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>
```

When you load a layout resource in your app, Android initializes each node of the layout into a runtime object you can use to define additional behaviors, query the object state, or modify the layout.

For a complete guide to creating a UI layout, see [XML Layouts](#).

## User Interface Components

You don't have to build all of your UI using [View](#) and [ViewGroup](#) objects. Android provides several app components that offer a standard UI layout for which you simply need to define the content. These UI components each have a unique set of APIs that are described in their respective documents, such as [ActionBar](#), [Dialogs](#), and [Status Notifications](#).

# Layouts

## In this document

1. [Write the XML](#)
2. [Load the XML Resource](#)
3. [Attributes](#)
  1. [ID](#)
  2. [Layout Parameters](#)
4. [Layout Position](#)
5. [Size, Padding and Margins](#)
6. [Common Layouts](#)
7. [Building Layouts with an Adapter](#)
  1. [Filling an adapter view with data](#)
  2. [Handling click events](#)

## Key classes

1. [View](#)
2. [ViewGroup](#)
3. [ViewGroup.LayoutParams](#)

## See also

1. [Building a Simple User Interface](#)

A layout defines the visual structure for a user interface, such as the UI for an [activity](#) or [app widget](#). You can declare a layout in two ways:

- **Declare UI elements in XML.** Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.
- **Instantiate layout elements at runtime.** Your application can create View and ViewGroup objects (and manipulate their properties) programmatically.

The Android framework gives you the flexibility to use either or both of these methods for declaring and managing your application's UI. For example, you could declare your application's default layouts in XML, including the screen elements that will appear in them and their properties. You could then add code in your application that would modify the state of the screen objects, including those declared in XML, at run time.

- The [ADT Plugin for Eclipse](#) offers a layout preview of your XML — with the XML file opened, select the **Layout** tab.
- You should also try the [Hierarchy Viewer](#) tool, for debugging layouts — it reveals layout property values, draws wireframes with padding/margin indicators, and full rendered views while you debug on the emulator or device.
- The [layoutopt](#) tool lets you quickly analyze your layouts and hierarchies for inefficiencies or other problems.

The advantage to declaring your UI in XML is that it enables you to better separate the presentation of your application from the code that controls its behavior. Your UI descriptions are external to your application code, which means that you can modify or adapt it without having to modify your source code and recompile. For example, you can create XML layouts for different screen orientations, different device screen sizes, and different

languages. Additionally, declaring the layout in XML makes it easier to visualize the structure of your UI, so it's easier to debug problems. As such, this document focuses on teaching you how to declare your layout in XML. If you're interested in instantiating View objects at runtime, refer to the [ViewGroup](#) and [View](#) class references.

In general, the XML vocabulary for declaring UI elements closely follows the structure and naming of the classes and methods, where element names correspond to class names and attribute names correspond to methods. In fact, the correspondence is often so direct that you can guess what XML attribute corresponds to a class method, or guess what class corresponds to a given xml element. However, note that not all vocabulary is identical. In some cases, there are slight naming differences. For example, the `EditText` element has a `text` attribute that corresponds to `EditText.setText()`.

**Tip:** Learn more about different layout types in [Common Layout Objects](#). There are also a collection of tutorials on building various layouts in the [Hello Views](#) tutorial guide.

## Write the XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a `View` or `ViewGroup` object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a `View` hierarchy that defines your layout. For example, here's an XML layout that uses a vertical [LinearLayout](#) to hold a [TextView](#) and a [Button](#):

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

After you've declared your layout in XML, save the file with the `.xml` extension, in your Android project's `res/layout/` directory, so it will properly compile.

More information about the syntax for a layout XML file is available in the [Layout Resources](#) document.

## Load the XML Resource

When you compile your application, each XML layout file is compiled into a [View](#) resource. You should load the layout resource from your application code, in your [Activity.onCreate\(\)](#) callback implementation. Do so by calling [setContentView\(\)](#), passing it the reference to your layout resource in the form of: `R.layout.layout_file_name` For example, if your XML layout is saved as `main_layout.xml`, you would load it for your Activity like so:

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

The `onCreate()` callback method in your Activity is called by the Android framework when your Activity is launched (see the discussion about lifecycles, in the [Activities](#) document).

## Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, `TextView` supports the `textSize` attribute), but these attributes are also inherited by any View objects that may extend this class. Some are common to all View objects, because they are inherited from the root View class (like the `id` attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

### ID

Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the application is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the `id` attribute. This is an XML attribute common to all View objects (defined by the [View](#) class) and you will use it very often. The syntax for an ID, inside an XML tag is:

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

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 (in the `R.java` file). There are a number of other ID resources that are offered by the Android framework. When referencing an Android resource ID, you do not need the plus-symbol, but must add the `android` package namespace, like so:

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

With the `android` package namespace in place, we're now referencing an ID from the `android.R` resources class, rather than the local resources class.

In order to create views and reference them from the application, a common pattern is to:

1. Define a view/widget in the layout file and assign it a unique ID:

```
<Button android:id="@+id/my_button"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="@string/my_button_text"/>
```

2. Then create an instance of the view object and capture it from the layout (typically in the [onCreate\(\)](#) method):

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

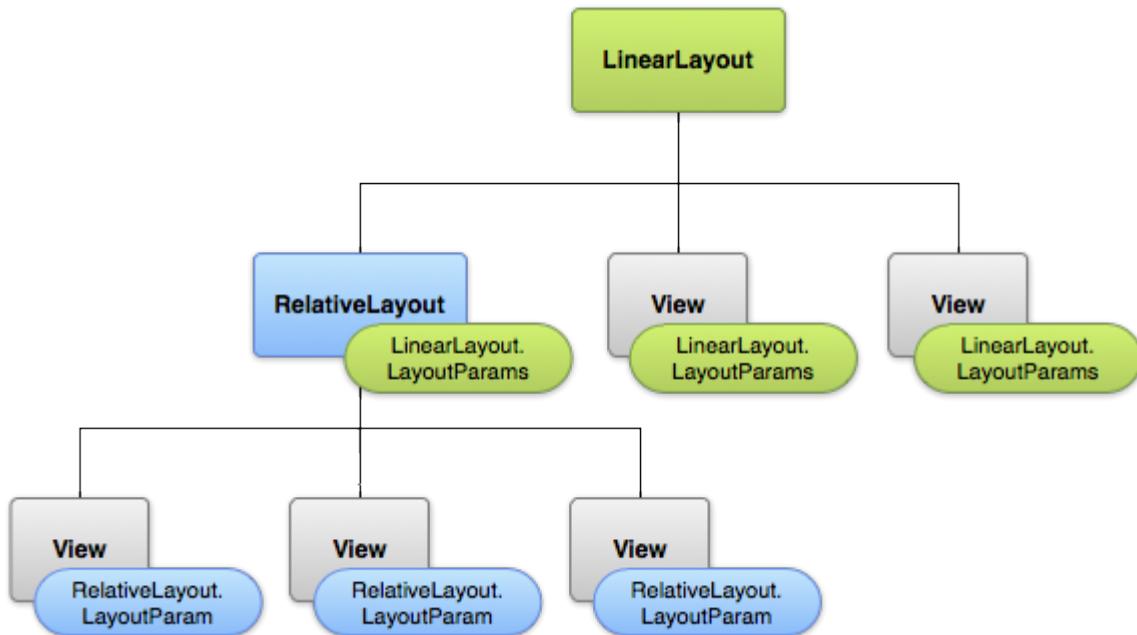
Defining IDs for view objects is important when creating a [RelativeLayout](#). In a relative layout, sibling views can define their layout relative to another sibling view, which is referenced by the unique ID.

An ID need not be unique throughout the entire tree, but it should be unique within the part of the tree you are searching (which may often be the entire tree, so it's best to be completely unique when possible).

## Layout Parameters

XML layout attributes named `layout_something` define layout parameters for the View that are appropriate for the ViewGroup in which it resides.

Every ViewGroup class implements a nested class that extends [ViewGroup.LayoutParams](#). This subclass contains property types that define the size and position for each child view, as appropriate for the view group. As you can see in figure 1, the parent view group defines layout parameters for each child view (including the child view group).



**Figure 1.** Visualization of a view hierarchy with layout parameters associated with each view.

Note that every `LayoutParams` subclass has its own syntax for setting values. Each child element must define `LayoutParams` that are appropriate for its parent, though it may also define different `LayoutParams` for its own children.

All view groups include a width and height (`layout_width` and `layout_height`), and each view is required to define them. Many LayoutParams also include optional margins and borders.

You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will use one of these constants to set the width or height:

- *wrap\_content* tells your view to size itself to the dimensions required by its content
  - *fill\_parent* (renamed *match\_parent* in API Level 8) tells your view to become as big as its parent view group will allow.

In general, specifying a layout width and height using absolute units such as pixels is not recommended. Instead, using relative measurements such as density-independent pixel units (*dp*), *wrap\_content*, or *fill\_parent*, is a better approach, because it helps ensure that your application will display properly across a variety of device screen sizes. The accepted measurement types are defined in the [Available Resources](#) document.

# Layout Position

The geometry of a view is that of a rectangle. A view has a location, expressed as a pair of *left* and *top* coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the pixel.

It is possible to retrieve the location of a view by invoking the methods [getLeft\(\)](#) and [getTop\(\)](#). The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view. These methods both return the location of the view relative to its parent. For instance, when getLeft() returns 20, that means the view is located 20 pixels to the right of the left edge of its direct parent.

In addition, several convenience methods are offered to avoid unnecessary computations, namely [getRight\(\)](#) and [getBottom\(\)](#). These methods return the coordinates of the right and bottom edges of the rectangle representing the view. For instance, calling [getRight\(\)](#) is similar to the following computation: `getLeft() + getWidth()`.

## Size, Padding and Margins

The size of a view is expressed with a width and a height. A view actually possess two pairs of width and height values.

The first pair is known as *measured width* and *measured height*. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling [getMeasuredWidth\(\)](#) and [getMeasuredHeight\(\)](#).

The second pair is simply known as *width* and *height*, or sometimes *drawing width* and *drawing height*. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling [getWidth\(\)](#) and [getHeight\(\)](#).

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific amount of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the [setPadding\(int, int, int, int\)](#) method and queried by calling [getPaddingLeft\(\)](#), [getPaddingTop\(\)](#), [getPaddingRight\(\)](#) and [getPaddingBottom\(\)](#).

Even though a view can define a padding, it does not provide any support for margins. However, view groups provide such a support. Refer to [ViewGroup](#) and [ViewGroup.MarginLayoutParams](#) for further information.

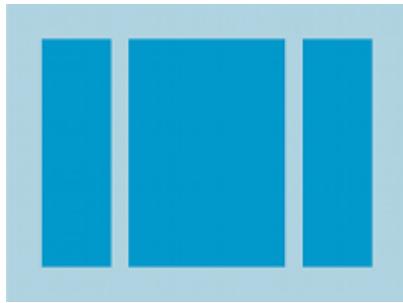
For more information about dimensions, see [Dimension Values](#).

## Common Layouts

Each subclass of the [ViewGroup](#) class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.

**Note:** Although you can nest one or more layouts within another layout to achieve your UI design, you should strive to keep your layout hierarchy as shallow as possible. Your layout draws faster if it has fewer nested layouts (a wide view hierarchy is better than a deep view hierarchy).

## Linear Layout



A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.

## Relative Layout



Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

## Web View



Displays web pages.

# Building Layouts with an Adapter

When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses [AdapterView](#) to populate the layout with views at runtime. A subclass of the [AdapterView](#) class uses an [Adapter](#) to bind data to its layout. The [Adapter](#) behaves as a middle-man between the data source and the [AdapterView](#) layout—the [Adapter](#) retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the [AdapterView](#) layout.

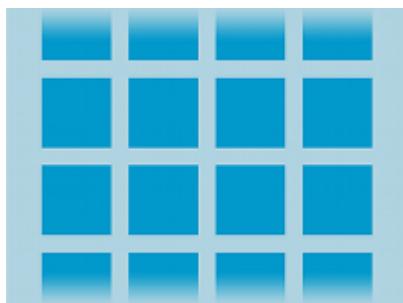
Common layouts backed by an adapter include:

## List View



Displays a scrolling single column list.

## Grid View



Displays a scrolling grid of columns and rows.

## Filling an adapter view with data

You can populate an [AdapterView](#) such as [ListView](#) or [GridView](#) by binding the [AdapterView](#) instance 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](#). The two most common adapters are:

### ArrayAdapter

Use this adapter when your data source is an array. By default, [ArrayAdapter](#) creates a view for each array item by calling [toString\(\)](#) on each item and placing the contents in a [TextView](#).

For example, if you have an array of strings you want to display in a [ListView](#), initialize a new  [ArrayAdapter](#) using a constructor to specify the layout for each string and the string array:

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

The arguments for this constructor are:

- Your app [Context](#)
- The layout that contains a [TextView](#) for each string in the array
- The string array

Then simply call [setAdapter\(\)](#) on your [ListView](#):

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

To customize the appearance of each item you can override the [`toString\(\)`](#) method for the objects in your array. Or, to create a view for each item that's something other than a [`TextView`](#) (for example, if you want an [`ImageView`](#) for each array item), extend the [`ArrayAdapter`](#) class and override [`getView\(\)`](#) to return the type of view you want for each item.

### [SimpleCursorAdapter](#)

Use this adapter when your data comes from a [`Cursor`](#). When using [`SimpleCursorAdapter`](#), you must specify a layout to use for each row in the [`Cursor`](#) and which columns in the [`Cursor`](#) should be inserted into which views of the layout. For example, if you want to create a list of people's names and phone numbers, you can perform a query that returns a [`Cursor`](#) containing a row for each person and columns for the names and numbers. You then create a string array specifying which columns from the [`Cursor`](#) you want in the layout for each result and an integer array specifying the corresponding views that each column should be placed:

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

When you instantiate the [`SimpleCursorAdapter`](#), pass the layout to use for each result, the [`Cursor`](#) containing the results, and these two arrays:

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

The [`SimpleCursorAdapter`](#) then creates a view for each row in the [`Cursor`](#) using the provided layout by inserting each `fromColumns` item into the corresponding `toViews` view.

If, during the course of your application's life, you change the underlying data that is read by your adapter, you should call [`notifyDataSetChanged\(\)`](#). This will notify the attached view that the data has been changed and it should refresh itself.

## Handling click events

You can respond to click events on each item in an [`AdapterView`](#) by implementing the [`AdapterView.OnItemClickListener`](#) interface. For example:

```
// Create a message handling object as an anonymous class.
private OnItemClickListener mMessageClickedHandler = new OnItemClickListener()
    public void onItemClick(AdapterView parent, View v, int position, long id)
        // Do something in response to the click
    }
};

listView.setOnItemClickListener(mMessageClickedHandler);
```

# Linear Layout

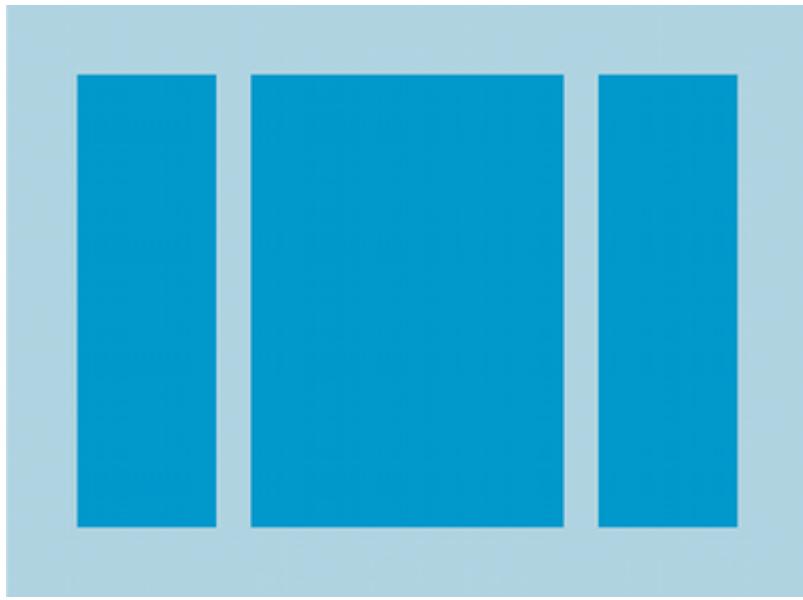
## In this document

1. [Layout Weight](#)
2. [Example](#)

## Key classes

1. [LinearLayout](#)
2. [LinearLayout.LayoutParams](#)

[LinearLayout](#) is a view group that aligns all children in a single direction, vertically or horizontally. You can specify the layout direction with the [android:orientation](#) attribute.



All children of a [LinearLayout](#) are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding). A [LinearLayout](#) respects *margins* between children and the *gravity* (right, center, or left alignment) of each child.

## Layout Weight

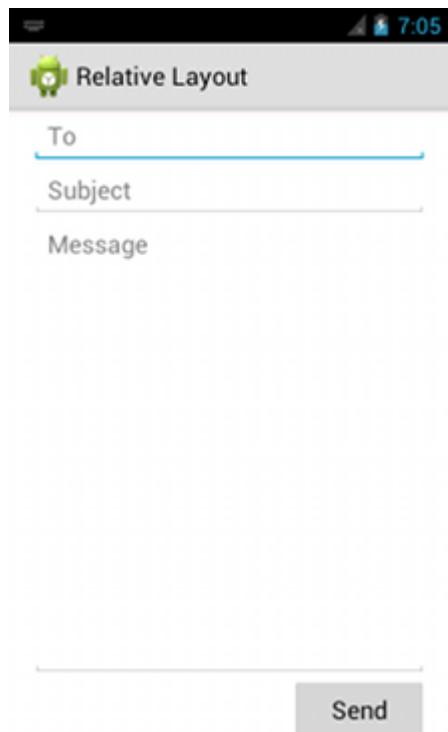
### Equally weighted children

To create a linear layout in which each child uses the same amount of space on the screen, set the [android:layout\\_height](#) of each view to "0dp" (for a vertical layout) or the [android:layout\\_width](#) of each view to "0dp" (for a horizontal layout). Then set the [android:layout\\_weight](#) of each view to "1".

[LinearLayout](#) also supports assigning a *weight* to individual children with the [android:layout\\_weight](#) attribute. This attribute assigns an "importance" value to a view in terms of how much space it should occupy on the screen. A larger weight value allows it to expand to fill any remaining space in the parent view. Child views can specify a weight value, and then any remaining space in the view group is assigned to children in the proportion of their declared weight. Default weight is zero.

For example, if there are three text fields and two of them declare a weight of 1, while the other is given no weight, the third text field without weight will not grow and will only occupy the area required by its content. The other two will expand equally to fill the space remaining after all three fields are measured. If the third field is then given a weight of 2 (instead of 0), then it is now declared more important than both the others, so it gets half the total remaining space, while the first two share the rest equally.

## Example



```
<?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:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:orientation="vertical" >
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/to" />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/subject" />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/message" />
    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right" />
```

```
    android:text="@string/send" />
</LinearLayout>
```

For details about the attributes available to each child view of a [LinearLayout](#), see [LinearLayout.LayoutParams](#).

# Relative Layout

## In this document

1. [Positioning Views](#)
2. [Example](#)

## Key classes

1. [RelativeLayout](#)
2. [RelativeLayout.LayoutParams](#)

[RelativeLayout](#) is a view group that displays child views in relative positions. The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent [RelativeLayout](#) area (such as aligned to the bottom, left of center).



A [RelativeLayout](#) is a very powerful utility for designing a user interface because it can eliminate nested view groups and keep your layout hierarchy flat, which improves performance. If you find yourself using several nested [LinearLayout](#) groups, you may be able to replace them with a single [RelativeLayout](#).

## Positioning Views

[RelativeLayout](#) lets child views specify their position relative to the parent view or to each other (specified by ID). So you can align two elements by right border, or make one below another, centered in the screen, centered left, and so on. By default, all child views are drawn at the top-left of the layout, so you must define the position of each view using the various layout properties available from [RelativeLayout.LayoutParams](#).

Some of the many layout properties available to views in a [RelativeLayout](#) include:

### `android:layout_alignParentTop`

If "true", makes the top edge of this view match the top edge of the parent.

### [android:layout\\_centerVertical](#)

If "true", centers this child vertically within its parent.

### [android:layout\\_below](#)

Positions the top edge of this view below the view specified with a resource ID.

### [android:layout\\_toRightOf](#)

Positions the left edge of this view to the right of the view specified with a resource ID.

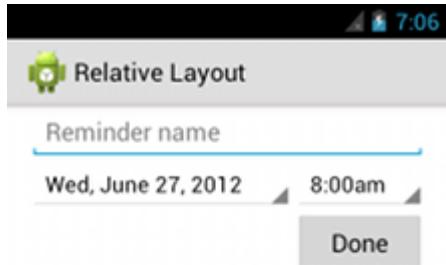
These are just a few examples. All layout attributes are documented at [RelativeLayout.LayoutParams](#).

The value for each layout property is either a boolean to enable a layout position relative to the parent [RelativeLayout](#) or an ID that references another view in the layout against which the view should be positioned.

In your XML layout, dependencies against other views in the layout can be declared in any order. For example, you can declare that "view1" be positioned below "view2" even if "view2" is the last view declared in the hierarchy. The example below demonstrates such a scenario.

## Example

Each of the attributes that control the relative position of each view are emphasized.



```
<?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="fill_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
    <EditText
        android:id="@+id/name"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
```

```
<Spinner
    android:id="@+id/dates"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_below="@id/name"
    android:layout_alignParentLeft="true"
    android:layout_toLeftOf="@+id/times" />
<Spinner
    android:id="@+id/times"
    android:layout_width="96dp"
    android:layout_height="wrap_content"
    android:layout_below="@id/name"
    android:layout_alignParentRight="true" />
<Button
    android:layout_width="96dp"
    android:layout_height="wrap_content"
    android:layout_below="@+id/times"
    android:layout_alignParentRight="true"
    android:text="@string/done" />
</RelativeLayout>
```

For details about all the layout attributes available to each child view of a [RelativeLayout](#), see [RelativeLayout.LayoutParams](#).

# Table

## In this document

1. [Example](#)

## Key classes

1. [TableLayout](#)
2. [TableRow](#)
3. [TextView](#)

[TableLayout](#) is a [ViewGroup](#) that displays child [View](#) elements in rows and columns.



[TableLayout](#) positions its children into rows and columns. TableLayout containers do not display border lines for their rows, columns, or cells. The table will have as many columns as the row with the most cells. A table can leave cells empty, but cells cannot span columns, as they can in HTML.

[TableRow](#) objects are the child views of a TableLayout (each TableRow defines a single row in the table). Each row has zero or more cells, each of which is defined by any kind of other View. So, the cells of a row may be composed of a variety of View objects, like ImageView or TextView objects. A cell may also be a ViewGroup object (for example, you can nest another TableLayout as a cell).

The following sample layout has two rows and two cells in each. The accompanying screenshot shows the result, with cell borders displayed as dotted lines (added for visual effect).

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1">
    <TableRow>
        <TextView
            android:text="@string/table_layout_4_open"
            android:padding="3dip" />
        <TextView
            android:text="@string/table_layout_4_open_shortcut"
            android:gravity="right"
            android:padding="3dip" />
    </TableRow>

    <TableRow>
        <TextView
            android:text="@string/table_layout_4_save"
            android:padding="3dip" />
        <TextView
            android:text="@string/table_layout_4_save_shortcut"
            android:gravity="right"
            android:padding="3dip" />
    </TableRow>
</TableLayout>
```

Views/Lay...  
Open...  
Save As...  
.....

Columns can be hidden, marked to stretch and fill the available screen space, or can be marked as shrinkable to force the column to shrink until the table fits the screen. See the [TableLayout reference](#) documentation for more details.

## Example

1. Start a new project named *HelloTableLayout*.
2. Open the `res/layout/main.xml` file and insert the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1">

    <TableRow>
        <TextView
            android:layout_column="1"
            android:text="Open..."
            android:padding="3dip" />
        <TextView
            android:text="Ctrl-O"
            android:gravity="right"
            android:padding="3dip" />
    </TableRow>
```

```
<TableRow>
    <TextView
        android:layout_column="1"
        android:text="Save..."
        android:padding="3dip" />
    <TextView
        android:text="Ctrl-S"
        android:gravity="right"
        android:padding="3dip" />
</TableRow>

<TableRow>
    <TextView
        android:layout_column="1"
        android:text="Save As..."
        android:padding="3dip" />
    <TextView
        android:text="Ctrl-Shift-S"
        android:gravity="right"
        android:padding="3dip" />
</TableRow>

<View
    android:layout_height="2dip"
    android:background="#FF909090" />

<TableRow>
    <TextView
        android:text="X"
        android:padding="3dip" />
    <TextView
        android:text="Import..."
        android:padding="3dip" />
</TableRow>

<TableRow>
    <TextView
        android:text="X"
        android:padding="3dip" />
    <TextView
        android:text="Export..."
        android:padding="3dip" />
    <TextView
        android:text="Ctrl-E"
        android:gravity="right"
        android:padding="3dip" />
</TableRow>

<View
    android:layout_height="2dip"
    android:background="#FF909090" />

<TableRow>
```

```
<TextView  
    android:layout_column="1"  
    android:text="Quit"  
    android:padding="3dip" />  
</TableRow>  
</TableLayout>
```

Notice how this resembles the structure of an HTML table. The [TableLayout](#) element is like the HTML `<table>` element; [TableRow](#) is like a `><tr>>` element; but for the cells, you can use any kind of [View](#) element. In this example, a [TextView](#) is used for each cell. In between some of the rows, there is also a basic [View](#), which is used to draw a horizontal line.

3. Make sure your *HelloTableLayout* Activity loads this layout in the [onCreate\(\)](#) method:

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

The [setContentView\(int\)](#) method loads the layout file for the [Activity](#), specified by the resource ID — `R.layout.main` refers to the `res/layout/main.xml` layout file.

4. Run the application.

You should see the following:

# List View

## In this document

1. [Using a Loader](#)
2. [Example](#)

## Key classes

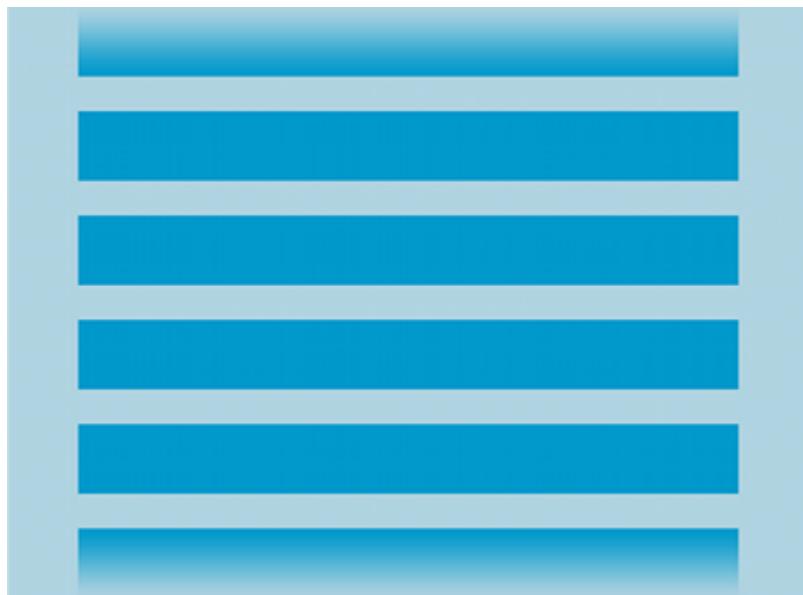
1. [ListView](#)
2. [Adapter](#)
3. [CursorLoader](#)

## See also

1. [Loaders](#)

[ListView](#) is a view group that displays a list of scrollable items. The list items are automatically inserted to the list using an [Adapter](#) that pulls content from a source such as an array or database query and converts each item result into a view that's placed into the list.

For an introduction to how you can dynamically insert views using an adapter, read [Building Layouts with an Adapter](#).



## Using a Loader

Using a [CursorLoader](#) is the standard way to query a [Cursor](#) as an asynchronous task in order to avoid blocking your app's main thread with the query. When the [CursorLoader](#) receives the [Cursor](#) result, the [LoaderCallbacks](#) receives a callback to [onLoadFinished\(\)](#), which is where you update your [Adapter](#) with the new [Cursor](#) and the list view then displays the results.

Although the [CursorLoader](#) APIs were first introduced in Android 3.0 (API level 11), they are also available in the [Support Library](#) so that your app may use them while supporting devices running Android 1.6 or higher.

For more information about using a [Loader](#) to asynchronously load data, see the [Loaders](#) guide.

## Example

The following example uses [ListActivity](#), which is an activity that includes a [ListView](#) as its only layout element by default. It performs a query to the [Contacts Provider](#) for a list of names and phone numbers.

The activity implements the [LoaderCallbacks](#) interface in order to use a [CursorLoader](#) that dynamically loads the data for the list view.

```
public class ListViewLoader extends ListActivity
    implements LoaderManager.LoaderCallbacks<Cursor> {

    // This is the Adapter being used to display the list's data
    SimpleCursorAdapter mAdapter;

    // These are the Contacts rows that we will retrieve
    static final String[] PROJECTION = new String[] {ContactsContract.Data._ID,
        ContactsContract.Data.DISPLAY_NAME};

    // This is the select criteria
    static final String SELECTION = "(((" +
        ContactsContract.Data.DISPLAY_NAME + " NOTNULL) AND (" +
        ContactsContract.Data.DISPLAY_NAME + " != '' ))";

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

        // Create a progress bar to display while the list loads
        ProgressBar progressBar = new ProgressBar(this);
        progressBar.setLayoutParams(new LayoutParams(LayoutParams.WRAP_CONTENT,
            LayoutParams.WRAP_CONTENT, Gravity.CENTER));
        progressBar.setIndeterminate(true);
        getListView().setEmptyView(progressBar);

        // Must add the progress bar to the root of the layout
        ViewGroup root = (ViewGroup) findViewById(android.R.id.content);
        root.addView(progressBar);

        // For the cursor adapter, specify which columns go into which views
        String[] fromColumns = {ContactsContract.Data.DISPLAY_NAME};
        int[] toViews = {android.R.id.text1}; // The TextView in simple_list_it

        // Create an empty adapter we will use to display the loaded data.
        // We pass null for the cursor, then update it in onLoadFinished()
        mAdapter = new SimpleCursorAdapter(this,
            android.R.layout.simple_list_item_1, null,
            fromColumns, toViews, 0);
        setListAdapter(mAdapter);

        // Prepare the loader. Either re-connect with an existing one,
        // or start a new one.
```

```

        getLoaderManager().initLoader(0, null, this);
    }

    // Called when a new Loader needs to be created
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        // Now create and return a CursorLoader that will take care of
        // creating a Cursor for the data being displayed.
        return new CursorLoader(this, ContactsContract.Data.CONTENT_URI,
                               PROJECTION, SELECTION, null, null);
    }

    // Called when a previously created loader has finished loading
    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        // Swap the new cursor in.  (The framework will take care of closing the
        // old cursor once we return.)
        mAdapter.swapCursor(data);
    }

    // Called when a previously created loader is reset, making the data unavail-
    public void onLoaderReset(Loader<Cursor> loader) {
        // This is called when the last Cursor provided to onLoadFinished()
        // above is about to be closed.  We need to make sure we are no
        // longer using it.
        mAdapter.swapCursor(null);
    }

    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        // Do something when a list item is clicked
    }
}

```

**Note:** Because this sample performs a query on the [Contacts Provider](#), if you want to try this code, your app must request the [READ\\_CONTACTS](#) permission in the manifest file:

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

# Grid View

## In this document

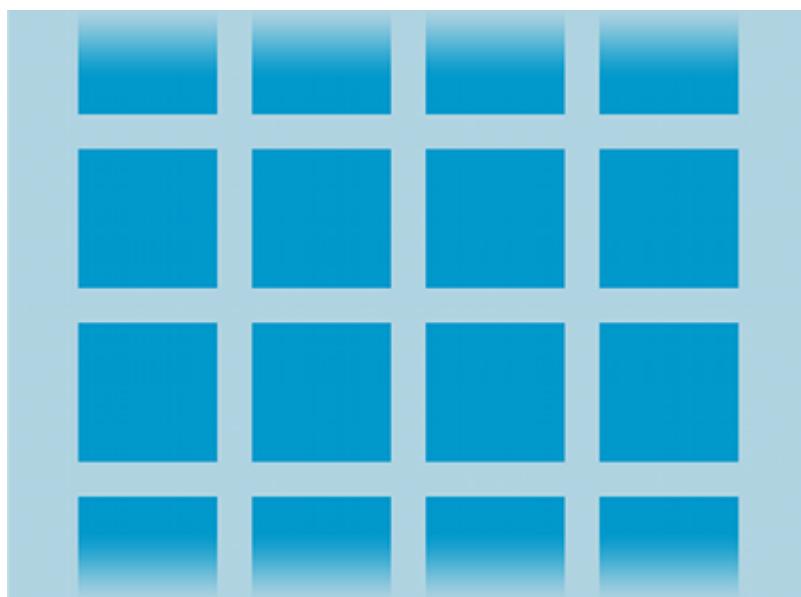
1. [Example](#)

## Key classes

1. [GridView](#)
2. [ImageView](#)
3. [BaseAdapter](#)
4. [AdapterView.OnItemClickListener](#)

`GridView` is a [ViewGroup](#) that displays items in a two-dimensional, scrollable grid. The grid items are automatically inserted to the layout using a [ListAdapter](#).

For an introduction to how you can dynamically insert views using an adapter, read [Building Layouts with an Adapter](#).



## Example

In this tutorial, you'll create a grid of image thumbnails. When an item is selected, a toast message will display the position of the image.

1. Start a new project named *HelloGridView*.
2. Find some photos you'd like to use, or [download these sample images](#). Save the image files into the project's `res/drawable/` directory.
3. Open the `res/layout/main.xml` file and insert the following:

```
<?xml version="1.0" encoding="utf-8"?>
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
```

```

        android:columnWidth="90dp"
        android:numColumns="auto_fit"
        android:verticalSpacing="10dp"
        android:horizontalSpacing="10dp"
        android:stretchMode="columnWidth"
        android:gravity="center"
    />

```

This [GridView](#) will fill the entire screen. The attributes are rather self explanatory. For more information about valid attributes, see the [GridView](#) reference.

4. Open `HelloGridView.java` and insert the following code for the [onCreate\(\)](#) method:

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    GridView gridview = (GridView) findViewById(R.id.gridview);
    gridview.setAdapter(new ImageAdapter(this));

    gridview.setOnItemClickListener(new OnItemClickListener() {
        public void onItemClick(AdapterView<?> parent, View v, int position) {
            Toast.makeText(HelloGridView.this, "" + position, Toast.LENGTH_SHORT).show();
        }
    });
}

```

After the `main.xml` layout is set for the content view, the [GridView](#) is captured from the layout with [findViewById\(int\)](#). The [setAdapter\(\)](#) method then sets a custom adapter (`ImageAdapter`) as the source for all items to be displayed in the grid. The `ImageAdapter` is created in the next step.

To do something when an item in the grid is clicked, the [setOnItemClickListener\(\)](#) method is passed a new [AdapterView.OnItemClickListener](#). This anonymous instance defines the [onItemClick\(\)](#) callback method to show a [Toast](#) that displays the index position (zero-based) of the selected item (in a real world scenario, the position could be used to get the full sized image for some other task).

5. Create a new class called `ImageAdapter` that extends [BaseAdapter](#):

```

public class ImageAdapter extends BaseAdapter {
    private Context mContext;

    public ImageAdapter(Context c) {
        mContext = c;
    }

    public int getCount() {
        return mThumbIds.length;
    }

    public Object getItem(int position) {
        return null;
    }
}

```

```

public long getItemId(int position) {
    return 0;
}

// create a new ImageView for each item referenced by the Adapter
public View getView(int position, View convertView, ViewGroup parent)
    ImageView imageView;
    if (convertView == null) { // if it's not recycled, initialize some
        imageView = new ImageView(mContext);
        imageView.setLayoutParams(new GridView.LayoutParams(85, 85));
        imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
        imageView.setPadding(8, 8, 8, 8);
    } else {
        imageView = (ImageView) convertView;
    }

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

// references to our images
private Integer[] mThumbIds = {
    R.drawable.sample_2, R.drawable.sample_3,
    R.drawable.sample_4, R.drawable.sample_5,
    R.drawable.sample_6, R.drawable.sample_7,
    R.drawable.sample_0, R.drawable.sample_1,
    R.drawable.sample_2, R.drawable.sample_3,
    R.drawable.sample_4, R.drawable.sample_5,
    R.drawable.sample_6, R.drawable.sample_7,
    R.drawable.sample_0, R.drawable.sample_1,
    R.drawable.sample_2, R.drawable.sample_3,
    R.drawable.sample_4, R.drawable.sample_5,
    R.drawable.sample_6, R.drawable.sample_7
};
}

```

First, this implements some required methods inherited from [BaseAdapter](#). The constructor and [getCount\(\)](#) are self-explanatory. Normally, [getItem\(int\)](#) should return the actual object at the specified position in the adapter, but it's ignored for this example. Likewise, [getItemId\(int\)](#) should return the row id of the item, but it's not needed here.

The first method necessary is [getView\(\)](#). This method creates a new [View](#) for each image added to the `ImageAdapter`. When this is called, a [View](#) is passed in, which is normally a recycled object (at least after this has been called once), so there's a check to see if the object is null. If it *is* null, an [Image](#) is instantiated and configured with desired properties for the image presentation:

- [setLayoutParams\(ViewGroup.LayoutParams\)](#) sets the height and width for the View—this ensures that, no matter the size of the drawable, each image is resized and cropped to fit in these dimensions, as appropriate.
- [setScaleType\(ImageView.ScaleType\)](#) declares that images should be cropped toward the center (if necessary).

- `setPadding(int, int, int, int)` defines the padding for all sides. (Note that, if the images have different aspect-ratios, then less padding will cause more cropping of the image if it does not match the dimensions given to the `ImageView`.)

If the `View` passed to `getView()` is *not* null, then the local `ImageView` is initialized with the recycled `View` object.

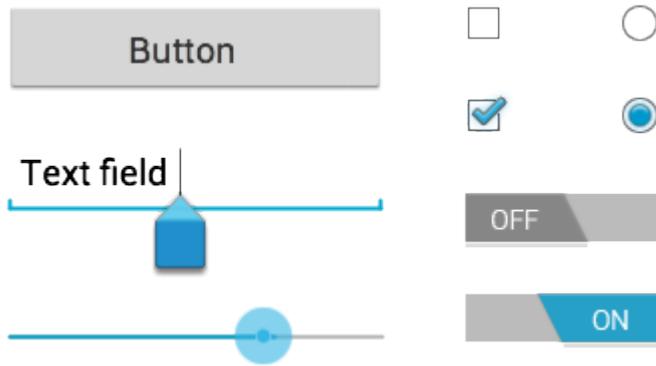
At the end of the `getView()` method, the position integer passed into the method is used to select an image from the `mThumbIds` array, which is set as the image resource for the `ImageView`.

All that's left is to define the `mThumbIds` array of drawable resources.

## 6. Run the application.

Try experimenting with the behaviors of the `GridView` and `ImageView` elements by adjusting their properties. For example, instead of using `setLayoutParams(ViewGroup.LayoutParams)`, try using `setAdjustViewBounds(boolean)`.

# Input 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, checkboxes, zoom buttons, toggle buttons, and many more.

Adding an input control to your UI is as simple as adding an XML element to your [XML layout](#). For example, here's a layout with a text field and button:

```
<?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="horizontal">
    <EditText android:id="@+id/edit_message"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
    <Button android:id="@+id/button_send"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send"
        android:onClick="sendMessage" />
</LinearLayout>
```

Each input control supports a specific set of input events so you can handle events such as when the user enters text or touches a button.

## Common Controls

Here's a list of some common controls that you can use in your app. Follow the links to learn more about using each one.

**Note:** Android provides several more controls than are listed here. Browse the [android.widget](#) package to discover more. If your app requires a specific kind of input control, you can build your own [custom components](#).

Control Type	Description	Related Classes
--------------	-------------	-----------------

<a href="#">Button</a>	A push-button that can be pressed, or clicked, by the user to perform an action.	<a href="#">Button</a>
<a href="#">Text field</a>	An editable text field. You can use the <code>AutoCompleteTextView</code> widget to create a text entry widget that provides auto-complete suggestions	<a href="#">EditText</a> , <a href="#">AutoCompleteTextView</a>
<a href="#">Checkbox</a>	An on/off switch that can be toggled by the user. You should use checkboxes when presenting users with a group of selectable options that are not mutually exclusive.	<a href="#">CheckBox</a>
<a href="#">Radio button</a>	Similar to checkboxes, except that only one option can be selected in the group.	<a href="#">RadioGroup</a> <a href="#">RadioButton</a>
<a href="#">Toggle button</a>	An on/off button with a light indicator.	<a href="#">ToggleButton</a>
<a href="#">Spinner</a>	A drop-down list that allows users to select one value from a set.	<a href="#">Spinner</a>
<a href="#">Pickers</a>	A dialog for users to select a single value for a set by using up/down buttons or via a swipe gesture. Use a <code>DatePicker</code> widget to enter the values for the date (month, day, year) or a <code>TimePicker</code> widget to enter the values for a time (hour, minute, AM/PM), which will be formatted automatically for the user's locale.	<a href="#">DatePicker</a> , <a href="#">TimePicker</a>

# Buttons

## In this document

1. [Responding to Click Events](#)
  1. [Using an OnClickListener](#)
2. [Styling Your Button](#)
  1. [Borderless button](#)
  2. [Custom background](#)

## Key classes

1. [Button](#)
2. [ImageButton](#)

A button consists of text or an icon (or both text and an icon) that communicates what action occurs when the user touches it.



Depending on whether you want a button with text, an icon, or both, you can create the button in your layout in three ways:

- With text, using the [Button](#) class:

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/button_text"  
    ... />
```

- With an icon, using the [ImageButton](#) class:

```
<ImageButton  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:src="@drawable/button_icon"  
    ... />
```

- With text and an icon, using the [Button](#) class with the [android:drawableLeft](#) attribute:

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/button_text"  
    android:drawableLeft="@drawable/button_icon"  
    ... />
```

# Responding to Click Events

When the user clicks a button, the [Button](#) object receives an on-click event.

To define the click event handler for a button, add the [android:onClick](#) attribute to the <Button> element in your XML layout. The value for this attribute must be the name of the method you want to call in response to a click event. The [Activity](#) hosting the layout must then implement the corresponding method.

For example, here's a layout with a button using [android:onClick](#):

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/button_send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage" />
```

Within the [Activity](#) that hosts this layout, the following method handles the click event:

```
/** Called when the user touches the button */
public void sendMessage(View view) {
    // Do something in response to button click
}
```

The method you declare in the [android:onClick](#) attribute must have a signature exactly as shown above. Specifically, the method must:

- Be public
- Return void
- Define a [View](#) as its only parameter (this will be the [View](#) that was clicked)

## Using an OnClickListener

You can also declare the click event handler pragmatically rather than in an XML layout. This might be necessary if you instantiate the [Button](#) at runtime or you need to declare the click behavior in a [Fragment](#) subclass.

To declare the event handler programmatically, create an [View.OnClickListener](#) object and assign it to the button by calling [setOnItemClickListener\(View.OnClickListener\)](#). For example:

```
Button button = (Button) findViewById(R.id.button_send);
button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Do something in response to button click
    }
});
```

## Styling Your Button

The appearance of your button (background image and font) may vary from one device to another, because devices by different manufacturers often have different default styles for input controls.

You can control exactly how your controls are styled using a theme that you apply to your entire application. For instance, to ensure that all devices running Android 4.0 and higher use the Holo theme in your app, declare `android:theme="@android:style/Theme.Holo"` in your manifest's `<application>` element. Also read the blog post, [Holo Everywhere](#) for information about using the Holo theme while supporting older devices.

To customize individual buttons with a different background, specify the [android:background](#) attribute with a drawable or color resource. Alternatively, you can apply a *style* for the button, which works in a manner similar to HTML styles to define multiple style properties such as the background, font, size, and others. For more information about applying styles, see [Styles and Themes](#).

## Borderless button

One design that can be useful is a "borderless" button. Borderless buttons resemble basic buttons except that they have no borders or background but still change appearance during different states, such as when clicked.

To create a borderless button, apply the [borderlessButtonStyle](#) style to the button. For example:

```
<Button  
    android:id="@+id/button_send"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/button_send"  
    android:onClick="sendMessage"  
    style="?android:attr/borderlessButtonStyle" />
```

## Custom background

If you want to truly redefine the appearance of your button, you can specify a custom background. Instead of supplying a simple bitmap or color, however, your background should be a state list resource that changes appearance depending on the button's current state.

You can define the state list in an XML file that defines three different images or colors to use for the different button states.

To create a state list drawable for your button background:

1. Create three bitmaps for the button background that represent the default, pressed, and focused button states.

To ensure that your images fit buttons of various sizes, create the bitmaps as [Nine-patch](#) bitmaps.

2. Place the bitmaps into the `res/drawable/` directory of your project. Be sure each bitmap is named properly to reflect the button state that they each represent, such as `button_default.9.png`, `button_pressed.9.png`, and `button_focused.9.png`.
3. Create a new XML file in the `res/drawable/` directory (name it something like `button_custom.xml`). Insert the following XML:

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

```
<item android:drawable="@drawable/button_default" />
</selector>
```

This defines a single drawable resource, which will change its image based on the current state of the button.

- The first `<item>` defines the bitmap to use when the button is pressed (activated).
- The second `<item>` defines the bitmap to use when the button is focused (when the button is highlighted using the trackball or directional pad).
- The third `<item>` defines the bitmap to use when the button is in the default state (it's neither pressed nor focused).

**Note:** The order of the `<item>` elements is important. When this drawable is referenced, the `<item>` elements are traversed in-order to determine which one is appropriate for the current button state. Because the default bitmap is last, it is only applied when the conditions `android:state_pressed` and `android:state_focused` have both evaluated as false.

This XML file now represents a single drawable resource and when referenced by a [Button](#) for its background, the image displayed will change based on these three states.

4. Then simply apply the drawable XML file as the button background:

```
<Button
    android:id="@+id/button_send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage"
    android:background="@drawable/button_custom" />
```

For more information about this XML syntax, including how to define a disabled, hovered, or other button states, read about [State List Drawable](#).

# Text Fields

## In this document

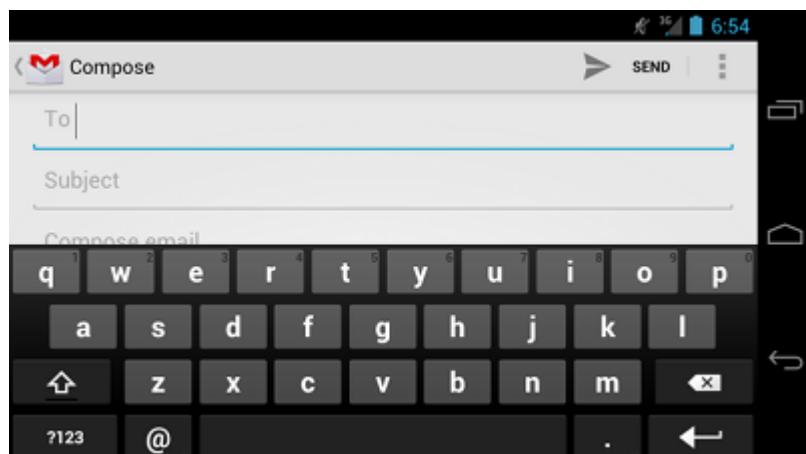
1. [Specifying the Keyboard Type](#)
  1. [Controlling other behaviors](#)
2. [Specifying Keyboard Actions](#)
  1. [Responding to action button events](#)
  2. [Setting a custom action button label](#)
3. [Adding Other Keyboard Flags](#)
4. [Providing Auto-complete Suggestions](#)

## Key classes

1. [EditText](#)
2. [AutoCompleteTextView](#)

A text field allows the user to type text into your app. It can be either single line or multi-line. Touching a text field places the cursor and automatically displays the keyboard. In addition to typing, text fields allow for a variety of other activities, such as text selection (cut, copy, paste) and data look-up via auto-completion.

You can add a text field to you layout with the [EditText](#) object. You should usually do so in your XML layout with a <EditText> element.



## Specifying the Keyboard Type



Figure 1. The default text input type.



**Figure 2.** The `textEmailAddress` input type.



**Figure 3.** The `phone` input type.

Text fields can have different input types, such as number, date, password, or email address. The type determines what kind of characters are allowed inside the field, and may prompt the virtual keyboard to optimize its layout for frequently used characters.

You can specify the type of keyboard you want for your `EditText` object with the `android:inputType` attribute. For example, if you want the user to input an email address, you should use the `textEmailAddress` input type:

```
<EditText  
    android:id="@+id/email_address"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:hint="@string/email_hint"  
    android:inputType="textEmailAddress" />
```

There are several different input types available for different situations. Here are some of the more common values for `android:inputType`:

**"text"**

Normal text keyboard.

**"textEmailAddress"**

Normal text keyboard with the @ character.

**"textUri"**

Normal text keyboard with the / character.

**"number"**

Basic number keypad.

## "phone"

Phone-style keypad.

## Controlling other behaviors

The [android:inputType](#) also allows you to specify certain keyboard behaviors, such as whether to capitalize all new words or use features like auto-complete and spelling suggestions.

The [android:inputType](#) attribute allows bitwise combinations so you can specify both a keyboard layout and one or more behaviors at once.

Here are some of the common input type values that define keyboard behaviors:

### "textCapSentences"

Normal text keyboard that capitalizes the first letter for each new sentence.

### "textCapWords"

Normal text keyboard that capitalizes every word. Good for titles or person names.

### "textAutoCorrect"

Normal text keyboard that corrects commonly misspelled words.

### "textPassword"

Normal text keyboard, but the characters entered turn into dots.

### "textMultiLine"

Normal text keyboard that allow users to input long strings of text that include line breaks (carriage returns).

For example, here's how you can collect a postal address, capitalize each word, and disable text suggestions:

```
<EditText  
    android:id="@+id/postal_address"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:hint="@string/postal_address_hint"  
    android:inputType="textPostalAddress |  
        textCapWords |  
        textNoSuggestions" />
```

All behaviors are also listed with the [android:inputType](#) documentation.

## Specifying Keyboard Actions



**Figure 4.** If you declare `android:imeOptions="actionSend"`, the keyboard includes the Send action.

In addition to changing the keyboard's input type, Android allows you to specify an action to be made when users have completed their input. The action specifies the button that appears in place of the carriage return key and the action to be made, such as "Search" or "Send."

You can specify the action by setting the [android:imeOptions](#) attribute. For example, here's how you can specify the Send action:

```
<EditText  
    android:id="@+id/search"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:hint="@string/search_hint"  
    android:inputType="text"  
    android:imeOptions="actionSend" />
```

If you do not explicitly specify an input action then the system attempts to determine if there are any subsequent [android:focusable](#) fields. If any focusable fields are found following this one, the system applies the {@code actionNext} action to the current [EditText](#) so the user can select Next to move to the next field. If there's no subsequent focusable field, the system applies the "actionDone" action. You can override this by setting the [android:imeOptions](#) attribute to any other value such as "actionSend" or "actionSearch" or suppress the default behavior by using the "actionNone" action.

## Responding to action button events

If you have specified a keyboard action for the input method using [android:imeOptions](#) attribute (such as "actionSend"), you can listen for the specific action event using an [TextView.OnEditorActionListener](#). The [TextView.OnEditorActionListener](#) interface provides a callback method called [onEditorAction\(\)](#) that indicates the action type invoked with an action ID such as [IME\\_ACTION\\_SEND](#) or [IME\\_ACTION\\_SEARCH](#).

For example, here's how you can listen for when the user clicks the Send button on the keyboard:

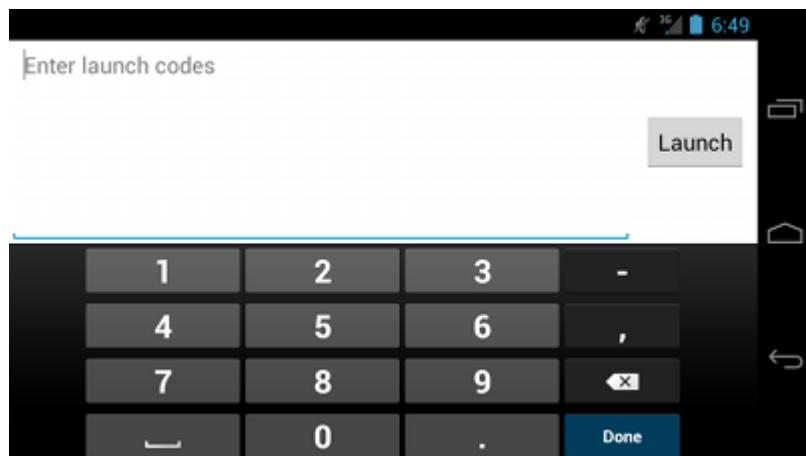
```
EditText editText = (EditText) findViewById(R.id.search);  
editText.setOnEditorActionListener(new OnEditorActionListener() {  
    @Override  
    public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {  
        boolean handled = false;  
        if (actionId == EditorInfo.IME_ACTION_SEND) {  
            sendMessage();  
            handled = true;  
        }  
        return handled;  
    }  
});
```

## Setting a custom action button label

If the keyboard is too large to reasonably share space with the underlying application (such as when a handset device is in landscape orientation) then fullscreen ("extract mode") is triggered. In this mode, a labeled action button is displayed next to the input. You can customize the text of this button by setting the [android:imeActionButton](#) attribute:

```
<EditText  
    android:id="@+id/launch_codes"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:hint="@string/enter_launch_codes"
```

```
    android:inputType="number"
    android:imeActionLabel="@string/launch" />
```

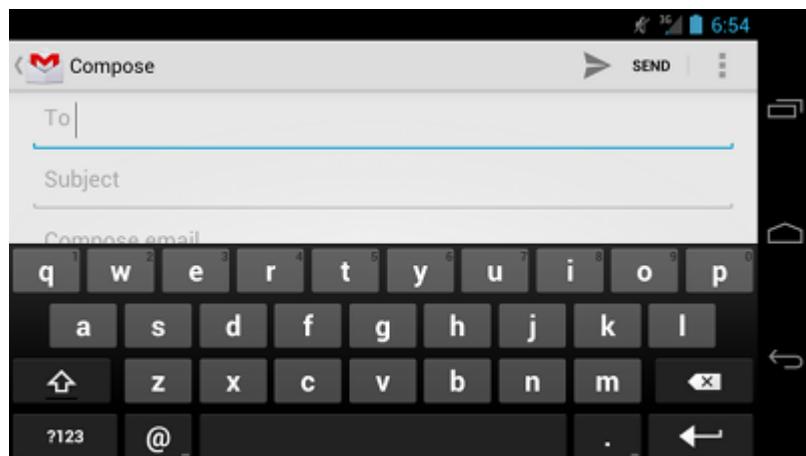


**Figure 5.** A custom action label with [android:imeActionLabel](#).

## Adding Other Keyboard Flags

In addition to the actions you can specify with the [android:imeOptions](#) attribute, you can add additional flags to specify other keyboard behaviors. All available flags are listed along with the actions in the [an-droid:imeOptions](#) documentation.

For example, figure 5 shows how the system enables a fullscreen text field when a handset device is in landscape orientation (or the screen space is otherwise constrained for space). You can disable the fullscreen input mode with `flagNoExtractUi` in the [android:imeOptions](#) attribute, as shown in figure 6.



**Figure 6.** The fullscreen text field ("extract mode") is disabled with [an-droid:imeOptions="flagNoExtractUi"](#).

## Providing Auto-complete Suggestions

If you want to provide suggestions to users as they type, you can use a subclass of [EditText](#) called [Auto-CompleteTextView](#). To implement auto-complete, you must specify an (@link android.widget.Adapter) that provides the text suggestions. There are several kinds of adapters available, depending on where the data is coming from, such as from a database or an array.



**Figure 7.** Example of [AutoCompleteTextView](#) with text suggestions.

The following procedure describes how to set up an [AutoCompleteTextView](#) that provides suggestions from an array, using [ArrayAdapter](#):

1. Add the [AutoCompleteTextView](#) to your layout. Here's a layout with only the text field:

```
<?xml version="1.0" encoding="utf-8"?>
<AutoCompleteTextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/autocomplete_country"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

2. Define the array that contains all text suggestions. For example, here's an array of country names that's defined in an XML resource file (`res/values/strings.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="countries_array">
        <item>Afghanistan</item>
        <item>Albania</item>
        <item>Algeria</item>
        <item>American Samoa</item>
        <item>Andorra</item>
        <item>Angola</item>
        <item>Anguilla</item>
        <item>Antarctica</item>
        ...
    </string-array>
</resources>
```

3. In your [Activity](#) or [Fragment](#), use the following code to specify the adapter that supplies the suggestions:

```
// Get a reference to the AutoCompleteTextView in the layout
AutoCompleteTextView textView = (AutoCompleteTextView) findViewById(R.id.autocomplete_country);
// Get the string array
String[] countries = getResources().getStringArray(R.array.countries_array);
// Create the adapter and set it to the AutoCompleteTextView
ArrayAdapter<String> adapter =
    new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, countries);
textView.setAdapter(adapter);
```

Here, a new [ArrayAdapter](#) is initialized to bind each item in the COUNTRIES string array to a [TextView](#) that exists in the `simple_list_item_1` layout (this is a layout provided by Android that provides a standard appearance for text in a list).

Then assign the adapter to the [AutoCompleteTextView](#) by calling [setAdapter\(\)](#).

# Checkboxes

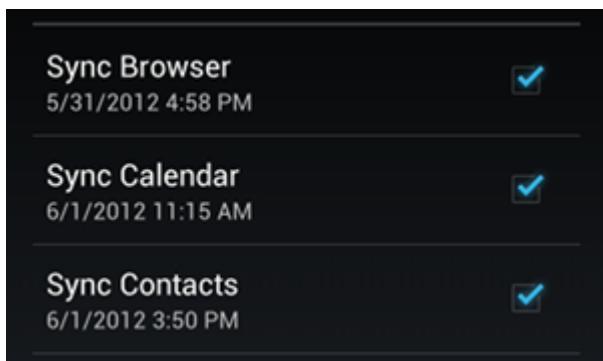
## In this document

1. [Responding to Click Events](#)

## Key classes

1. [CheckBox](#)

Checkboxes allow the user to select one or more options from a set. Typically, you should present each checkbox option in a vertical list.



To create each checkbox option, create a [CheckBox](#) in your layout. Because a set of checkbox options allows the user to select multiple items, each checkbox is managed separately and you must register a click listener for each one.

## Responding to Click Events

When the user selects a checkbox, the [CheckBox](#) object receives an on-click event.

To define the click event handler for a checkbox, add the [android:onClick](#) attribute to the <CheckBox> element in your XML layout. The value for this attribute must be the name of the method you want to call in response to a click event. The [Activity](#) hosting the layout must then implement the corresponding method.

For example, here are a couple [CheckBox](#) objects in a list:

```
<?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">
    <CheckBox android:id="@+id/checkbox_meat"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/meat"
        android:onClick="onCheckboxClicked"/>
    <CheckBox android:id="@+id/checkbox_cheese"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/cheese"
```

```
    android:onClick="onCheckboxClicked"/>
</LinearLayout>
```

Within the [Activity](#) that hosts this layout, the following method handles the click event for both checkboxes:

```
public void onCheckboxClicked(View view) {
    // Is the view now checked?
    boolean checked = ((CheckBox) view).isChecked();

    // Check which checkbox was clicked
    switch(view.getId()) {
        case R.id.checkbox_meat:
            if (checked)
                // Put some meat on the sandwich
            else
                // Remove the meat
            break;
        case R.id.checkbox_cheese:
            if (checked)
                // Cheese me
            else
                // I'm lactose intolerant
            break;
        // TODO: Veggie sandwich
    }
}
```

The method you declare in the [android:onClick](#) attribute must have a signature exactly as shown above. Specifically, the method must:

- Be public
- Return void
- Define a [View](#) as its only parameter (this will be the [view](#) that was clicked)

**Tip:** If you need to change the radio button state yourself (such as when loading a saved [CheckBoxPreference](#)), use the [setChecked\(boolean\)](#) or [toggle\(\)](#) method.

# Radio Buttons

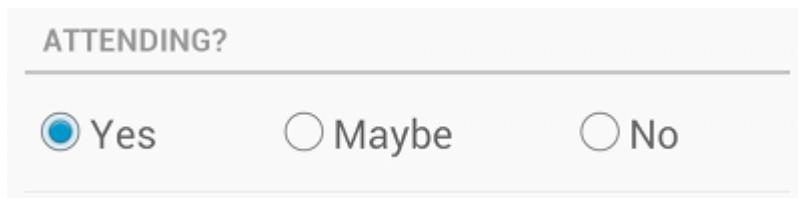
## In this document

1. [Responding to Click Events](#)

## Key classes

1. [RadioButton](#)
2. [RadioGroup](#)

Radio buttons allow the user to select one option from a set. You should use radio buttons for optional sets that are mutually exclusive if you think that the user needs to see all available options side-by-side. If it's not necessary to show all options side-by-side, use a [spinner](#) instead.



To create each radio button option, create a [RadioButton](#) in your layout. However, because radio buttons are mutually exclusive, you must group them together inside a [RadioGroup](#). By grouping them together, the system ensures that only one radio button can be selected at a time.

## Responding to Click Events

When the user selects one of the radio buttons, the corresponding [RadioButton](#) object receives an on-click event.

To define the click event handler for a button, add the [android:onClick](#) attribute to the <RadioButton> element in your XML layout. The value for this attribute must be the name of the method you want to call in response to a click event. The [Activity](#) hosting the layout must then implement the corresponding method.

For example, here are a couple [RadioButton](#) objects:

```
<?xml version="1.0" encoding="utf-8"?>
<RadioGroup xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <RadioButton android:id="@+id/radio_pirates"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/pirates"
        android:onClick="onRadioButtonClicked"/>
    <RadioButton android:id="@+id/radio_ninjas"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/ninjas"
```

```
    android:onClick="onRadioButtonClicked"/>
</RadioGroup>
```

**Note:** The [RadioGroup](#) is a subclass of [LinearLayout](#) that has a vertical orientation by default.

Within the [Activity](#) that hosts this layout, the following method handles the click event for both radio buttons:

```
public void onRadioButtonClicked(View view) {
    // Is the button now checked?
    boolean checked = ((RadioButton) view).isChecked();

    // Check which radio button was clicked
    switch(view.getId()) {
        case R.id.radio_pirates:
            if (checked)
                // Pirates are the best
                break;
        case R.id.radio_ninjas:
            if (checked)
                // Ninjas rule
            break;
    }
}
```

The method you declare in the [android:onClick](#) attribute must have a signature exactly as shown above. Specifically, the method must:

- Be public
- Return void
- Define a [View](#) as its only parameter (this will be the [View](#) that was clicked)

**Tip:** If you need to change the radio button state yourself (such as when loading a saved [CheckBoxPreference](#)), use the [setChecked\(boolean\)](#) or [toggle\(\)](#) method.

# Toggle Buttons

## In this document

1. [Responding to Click Events](#)
  1. [Using an OnCheckedChangeListener](#)

## Key classes

1. [ToggleButton](#)
2. [Switch](#)

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

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



*Toggle buttons*



*Switches (in Android 4.0+)*

The [ToggleButton](#) and [Switch](#) controls are subclasses of [CompoundButton](#) and function in the same manner, so you can implement their behavior the same way.

## Responding to Click Events

When the user selects a [ToggleButton](#) and [Switch](#), the object receives an on-click event.

To define the click event handler, add the [android:onClick](#) attribute to the <ToggleButton> or <Switch> element in your XML layout. The value for this attribute must be the name of the method you want to call in response to a click event. The [Activity](#) hosting the layout must then implement the corresponding method.

For example, here's a [ToggleButton](#) with the [android:onClick](#) attribute:

```
<ToggleButton  
    android:id="@+id/togglebutton"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:textOn="Vibrate on"  
    android:textOff="Vibrate off"  
    android:onClick="onToggleClicked"/>
```

Within the [Activity](#) that hosts this layout, the following method handles the click event:

```
public void onToggleClicked(View view) {  
    // Is the toggle on?  
    boolean on = ((ToggleButton) view).isChecked();  
  
    if (on) {  
        // Enable vibrate  
    } else {  
        // Disable vibrate  
    }  
}
```

The method you declare in the [android:onClick](#) attribute must have a signature exactly as shown above. Specifically, the method must:

- Be public
- Return void
- Define a [View](#) as its only parameter (this will be the [View](#) that was clicked)

**Tip:** If you need to change the state yourself, use the [setChecked\(boolean\)](#) or [toggle\(\)](#) method to change the state.

## Using an OnCheckedChangeListener

You can also declare a click event handler pragmatically rather than in an XML layout. This might be necessary if you instantiate the [ToggleButton](#) or [Switch](#) at runtime or you need to declare the click behavior in a [Fragment](#) subclass.

To declare the event handler programmatically, create an [CompoundButton.OnCheckedChangeListener](#) object and assign it to the button by calling [setOnCheckedChangeListener\(CompoundButton.OnCheckedChangeListener\)](#). For example:

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

# Spinners

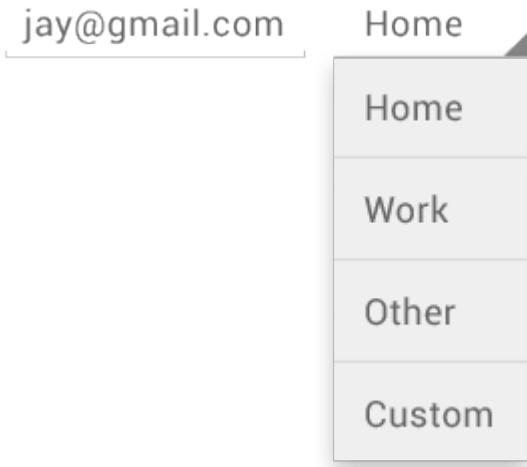
## In this document

1. [Populate the Spinner with User Choices](#)
2. [Responding to User Selections](#)

## Key classes

1. [Spinner](#)
2. [SpinnerAdapter](#)
3. [AdapterView.OnItemSelectedListener](#)

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.



You can add a spinner to your layout with the [Spinner](#) object. You should usually do so in your XML layout with a <Spinner> element. For example:

```
<Spinner  
    android:id="@+id/planets_spinner"  
    android:layout_width="fill_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.

## Populate the Spinner with User Choices

The choices you provide for the spinner can come from any source, but must be provided through an [SpinnerAdapter](#), such as an [ArrayAdapter](#) if the choices are available in an array or a [CursorAdapter](#) if the choices are available from a database query.

For instance, if the available choices for your spinner are pre-determined, you can provide them with a string array defined in a [string resource file](#):

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

With an array such as this one, you can use the following code in your [Activity](#) or [Fragment](#) to supply the spinner with the array using an instance of [ArrayAdapter](#):

```
Spinner spinner = (Spinner) findViewById(R.id.spinner);
// Create an ArrayAdapter using the string array and a default spinner layout
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
    R.array.planets_array, android.R.layout.simple_spinner_item);
// Specify the layout to use when the list of choices appears
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
// Apply the adapter to the spinner
spinner.setAdapter(adapter);
```

The [createFromResource\(\)](#) method allows you to create an [ArrayAdapter](#) from the string array. The third argument for this method is a layout resource that defines how the selected choice appears in the spinner control. The [simple\\_spinner\\_item](#) layout is provided by the platform and is the default layout you should use unless you'd like to define your own layout for the spinner's appearance.

You should then call [setDropDownViewResource\(int\)](#) to specify the layout the adapter should use to display the list of spinner choices ([simple\\_spinner\\_dropdown\\_item](#) is another standard layout defined by the platform).

Call [setAdapter\(\)](#) to apply the adapter to your [Spinner](#).

## Responding to User Selections

When the user selects an item from the drop-down, the [Spinner](#) object receives an on-item-selected event.

To define the selection event handler for a spinner, implement the [AdapterView.OnItemSelectedListener](#) interface and the corresponding [onItemSelected\(\)](#) callback method. For example, here's an implementation of the interface in an [Activity](#):

```
public class SpinnerActivity extends Activity implements OnItemSelectedListener {
    ...
    public void onItemSelected(AdapterView<?> parent, View view,
        int pos, long id) {
        // An item was selected. You can retrieve the selected item using
        // parent.getItemAtPosition(pos)
    }
}
```

```
public void onNothingSelected(AdapterView<?> parent) {  
    // Another interface callback  
}  
}
```

The [AdapterView.OnItemSelectedListener](#) requires the [onItemSelected\(\)](#) and [onNothingSelected\(\)](#) callback methods.

Then you need to specify the interface implementation by calling [setOnItemSelectedListener\(\)](#):

```
Spinner spinner = (Spinner) findViewById(R.id.spinner);  
spinner.setOnItemSelectedListener(this);
```

If you implement the [AdapterView.OnItemSelectedListener](#) interface with your [Activity](#) or [Fragment](#) (such as in the example above), you can pass `this` as the interface instance.

# Pickers

## In this document

1. [Creating a Time Picker](#)
  1. [Extending DialogFragment for a time picker](#)
  2. [Showing the time picker](#)
2. [Creating a Date Picker](#)
  1. [Extending DialogFragment for a date picker](#)
  2. [Showing the date picker](#)

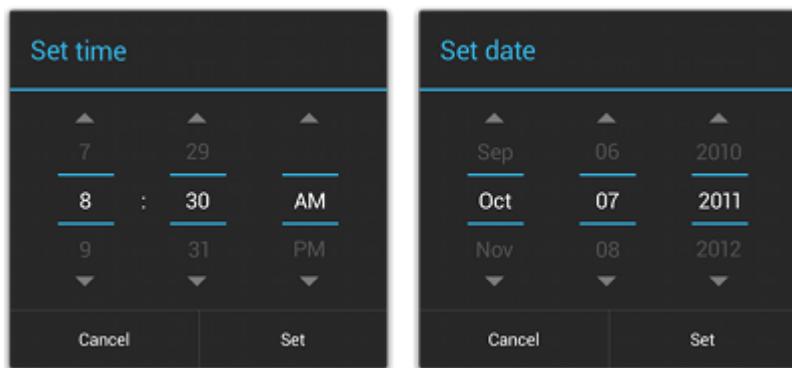
## Key classes

1. [DatePickerDialog](#)
2. [TimePickerDialog](#)
3. [DialogFragment](#)

## See also

1. [Fragments](#)

Android provides controls for the user to pick a time or pick a date as ready-to-use dialogs. Each picker provides controls for selecting each part of the time (hour, minute, AM/PM) or date (month, day, year). Using these pickers helps ensure that your users can pick a time or date that is valid, formatted correctly, and adjusted to the user's locale.



We recommend that you use [DialogFragment](#) to host each time or date picker. The [DialogFragment](#) manages the dialog lifecycle for you and allows you to display the pickers in different layout configurations, such as in a basic dialog on handsets or as an embedded part of the layout on large screens.

Although [DialogFragment](#) was first added to the platform in Android 3.0 (API level 11), if your app supports versions of Android older than 3.0—even as low as Android 1.6—you can use the [DialogFragment](#) class that's available in the [support library](#) for backward compatibility.

**Note:** The code samples below show how to create dialogs for a time picker and date picker using the [support library](#) APIs for [DialogFragment](#). If your app's [minSdkVersion](#) is 11 or higher, you can instead use the platform version of [DialogFragment](#).

# Creating a Time Picker

To display a [TimePickerDialog](#) using [DialogFragment](#), you need to define a fragment class that extends [DialogFragment](#) and return a [TimePickerDialog](#) from the fragment's [onCreateDialog\(\)](#) method.

**Note:** If your app supports versions of Android older than 3.0, be sure you've set up your Android project with the support library as described in [Setting Up a Project to Use a Library](#).

## Extending DialogFragment for a time picker

To define a [DialogFragment](#) for a [TimePickerDialog](#), you must:

- Define the [onCreateDialog\(\)](#) method to return an instance of [TimePickerDialog](#)
- Implement the [TimePickerDialog.OnTimeSetListener](#) interface to receive a callback when the user sets the time.

Here's an example:

```
public static class TimePickerFragment extends DialogFragment
        implements TimePickerDialog.OnTimeSetListener {

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the current time as the default values for the picker
        final Calendar c = Calendar.getInstance();
        int hour = c.get(Calendar.HOUR_OF_DAY);
        int minute = c.get(Calendar.MINUTE);

        // Create a new instance of TimePickerDialog and return it
        return new TimePickerDialog(getActivity(), this, hour, minute,
                DateFormat.is24HourFormat(getActivity()));
    }

    public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
        // Do something with the time chosen by the user
    }
}
```

See the [TimePickerDialog](#) class for information about the constructor arguments.

Now all you need is an event that adds an instance of this fragment to your activity.

## Showing the time picker

Once you've defined a [DialogFragment](#) like the one shown above, you can display the time picker by creating an instance of the [DialogFragment](#) and calling [show\(\)](#).

For example, here's a button that, when clicked, calls a method to show the dialog:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```
    android:text="@string/pick_time"
    android:onClick="showTimePickerDialog" />
```

When the user clicks this button, the system calls the following method:

```
public void showTimePickerDialog(View v) {
    DialogFragment newFragment = new TimePickerFragment();
    newFragment.show(getSupportFragmentManager(), "timePicker");
}
```

This method calls [show\(\)](#) on a new instance of the [DialogFragment](#) defined above. The [show\(\)](#) method requires an instance of [FragmentManager](#) and a unique tag name for the fragment.

**Caution:** If your app supports versions of Android lower than 3.0, be sure that you call [getSupportFragmentManager\(\)](#) to acquire an instance of [FragmentManager](#). Also make sure that your activity that displays the time picker extends [FragmentActivity](#) instead of the standard [Activity](#) class.

## Creating a Date Picker

Creating a [DatePickerDialog](#) is just like creating a [TimePickerDialog](#). The only difference is the dialog you create for the fragment.

To display a [DatePickerDialog](#) using [DialogFragment](#), you need to define a fragment class that extends [DialogFragment](#) and return a [DatePickerDialog](#) from the fragment's [onCreateDialog\(\)](#) method.

**Note:** If your app supports versions of Android older than 3.0, be sure you've set up your Android project with the support library as described in [Setting Up a Project to Use a Library](#).

### Extending DialogFragment for a date picker

To define a [DialogFragment](#) for a [DatePickerDialog](#), you must:

- Define the [onCreateDialog\(\)](#) method to return an instance of [DatePickerDialog](#)
- Implement the [DatePickerDialog.OnDateSetListener](#) interface to receive a callback when the user sets the date.

Here's an example:

```
public static class DatePickerFragment extends DialogFragment
    implements DatePickerDialog.OnDateSetListener {

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the current date as the default date in the picker
        final Calendar c = Calendar.getInstance();
        int year = c.get(Calendar.YEAR);
        int month = c.get(Calendar.MONTH);
        int day = c.get(Calendar.DAY_OF_MONTH);

        // Create a new instance of DatePickerDialog and return it
        return new DatePickerDialog(getActivity(), this, year, month, day);
    }
}
```

```
public void onDateSet(DatePicker view, int year, int month, int day) {  
    // Do something with the date chosen by the user  
}  
}
```

See the [DatePickerDialog](#) class for information about the constructor arguments.

Now all you need is an event that adds an instance of this fragment to your activity.

## Showing the date picker

Once you've defined a [DialogFragment](#) like the one shown above, you can display the date picker by creating an instance of the [DialogFragment](#) and calling [show\(\)](#).

For example, here's a button that, when clicked, calls a method to show the dialog:

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/pick_date"  
    android:onClick="showDatePickerDialog" />
```

When the user clicks this button, the system calls the following method:

```
public void showDatePickerDialog(View v) {  
    DialogFragment newFragment = new DatePickerFragment();  
    newFragment.show(getSupportFragmentManager(), "datePicker");  
}
```

This method calls [show\(\)](#) on a new instance of the [DialogFragment](#) defined above. The [show\(\)](#) method requires an instance of [FragmentManager](#) and a unique tag name for the fragment.

**Caution:** If your app supports versions of Android lower than 3.0, be sure that you call [getSupportFragmentManager\(\)](#) to acquire an instance of [FragmentManager](#). Also make sure that your activity that displays the time picker extends [FragmentActivity](#) instead of the standard [Activity](#) class.

# Input Events

## In this document

1. [Event Listeners](#)
2. [Event Handlers](#)
3. [Touch Mode](#)
4. [Handling Focus](#)

On Android, there's more than one way to intercept the events from a user's interaction with your application. When considering events within your user interface, the approach is to capture the events from the specific View object that the user interacts with. The View class provides the means to do so.

Within the various View classes that you'll use to compose your layout, you may notice several public callback methods that look useful for UI events. These methods are called by the Android framework when the respective action occurs on that object. For instance, when a View (such as a Button) is touched, the `onTouchEvent()` method is called on that object. However, in order to intercept this, you must extend the class and override the method. However, extending every View object in order to handle such an event would not be practical. This is why the View class also contains a collection of nested interfaces with callbacks that you can much more easily define. These interfaces, called [event listeners](#), are your ticket to capturing the user interaction with your UI.

While you will more commonly use the event listeners to listen for user interaction, there may come a time when you do want to extend a View class, in order to build a custom component. Perhaps you want to extend the [Button](#) class to make something more fancy. In this case, you'll be able to define the default event behaviors for your class using the class [event handlers](#).

## 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.

Included in the event listener interfaces are the following callback methods:

### `onClick()`

From [View.OnClickListener](#). This is called when the user either touches the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses the suitable "enter" key or presses down on the trackball.

### `onLongClick()`

From [View.OnLongClickListener](#). This is called when the user either touches and holds the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses and holds the suitable "enter" key or presses and holds down on the trackball (for one second).

### `onFocusChange()`

From [View.OnFocusChangeListener](#). This is called when the user navigates onto or away from the item, using the navigation-keys or trackball.

### `onKey()`

From [View.OnKeyListener](#). This is called when the user is focused on the item and presses or releases a hardware key on the device.

## **onTouch()**

From [View.OnTouchListener](#). This is called when the user performs an action qualified as a touch event, including a press, a release, or any movement gesture on the screen (within the bounds of the item).

## **onCreateContextMenu()**

From [View.OnCreateContextMenuListener](#). This is called when a Context Menu is being built (as the result of a sustained "long click"). See the discussion on context menus in the [Menus](#) developer guide.

These methods are the sole inhabitants of their respective interface. To define one of these methods and handle your events, implement the nested interface in your Activity or define it as an anonymous class. Then, pass an instance of your implementation to the respective `View.setOnClickListener()` method. (E.g., call [setOnTouchListener\(\)](#) and pass it your implementation of the [OnClickListener](#).)

The example below shows how to register an on-click listener for a Button.

```
// Create an anonymous implementation of OnClickListener
private OnClickListener mCorkyListener = new OnClickListener() {
    public void onClick(View v) {
        // do something when the button is clicked
    }
};

protected void onCreate(Bundle savedInstanceState) {
    ...
    // Capture our button from layout
    Button button = (Button) findViewById(R.id.corky);
    // Register the onClick listener with the implementation above
    button.setOnClickListener(mCorkyListener);
    ...
}
```

You may also find it more convenient to implement `OnClickListener` as a part of your Activity. This will avoid the extra class load and object allocation. For example:

```
public class ExampleActivity extends Activity implements OnClickListener {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        Button button = (Button) findViewById(R.id.corky);
        button.setOnClickListener(this);
    }

    // Implement the OnClickListener callback
    public void onClick(View v) {
        // do something when the button is clicked
    }
    ...
}
```

Notice that the `onClick()` callback in the above example has no return value, but some other event listener methods must return a boolean. The reason depends on the event. For the few that do, here's why:

- [onLongClick\(\)](#) - This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return `true` to indicate that you have handled the event and it

should stop here; return *false* if you have not handled it and/or the event should continue to any other on-click listeners.

- [onKey \(\)](#) - This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return *true* to indicate that you have handled the event and it should stop here; return *false* if you have not handled it and/or the event should continue to any other on-key listeners.
- [onTouch \(\)](#) - This returns a boolean to indicate whether your listener consumes this event. The important thing is that this event can have multiple actions that follow each other. So, if you return *false* when the down action event is received, you indicate that you have not consumed the event and are also not interested in subsequent actions from this event. Thus, you will not be called for any other actions within the event, such as a finger gesture, or the eventual up action event.

Remember that hardware key events are always delivered to the View currently in focus. They are dispatched starting from the top of the View hierarchy, and then down, until they reach the appropriate destination. If your View (or a child of your View) currently has focus, then you can see the event travel through the [dispatchKeyEvent \(\)](#) method. As an alternative to capturing key events through your View, you can also receive all of the events inside your Activity with [onKeyDown \(\)](#) and [onKeyUp \(\)](#).

Also, when thinking about text input for your application, remember that many devices only have software input methods. Such methods are not required to be key-based; some may use voice input, handwriting, and so on. Even if an input method presents a keyboard-like interface, it will generally **not** trigger the [onKeyDown \(\)](#) family of events. You should never build a UI that requires specific key presses to be controlled unless you want to limit your application to devices with a hardware keyboard. In particular, do not rely on these methods to validate input when the user presses the return key; instead, use actions like [IME ACTION DONE](#) to signal the input method how your application expects to react, so it may change its UI in a meaningful way. Avoid assumptions about how a software input method should work and just trust it to supply already formatted text to your application.

**Note:** Android will call event handlers first and then the appropriate default handlers from the class definition second. As such, returning *true* from these event listeners will stop the propagation of the event to other event listeners and will also block the callback to the default event handler in the View. So be certain that you want to terminate the event when you return *true*.

## Event Handlers

If you're building a custom component from View, then you'll be able to define several callback methods used as default event handlers. In the document about [Custom Components](#), you'll learn see some of the common callbacks used for event handling, including:

- [onKeyDown \(int, KeyEvent\)](#) - Called when a new key event occurs.
- [onKeyUp \(int, KeyEvent\)](#) - Called when a key up event occurs.
- [onTrackballEvent \(MotionEvent\)](#) - Called when a trackball motion event occurs.
- [onTouchEvent \(MotionEvent\)](#) - Called when a touch screen motion event occurs.
- [onFocusChanged \(boolean, int, Rect\)](#) - Called when the view gains or loses focus.

There are some other methods that you should be aware of, which are not part of the View class, but can directly impact the way you're able to handle events. So, when managing more complex events inside a layout, consider these other methods:

- [Activity.dispatchTouchEvent \(MotionEvent\)](#) - This allows your [Activity](#) to intercept all touch events before they are dispatched to the window.
- [ViewGroup.onInterceptTouchEvent \(MotionEvent\)](#) - This allows a [ViewGroup](#) to watch events as they are dispatched to child Views.

- `ViewParent.requestDisallowInterceptTouchEvent(boolean)` - Call this upon a parent View to indicate that it should not intercept touch events with `onInterceptTouchEvent(MotionEvent)`.

## Touch Mode

When a user is navigating a user interface with directional keys or a trackball, it is necessary to give focus to actionable items (like buttons) so the user can see what will accept input. If the device has touch capabilities, however, and the user begins interacting with the interface by touching it, then it is no longer necessary to highlight items, or give focus to a particular View. Thus, there is a mode for interaction named "touch mode."

For a touch-capable device, once the user touches the screen, the device will enter touch mode. From this point onward, only Views for which `isFocusableInTouchMode()` is true will be focusable, such as text editing widgets. Other Views that are touchable, like buttons, will not take focus when touched; they will simply fire their on-click listeners when pressed.

Any time a user hits a directional key or scrolls with a trackball, the device will exit touch mode, and find a view to take focus. Now, the user may resume interacting with the user interface without touching the screen.

The touch mode state is maintained throughout the entire system (all windows and activities). To query the current state, you can call `isInTouchMode()` to see whether the device is currently in touch mode.

## Handling Focus

The framework will handle routine focus movement in response to user input. This includes changing the focus as Views are removed or hidden, or as new Views become available. Views indicate their willingness to take focus through the `isFocusable()` method. To change whether a View can take focus, call `setFocusable()`. When in touch mode, you may query whether a View allows focus with `isFocusableInTouchMode()`. You can change this with `setFocusableInTouchMode()`.

Focus movement is based on an algorithm which finds the nearest neighbor in a given direction. In rare cases, the default algorithm may not match the intended behavior of the developer. In these situations, you can provide explicit overrides with the following XML attributes in the layout file: `nextFocusDown`, `nextFocusLeft`, `nextFocusRight`, and `nextFocusUp`. Add one of these attributes to the View *from* which the focus is leaving. Define the value of the attribute to be the id of the View *to* which focus should be given. For example:

```
<LinearLayout
    android:orientation="vertical"
    ...
    <Button android:id="@+id/top"
        android:nextFocusUp="@+id/bottom"
        ...
    >
    <Button android:id="@+id/bottom"
        android:nextFocusDown="@+id/top"
        ...
    >
</LinearLayout>
```

Ordinarily, in this vertical layout, navigating up from the first Button would not go anywhere, nor would navigating down from the second Button. Now that the top Button has defined the bottom one as the `nextFocusUp` (and vice versa), the navigation focus will cycle from top-to-bottom and bottom-to-top.

If you'd like to declare a View as focusable in your UI (when it is traditionally not), add the `android:focusable` XML attribute to the View, in your layout declaration. Set the value *true*. You can also declare a View as focusable while in Touch Mode with `android:focusableInTouchMode`.

To request a particular View to take focus, call [`requestFocus\(\)`](#).

To listen for focus events (be notified when a View receives or loses focus), use [`onFocusChange\(\)`](#), as discussed in the [Event Listeners](#) section, above.

# Menus

## In this document

1. [Defining a Menu in XML](#)
2. [Creating an Options Menu](#)
  1. [Handling click events](#)
  2. [Changing menu items at runtime](#)
3. [Creating Contextual Menus](#)
  1. [Creating a floating context menu](#)
  2. [Using the contextual action mode](#)
4. [Creating a Popup Menu](#)
  1. [Handling click events](#)
5. [Creating Menu Groups](#)
  1. [Using checkable menu items](#)
6. [Adding Menu Items Based on an Intent](#)
  1. [Allowing your activity to be added to other menus](#)

## Key classes

1. [Menu](#)
2. [MenuItem](#)
3. [ContextMenu](#)
4. [ActionMode](#)

## See also

1. [Action Bar](#)
2. [Menu Resource](#)
3. [Say Goodbye to the Menu Button](#)

Menus are a common user interface component in many types of applications. To provide a familiar and consistent user experience, you should use the [Menu](#) APIs to present user actions and other options in your activities.

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 action 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. This guide shows how to create the three fundamental types of menus or action presentations on all versions of Android:

### Options menu and action bar

The [options menu](#) is the primary collection of menu items for an activity. It's where you should place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."

If you're developing for Android 2.3 or lower, users can reveal the options menu panel by pressing the *Menu* button.

On Android 3.0 and higher, items from the options menu are presented by the [action bar](#) as a combination of on-screen action items and overflow options. Beginning with Android 3.0, the *Menu* button is deprecate-

ed (some devices don't have one), so you should migrate toward using the action bar to provide access to actions and other options.

See the section about [Creating an Options Menu](#).

## 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.

When developing for Android 3.0 and higher, you should instead use the [contextual action mode](#) to enable actions on selected content. This 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.

See the section about [Creating Contextual Menus](#).

## 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.

See the section about [Creating a Popup Menu](#).

# 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 your activity's code, you should define a menu and all its items in an XML [menu resource](#). You can then inflate the menu resource (load it as a [Menu](#) object) in your 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 your 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 you to categorize menu items so they share properties such as active state and visibility. For more information, see the section about [Creating Menu Groups](#).

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 you 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 can 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 [action bar](#).

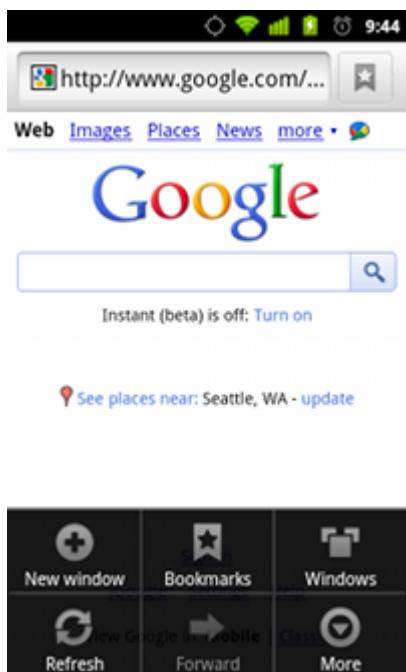
These are the most important attributes you should use, but there are many more available. For information about all the supported attributes, see the [Menu Resource](#) document.

You can add a submenu to an item in any menu (except a submenu) by adding a `<menu>` element as the child of an `<item>`. Submenus are useful when your application has a lot of functions that can be organized into topics, like items in a PC application's menu bar (File, Edit, View, etc.). For example:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/file"
          android:title="@string/file" >
        <!-- "file" submenu -->
        <menu>
            <item android:id="@+id/create_new"
                  android:title="@string/create_new" />
            <item android:id="@+id/open"
                  android:title="@string/open" />
        </menu>
    </item>
</menu>
```

To use the menu in your activity, you need to inflate the menu resource (convert the XML resource into a programmable object) using [MenuInflater.inflate\(\)](#). In the following sections, you'll see how to inflate a menu for each menu type.

# Creating an Options Menu



**Figure 1.** Options menu in the Browser, on Android 2.3.

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 your options menu appear on the screen depends on the version for which you've developed your application:

- If you've developed your application for **Android 2.3.x (API level 10) or lower**, the contents of your options menu appear at the bottom 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 your 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 you've developed your application for **Android 3.0 (API level 11) and higher**, items from the options menu are available in the [action 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 action bar (or by pressing the device *Menu* button, if available). To enable quick access to important actions, you can promote a few items to appear in the action bar by adding `android:showAsAction="ifRoom"` to the corresponding `<item>` elements (see figure 2).

For more information about action items and other action bar behaviors, see the [Action Bar](#) guide.

**Note:** Even if you're *not* developing for Android 3.0 or higher, you can build your own action bar layout for a similar effect. For an example of how you can support older versions of Android with an action bar, see the [Action Bar Compatibility](#) sample.



**Figure 2.** Action bar from the [Honeycomb Gallery](#) app, showing navigation tabs and a camera action item (plus the action overflow button).

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. For example:

```
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    MenuInflater inflater = getMenuInflater\(\);  
    inflater.inflate(R.menu.game_menu, menu);  
    return true;  
}
```

You can also add menu items using `add()` and retrieve items with `findItem()` to revise their properties with [MenuItem](#) APIs.

If you've developed your application for Android 2.3.x and lower, the system calls `onCreateOptionsMenu()` to create the options menu when the user opens the menu for the first time. If you've developed for Android 3.0 and higher, the system calls `onCreateOptionsMenu()` when starting the activity, in order to show items to the action bar.

## Handling click events

When the user selects an item from the options menu (including action items in the action bar), the system calls your activity's `onOptionsItemSelected()` method. This method passes the [MenuItem](#) selected. You 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). You can match this ID against known menu items to perform the appropriate action. For example:

```
@Override  
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);
    }
}

```

When you successfully handle a menu item, return `true`. If you don't handle the menu item, you should call the superclass implementation of [onOptionsItemSelected\(\)](#) (the default implementation returns false).

If your activity includes fragments, the system first calls [onOptionsItemSelected\(\)](#) for the activity then for each fragment (in the order each fragment was added) until one returns `true` or all fragments have been called.

**Tip:** Android 3.0 adds the ability for you to define the on-click behavior for a menu item in XML, using the `android:onClick` attribute. The value for the attribute must be the name of a method defined by the activity using the menu. The method must be public and accept a single [MenuItem](#) parameter—when the system calls this method, it passes the menu item selected. For more information and an example, see the [Menu Resource](#) document.

**Tip:** If your application contains multiple activities and some of them provide the same options menu, consider creating an activity that implements nothing except the [onCreateOptionsMenu\(\)](#) and [onOptionsItemSelected\(\)](#) methods. Then extend this class for each activity that should share the same options menu. This way, you can manage one set of code for handling menu actions and each descendant class inherits the menu behaviors. If you want to add menu items to one of the descendant activities, override [onCreateOptionsMenu\(\)](#) in that activity. Call `super.onCreateOptionsMenu(menu)` so the original menu items are created, then add new menu items with [menu.add\(\)](#). You can also override the super class's behavior for individual menu items.

## Changing menu items at runtime

After the system calls [onCreateOptionsMenu\(\)](#), it retains an instance of the [Menu](#) you populate and will not call [onCreateOptionsMenu\(\)](#) again unless the menu is invalidated for some reason. However, you should use [onCreateOptionsMenu\(\)](#) only to create the initial menu state and not to make changes during the activity lifecycle.

If you want to modify the options menu based on events that occur during the activity lifecycle, you can do so in the [onPrepareOptionsMenu\(\)](#) method. This method passes you the [Menu](#) object as it currently exists so you can modify it, such as add, remove, or disable items. (Fragments also provide an [onPrepareOptionsMenu\(\)](#) callback.)

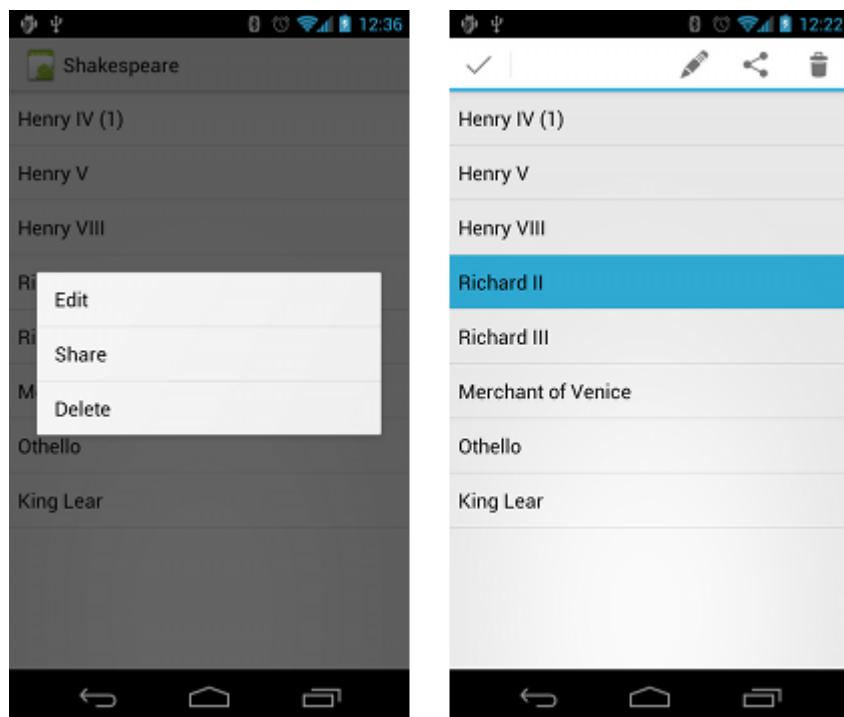
On Android 2.3.x and lower, the system calls [onPrepareOptionsMenu\(\)](#) each time the user opens the options menu (presses the *Menu* button).

On Android 3.0 and higher, the options menu is considered to always be open when menu items are presented in the action bar. When an event occurs and you want to perform a menu update, you must call [invalidateOptionsMenu\(\)](#) to request that the system call [onPrepareOptionsMenu\(\)](#).

**Note:** You should never change items in the options menu based on the [View](#) currently in focus. When in touch mode (when the user is not using a trackball or d-pad), views cannot take focus, so you should never use focus

as the basis for modifying items in the options menu. If you want to provide menu items that are context-sensitive to a [View](#), use a [Context Menu](#).

## Creating Contextual Menus



**Figure 3.** Screenshots of a floating context menu (left) and the contextual action bar (right).

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 [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).

**Note:** The contextual action mode is available on Android 3.0 (API level 11) and higher and is the preferred technique for displaying contextual actions when available. If your app supports versions lower than 3.0 then you should fall back to a floating context menu on those devices.

### 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. For example:

```
@Override  
public void onCreateContextMenu(ContextMenu menu, View v,  
                               ContextMenuItemInfo menuInfo) {  
    super.onCreateContextMenu(menu, v, menuInfo);  
    MenuInflater inflater = getMenuInflater();  
    inflater.inflate(R.menu.context_menu, menu);  
}
```

[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.ContextMenuItemInfo](#) 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.

## 3. Implement [onContextItemSelected\(\)](#).

When the user selects a menu item, the system calls this method so you can perform the appropriate action. For example:

```
@Override  
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);  
    }  
}
```

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](#).

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.

**Note:** The contextual action bar is not necessarily associated with the [action bar](#). They operate independently, even though the contextual action bar visually overtakes the action bar position.

If you're developing for Android 3.0 (API level 11) or higher, you should usually use the contextual action mode to present contextual actions, instead of the [floating context menu](#).

For views that provide contextual actions, you should usually invoke the contextual action mode upon one of two events (or both):

- The user performs a long-click on the view.
- The user selects a checkbox or similar UI component within the view.

How your application invokes the contextual action mode and defines the behavior for each action depends on your design. There are basically two designs:

- For contextual actions on individual, arbitrary views.
- For batch contextual actions on groups of items in a [ListView](#) or [GridView](#) (allowing the user to select multiple items and perform an action on them all).

The following sections describe the setup required for each scenario.

### Enabling the contextual action mode for individual views

If you want to invoke the contextual action mode only when the user selects specific views, you should:

1. Implement the [ActionMode.Callback](#) interface. In its callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other lifecycle events for the action mode.
2. Call [startActionMode\(\)](#) when you want to show the bar (such as when the user long-clicks the view).

For example:

1. Implement the [ActionMode.Callback](#) interface:

```
private ActionMode.Callback mActionModeCallback = new ActionMode.Callback() {  
  
    // Called when the action mode is created; startActionMode() was called  
    @Override  
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {  
        // Inflate a menu resource providing context menu items  
        MenuInflater inflater = mode.getMenuInflater();  
        inflater.inflate(R.menu.context_menu, menu);  
        return true;  
    }  
}
```

```

}

// Called each time the action mode is shown. Always called after onC
// may be called multiple times if the mode is invalidated.
@Override
public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
    return false; // Return false if nothing is done
}

// Called when the user selects a contextual menu item
@Override
public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_share:
            shareCurrentItem();
            mode.finish(); // Action picked, so close the CAB
            return true;
        default:
            return false;
    }
}

// Called when the user exits the action mode
@Override
public void onDestroyActionMode(ActionMode mode) {
    mActionMode = null;
}
};


```

Notice that these event callbacks are almost exactly the same as the callbacks for the [options menu](#), except each of these also pass the [ActionMode](#) object associated with the event. You can use [ActionMode](#) APIs to make various changes to the CAB, such as revise the title and subtitle with [setTitle\(\)](#) and [setSubtitle\(\)](#) (useful to indicate how many items are selected).

Also notice that the above sample sets the `mActionMode` variable null when the action mode is destroyed. In the next step, you'll see how it's initialized and how saving the member variable in your activity or fragment can be useful.

2. Call [startActionMode\(\)](#) to enable the contextual action mode when appropriate, such as in response to a long-click on a [View](#):

```

someView.setOnLongClickListener(new View.OnLongClickListener() {
    // Called when the user long-clicks on someView
    public boolean onLongClick(View view) {
        if (mActionMode != null) {
            return false;
        }

        // Start the CAB using the ActionMode.Callback defined above
        mActionMode = getActivity().startActionMode(mActionModeCallback);
        view.setSelected(true);
        return true;
    }
}) ;

```

When you call `startActionMode()`, the system returns the `ActionMode` created. By saving this in a member variable, you can make changes to the contextual action bar in response to other events. In the above sample, the `ActionMode` is used to ensure that the `ActionMode` instance is not recreated if it's already active, by checking whether the member is null before starting the action mode.

## Enabling batch contextual actions in a ListView or GridView

If you have a collection of items in a `ListView` or `GridView` (or another extension of `AbsListView`) and want to allow users to perform batch actions, you should:

- Implement the `AbsListView.MultiChoiceModeListener` interface and set it for the view group with `setMultiChoiceModeListener()`. In the listener's callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other callbacks inherited from the `ActionMode.Callback` interface.
- Call `setChoiceMode()` with the `CHOICE_MODE_MULTIPLE_MODAL` argument.

For example:

```
ListView listView = getListView();
listView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
listView.setMultiChoiceModeListener(new MultiChoiceModeListener() {

    @Override
    public void onItemCheckedStateChanged(ActionMode mode, int position,
                                         long id, boolean checked) {
        // Here you can do something when items are selected/de-selected,
        // such as update the title in the CAB
    }

    @Override
    public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
        // Respond to clicks on the actions in the CAB
        switch (item.getItemId()) {
            case R.id.menu_delete:
                deleteSelectedItems();
                mode.finish(); // Action picked, so close the CAB
                return true;
            default:
                return false;
        }
    }

    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        // Inflate the menu for the CAB
        MenuInflater inflater = mode.getMenuInflater();
        inflater.inflate(R.menu.context, menu);
        return true;
    }

    @Override
    public void onDestroyActionMode(ActionMode mode) {
        // Here you can make any necessary updates to the activity when
        // the CAB is removed. By default, selected items are deselected/unchecked
    }
})
```

```

    }

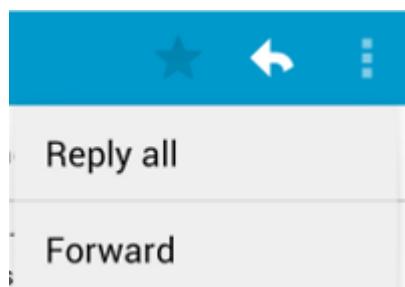
    @Override
    public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
        // Here you can perform updates to the CAB due to
        // an invalidate\(\) request
        return false;
    }
);

```

That's it. Now when the user selects an item with a long-click, the system calls the [onCreateActionMode \(\)](#) method and displays the contextual action bar with the specified actions. While the contextual action bar is visible, users can select additional items.

In some cases in which the contextual actions provide common action items, you might want to add a checkbox or a similar UI element that allows users to select items, because they might not discover the long-click behavior. When a user selects the checkbox, you can invoke the contextual action mode by setting the respective list item to the checked state with [setItemChecked \(\)](#).

## Creating a Popup Menu



**Figure 4.** A popup menu in the Gmail app, anchored to the overflow button at the top-right.

A [PopupMenu](#) is a modal menu anchored to a [View](#). It appears below the anchor view if there is room, or above the view otherwise. It's useful for:

- Providing an overflow-style menu for actions that *relate to* specific content (such as Gmail's email headers, shown in figure 4).

**Note:** This is not the same as a context menu, which is generally for actions that *affect* selected content. For actions that affect selected content, use the [contextual action mode](#) or [floating context menu](#).

- 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.

**Note:** [PopupMenu](#) is available with API level 11 and higher.

If you [define your menu in XML](#), here's how you 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 \(\)](#). On API level 14 and above, you can use [PopupMenu.inflate \(\)](#) instead.
3. Call [PopupMenu.show \(\)](#).

For example, here's a button with the [android:onClick](#) attribute that shows a popup menu:

```
<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, you 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. You can listen for the dismiss event using [PopupMenu.OnDismissListener](#).

## Handling click events

To perform an action when the user selects a menu item, you must implement the [PopupMenu.OnMenuItemClickListener](#) interface and register it with your [PopupMenu](#) by calling [setOnMenuItemClickListener\(\)](#). When the user selects an item, the system calls the [onMenuItemClick\(\)](#) callback in your interface.

For example:

```
public void showMenu(View v) {  
    PopupMenu popup = new PopupMenu(this, v);  
  
    // This activity implements OnMenuItemClickListener  
    popup.setOnMenuItemClickListener(this);  
    popup.inflate(R.menu.actions);  
    popup.show();  
}  
  
@Override  
public boolean onMenuItemClick(MenuItem item) {  
    switch (item.getItemId()) {  
        case R.id.archive:  
            archive(item);  
            return true;  
        case R.id.delete:  
            delete(item);  
            return true;  
        default:  
            return false;
```

```
}
```

## Creating Menu Groups

A menu group is a collection of menu items that share certain traits. With a group, you can:

- Show or hide all items with [setGroupVisible\(\)](#)
- Enable or disable all items with [setGroupEnabled\(\)](#)
- Specify whether all items are checkable with [setGroupCheckable\(\)](#)

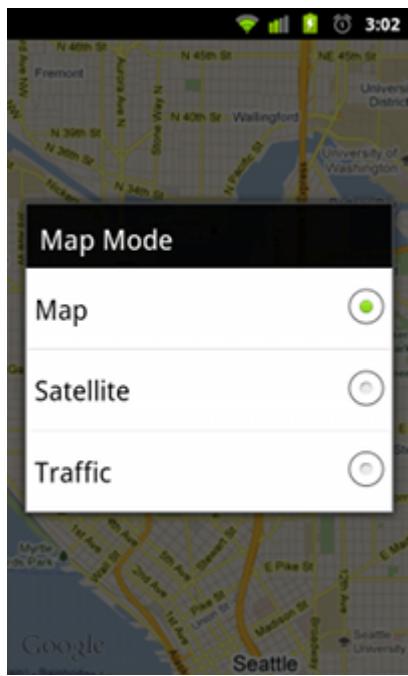
You can create a group by nesting `<item>` elements inside a `<group>` element in your menu resource or by specifying a group ID with the [add\(\)](#) method.

Here's an example menu resource that includes a group:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_save"
          android:icon="@drawable/menu_save"
          android:title="@string/menu_save" />
    <!-- menu group -->
    <group android:id="@+id/group_delete">
        <item android:id="@+id/menu_archive"
              android:title="@string/menu_archive" />
        <item android:id="@+id/menu_delete"
              android:title="@string/menu_delete" />
    </group>
</menu>
```

The items that are in the group appear at the same level as the first item—all three items in the menu are siblings. However, you can modify the traits of the two items in the group by referencing the group ID and using the methods listed above. The system will also never separate grouped items. For example, if you declare `android:showAsAction="ifRoom"` for each item, they will either both appear in the action bar or both appear in the action overflow.

## 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.

**Note:** Menu items in the Icon Menu (from the options menu) cannot display a checkbox or radio button. If you choose to make items in the Icon Menu checkable, you must manually indicate the checked state by swapping the icon and/or text each time the state changes.

You 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

You 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 your respective item-selected callback method (such as [onOptionsItemSelected\(\)](#)). It is here that you must set the state of the checkbox, because a checkbox or radio button does not change its state automatically. You 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\(\)](#). For example:

```
@Override  
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);  
    }  
}
```

If you don't set the checked state this way, then the visible state of the item (the checkbox or radio button) will not change when the user selects it. When you do set the state, the activity preserves the checked state of the item so that when the user opens the menu later, the checked state that you set is visible.

**Note:** Checkable menu items are intended to be used only on a per-session basis and not saved after the application is destroyed. If you have application settings that you would like to save for the user, you should store the data using [Shared Preferences](#).

## Adding Menu Items Based on an Intent

Sometimes you'll want a menu item to launch an activity using an [Intent](#) (whether it's an activity in your application or another application). When you know the intent you want to use and have a specific menu item that should initiate the intent, you can execute the intent with [startActivity\(\)](#) during the appropriate on-item-selected callback method (such as the [onOptionsItemSelected\(\)](#) callback).

However, if you are not certain that the user's device contains an application that handles the intent, then adding a menu item that invokes it can result in a non-functioning menu item, because the intent might not resolve to an activity. To solve this, Android lets you dynamically add menu items to your menu when Android finds activities on the device that handle your intent.

To add menu items based on available activities that accept an intent:

1. Define an intent with the category [CATEGORY\\_ALTERNATIVE](#) and/or [CATEGORY\\_SELECTED\\_ALTERNATIVE](#), plus any other requirements.
2. Call [Menu.addIntentOptions\(\)](#). Android then searches for any applications that can perform the intent and adds them to your menu.

If there are no applications installed that satisfy the intent, then no menu items are added.

**Note:** [CATEGORY\\_SELECTED\\_ALTERNATIVE](#) is used to handle the currently selected element on the screen. So, it should only be used when creating a Menu in [onCreateContextMenu\(\)](#).

For example:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);

    // Create an Intent that describes the requirements to fulfill, to be included
    // in our menu. The offering app must include a category value of Intent.CATEGORY_ALTERNATIVE.
    Intent intent = new Intent(null, dataUri);
    intent.addCategory(Intent.CATEGORY_ALTERNATIVE);

    // Search and populate the menu with acceptable offering applications.
    menu.addIntentOptions(
        R.id.intent_group, // Menu group to which new items will be added
        0,                // Unique item ID (none)
        0,                // Order for the items (none)
        this.getComponentName(), // The current activity name
        null,              // Specific items to place first (none)
        intent,             // Intent created above that describes our requirements
        0,                // Additional flags to control items (none)
        null);             // Array of MenuItem objects that correlate to specific items (none)

    return true;
}
```

For each activity found that provides an intent filter matching the intent defined, a menu item is added, using the value in the intent filter's `android:label` as the menu item title and the application icon as the menu item icon. The [addIntentOptions\(\)](#) method returns the number of menu items added.

**Note:** When you call [addIntentOptions\(\)](#), it overrides any and all menu items by the menu group specified in the first argument.

## Allowing your activity to be added to other menus

You can also offer the services of your activity to other applications, so your application can be included in the menu of others (reverse the roles described above).

To be included in other application menus, you need to define an intent filter as usual, but be sure to include the `CATEGORY_ALTERNATIVE` and/or `CATEGORY_SELECTED_ALTERNATIVE` values for the intent filter category. For example:

```
<intent-filter label="@string/resize_image">
    ...
    <category android:name="android.intent.category.ALTERNATIVE" />
    <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
    ...
</intent-filter>
```

Read more about writing intent filters in the [Intents and Intent Filters](#) document.

For a sample application using this technique, see the [Note Pad](#) sample code.

# Action Bar

## Design Guide

### Action Bar

## In this document

1. [Adding the Action Bar](#)
  1. [Removing the action bar](#)
  2. [Using a logo instead of an icon](#)
2. [Adding Action Items](#)
  1. [Handling clicks on action items](#)
  2. [Using split action bar](#)
3. [Navigating Up with the App Icon](#)
4. [Adding an Action View](#)
  1. [Handling collapsible action views](#)
5. [Adding an Action Provider](#)
  1. [Using the ShareActionProvider](#)
  2. [Creating a custom action provider](#)
6. [Adding Navigation Tabs](#)
7. [Adding Drop-down Navigation](#)
8. [Styling the Action Bar](#)
  1. [General appearance](#)
  2. [Action items](#)
  3. [Navigation tabs](#)
  4. [Drop-down lists](#)
  5. [Example theme](#)

## Key classes

1. [ActionBar](#)
2. [Menu](#)

The action bar is a window feature that identifies the user location, and provides user actions and navigation modes. Using the action bar offers your users a familiar interface across applications that the system gracefully adapts for different screen configurations.



**Figure 1.** An action bar that includes the [1] app icon, [2] two action items, and [3] action overflow.

The action bar provides several key functions:

- Provides a dedicated space for giving your app an identity and indicating the user's location in the app.
- Makes important actions prominent and accessible in a predictable way (such as *Search*).
- Supports consistent navigation and view switching within apps (with tabs or drop-down lists).

For more information about the action bar's interaction patterns and design guidelines, see the [Action Bar](#) design guide.

The [ActionBar](#) APIs were first added in Android 3.0 (API level 11) but they are also available in the [Support Library](#) for compatibility with Android 2.1 (API level 7) and above.

**This guide focuses on how to use the support library's action bar**, but if your app supports *only* Android 3.0 or higher, you should use the [ActionBar](#) APIs in the framework. Most of the APIs are the same—but reside in a different package namespace—with a few exceptions to method names or signatures that are noted in the sections below.

**Caution:** Be certain you import the `ActionBar` class (and related APIs) from the appropriate package:

- If supporting API levels *lower than* 11:  
`import android.support.v7.app.ActionBar`
- If supporting *only* API level 11 and higher:  
`import android.app.ActionBar`

**Note:** If you're looking for information about the *contextual action bar* for displaying contextual action items, see the [Menu](#) guide.

## Adding the Action Bar

As mentioned above, this guide focuses on how to use the [ActionBar](#) APIs in the support library. So before you can add the action bar, you must set up your project with the **appcompat v7** support library by following the instructions in the [Support Library Setup](#).

Once your project is set up with the support library, here's how to add the action bar:

1. Create your activity by extending [ActionBarActivity](#).
2. Use (or extend) one of the [Theme.AppCompat](#) themes for your activity. For example:

```
<activity android:theme="@style/Theme.AppCompat.Light" ... >
```

Now your activity includes the action bar when running on Android 2.1 (API level 7) or higher.

### On API level 11 or higher

The action bar is included in all activities that use the [Theme.Holo](#) theme (or one of its descendants), which is the default theme when either the [targetSdkVersion](#) or [minSdkVersion](#) attribute is set to "11" or higher. If you don't want the action bar for an activity, set the activity theme to [Theme.Holo.NoActionBar](#).

## Removing the action bar

You can hide the action bar at runtime by calling [hide\(\)](#). For example:

```
ActionBar actionBar = getSupportActionBar();  
actionBar.hide();
```

## On API level 11 or higher

Get the [ActionBar](#) with the [getActionBar\(\)](#) method.

When the action bar hides, the system adjusts your layout to fill the screen space now available. You can bring the action bar back by calling [show\(\)](#).

Beware that hiding and removing the action bar causes your activity to re-layout in order to account for the space consumed by the action bar. If your activity often hides and shows the action bar, you might want to enable *overlay mode*. Overlay mode draws the action bar in front of your activity layout, obscuring the top portion. This way, your layout remains fixed when the action bar hides and reappears. To enable overlay mode, create a custom theme for your activity and set [windowActionBarOverlay](#) to true. For more information, see the section below about [Styling the Action Bar](#).

## Using a logo instead of an icon

By default, the system uses your application icon in the action bar, as specified by the [icon](#) attribute in the [<application>](#) or [<activity>](#) element. However, if you also specify the [logo](#) attribute, then the action bar uses the logo image instead of the icon.

A logo should usually be wider than the icon, but should not include unnecessary text. You should generally use a logo only when it represents your brand in a traditional format that users recognize. A good example is the YouTube app's logo—the logo represents the expected user brand, whereas the app's icon is a modified version that conforms to the square requirement for the launcher icon.

## Adding Action Items



**Figure 2.** Action bar with three action buttons and the overflow button.

The action bar provides users access to the most important action items relating to the app's current context. Those that appear directly in the action bar with an icon and/or text are known as *action buttons*. Actions that can't fit in the action bar or aren't important enough are hidden in the action overflow. The user can reveal a list of the other actions by pressing the overflow button on the right side (or the device *Menu* button, if available).

When your activity starts, the system populates the action items by calling your activity's [onCreateOptionsMenu\(\)](#) method. Use this method to inflate a [menu resource](#) that defines all the action items. For example, here's a menu resource defining a couple of menu items:

res/menu/main\_activity\_actions.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >  
    <item android:id="@+id/action_search"  
          android:icon="@drawable/ic_action_search"  
          android:title="@string/action_search"/>  
    <item android:id="@+id/action_compose"  
          android:icon="@drawable/ic_action_compose"
```

```
        android:title="@string/action_compose" />
</menu>
```

Then in your activity's [onCreateOptionsMenu\(\)](#) method, inflate the menu resource into the given [Menu](#) to add each item to the action bar:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu items for use in the action bar
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main_activity_actions, menu);
    return super.onCreateOptionsMenu(menu);
}
```

To request that an item appear directly in the action bar as an action button, include `showAsAction="ifRoom"` in the `<item>` tag. For example:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:yourapp="http://schemas.android.com/apk/res-auto" >
    <item android:id="@+id/action_search"
          android:icon="@drawable/ic_action_search"
          android:title="@string/action_search"
          yourapp:showAsAction="ifRoom" />
    ...
</menu>
```

If there's not enough room for the item in the action bar, it will appear in the action overflow.

## Using XML attributes from the support library

Notice that the `showAsAction` attribute above uses a custom namespace defined in the `<menu>` tag. This is necessary when using any XML attributes defined by the support library, because these attributes do not exist in the Android framework on older devices. So you must use your own namespace as a prefix for all attributes defined by the support library.

If your menu item supplies both a title and an icon—with the `title` and `icon` attributes—then the action item shows only the icon by default. If you want to display the text title, add "withText" to the `showAsAction` attribute. For example:

```
<item yourapp:showAsAction="ifRoom|withText" ... />
```

**Note:** The "withText" value is a *hint* to the action bar that the text title should appear. The action bar will show the title when possible, but might not if an icon is available and the action bar is constrained for space.

You should always define the `title` for each item even if you don't declare that the title appear with the action item, for the following reasons:

- If there's not enough room in the action bar for the action item, the menu item appears in the overflow where only the title appears.
- Screen readers for sight-impaired users read the menu item's title.
- If the action item appears with only the icon, a user can long-press the item to reveal a tool-tip that displays the action title.

The icon is optional, but recommended. For icon design recommendations, see the [Iconography](#) design guide. You can also download a set of standard action bar icons (such as for Search or Discard) from the [Downloads](#) page.

You can also use "always" to declare that an item always appear as an action button. However, you **should not** force an item to appear in the action bar this way. Doing so can create layout problems on devices with a narrow screen. It's best to instead use "ifRoom" to request that an item appear in the action bar, but allow the system to move it into the overflow when there's not enough room. However, it might be necessary to use this value if the item includes an [action view](#) that cannot be collapsed and must always be visible to provide access to a critical feature.

## Handling clicks on action items

When the user presses an action, the system calls your activity's [onOptionsItemSelected\(\)](#) method. Using the [MenuItem](#) passed to this method, you can identify the action by calling [getItemId\(\)](#). This returns the unique ID provided by the <item> tag's id attribute so you can perform the appropriate action. For example:

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    // Handle presses on the action bar items  
    switch (item.getItemId()) {  
        case R.id.action_search:  
            openSearch();  
            return true;  
        case R.id.action_compose:  
            composeMessage();  
            return true;  
        default:  
            return super.onOptionsItemSelected(item);  
    }  
}
```

**Note:** If you inflate menu items from a fragment, via the [Fragment](#) class's [onCreateOptionsMenu\(\)](#) callback, the system calls [onOptionsItemSelected\(\)](#) for that fragment when the user selects one of those items. However, the activity gets a chance to handle the event first, so the system first calls [onOptionsItemSelected\(\)](#) on the activity, before calling the same callback for the fragment. To ensure that any fragments in the activity also have a chance to handle the callback, always pass the call to the superclass as the default behavior instead of returning false when you do not handle the item.



**Figure 3.** Mock-ups showing an action bar with tabs (left), then with split action bar (middle); and with the app icon and title disabled (right).

## Using split action bar

Split action bar provides a separate bar at the bottom of the screen to display all action items when the activity is running on a narrow screen (such as a portrait-oriented handset).

Separating the action items this way ensures that a reasonable amount of space is available to display all your action items on a narrow screen, while leaving room for navigation and title elements at the top.

To enable split action bar when using the support library, you must do two things:

1. Add `uiOptions="splitActionBarWhenNarrow"` to each `<activity>` element or to the `<application>` element. This attribute is understood only by API level 14 and higher (it is ignored by older versions).
2. To support older versions, add a `<meta-data>` element as a child of each `<activity>` element that declares the same value for `"android.support.UI_OPTIONS"`.

For example:

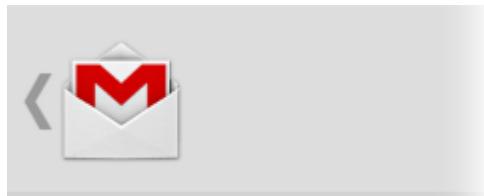
```
<manifest ...>
    <activity uiOptions="splitActionBarWhenNarrow" ... >
        <meta-data android:name="android.support.UI_OPTIONS"
                  android:value="splitActionBarWhenNarrow" />
    </activity>
</manifest>
```

Using split action bar also allows [navigation tabs](#) to collapse into the main action bar if you remove the icon and title (as shown on the right in figure 3). To create this effect, disable the action bar icon and title with [`setDisplayHomeAsUpEnabled\(false\)`](#) and [`setDisplayShowTitleEnabled\(false\)`](#).

## Navigating Up with the App Icon

### Design Guide

#### Navigation with Back and Up



**Figure 4.** The *Up* button in Gmail.

Enabling the app icon as an *Up* button allows the user to navigate your app based on the hierarchical relationships between screens. For instance, if screen A displays a list of items, and selecting an item leads to screen B, then screen B should include the *Up* button, which returns to screen A.

**Note:** Up navigation is distinct from the back navigation provided by the system *Back* button. The *Back* button is used to navigate in reverse chronological order through the history of screens the user has recently worked

with. It is generally based on the temporal relationships between screens, rather than the app's hierarchy structure (which is the basis for up navigation).

To enable the app icon as an *Up* button, call [setDisplayHomeAsUpEnabled\(\)](#). For example:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_details);  
  
    ActionBar actionBar = getSupportActionBar();  
    actionBar.setDisplayHomeAsUpEnabled(true);  
    ...  
}
```

Now the icon in the action bar appears with the *Up* caret (as shown in figure 4). However, it won't do anything by default. To specify the activity to open when the user presses *Up* button, you have two options:

- **Specify the parent activity in the manifest file.**

This is the best option when **the parent activity is always the same**. By declaring in the manifest which activity is the parent, the action bar automatically performs the correct action when the user presses the *Up* button.

Beginning in Android 4.1 (API level 16), you can declare the parent with the [parentActivityName](#) attribute in the [<activity>](#) element.

To support older devices with the support library, also include a [<meta-data>](#) element that specifies the parent activity as the value for `android.support.PARENT_ACTIVITY`. For example:

```
<application ... >  
    ...  
    <!-- The main/home activity (has no parent activity) -->  
    <activity  
        android:name="com.example.myfirstapp.MainActivity" ...>  
        ...  
    </activity>  
    <!-- A child of the main activity -->  
    <activity  
        android:name="com.example.myfirstapp.DisplayMessageActivity"  
        android:label="@string/title_activity_display_message"  
        android:parentActivityName="com.example.myfirstapp.MainActivity"  
        <!-- Parent activity meta-data to support API level 7+ -->  
        <meta-data  
            android:name="android.support.PARENT_ACTIVITY"  
            android:value="com.example.myfirstapp.MainActivity" />  
    </activity>  
</application>
```

Once the parent activity is specified in the manifest like this and you enable the *Up* button with [setDisplayHomeAsUpEnabled\(\)](#), your work is done and the action bar properly navigates up.

- **Or, override [getSupportActionBarParentIntent\(\)](#) and [onCreateSupportNavigateUpTaskStack\(\)](#) in your activity.**

This is appropriate when **the parent activity may be different** depending on how the user arrived at the current screen. That is, if there are many paths that the user could have taken to reach the current screen, the *Up* button should navigate backward along the path the user actually followed to get there.

The system calls [getSupportParentActivityIntent\(\)](#) when the user presses the *Up* button while navigating your app (within your app's own task). If the activity that should open upon up navigation differs depending on how the user arrived at the current location, then you should override this method to return the [Intent](#) that starts the appropriate parent activity.

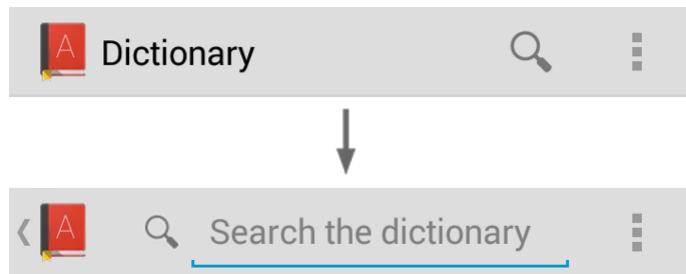
The system calls [onCreateSupportNavigateUpTaskStack\(\)](#) for your activity when the user presses the *Up* button while your activity is running in a task that does *not* belong to your app. Thus, you must use the [TaskStackBuilder](#) passed to this method to construct the appropriate back stack that should be synthesized when the user navigates up.

Even if you override [getSupportParentActivityIntent\(\)](#) to specify up navigation as the user navigates your app, you can avoid the need to implement [onCreateSupportNavigateUpTaskStack\(\)](#) by declaring "default" parent activities in the manifest file as shown above. Then the default implementation of [onCreateSupportNavigateUpTaskStack\(\)](#) will synthesize a back stack based on the parent activities declared in the manifest.

**Note:** If you've built your app hierarchy using a series of fragments instead of multiple activities, then neither of the above options will work. Instead, to navigate up through your fragments, override [onSupportNavigateUp\(\)](#) to perform the appropriate fragment transaction—usually by popping the current fragment from the back stack by calling [popBackStack\(\)](#).

For more information about implementing *Up* navigation, read [Providing Up Navigation](#).

## Adding an Action View



**Figure 5.** An action bar with a collapsible [SearchView](#).

An *action view* is a widget that appears in the action bar as a substitute for an action button. An action view provides fast access to rich actions without changing activities or fragments, and without replacing the action bar. For example, if you have an action for Search, you can add an action view to embed a [SearchView](#) widget in the action bar, as shown in figure 5.

To declare an action view, use either the `actionLayout` or `actionViewClass` attribute to specify either a layout resource or widget class to use, respectively. For example, here's how to add the [SearchView](#) widget:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:yourapp="http://schemas.android.com/apk/res-auto" >
    <item android:id="@+id/action_search"
          android:title="@string/action_search"
          android:icon="@drawable/ic_action_search"
```

```
        yourapp:showAsAction="ifRoom|collapseActionView"
        yourapp:actionViewClass="android.support.v7.widget.SearchView" />
</menu>
```

Notice that the `showAsAction` attribute also includes the `"collapseActionView"` value. This is optional and declares that the action view should be collapsed into a button. (This behavior is explained further in the following section about [Handling collapsible action views](#).)

If you need to configure the action view (such as to add event listeners), you can do so during the [`onCreateOptionsMenu\(\)`](#) callback. You can acquire the action view object by calling the static method [`MenuItemCompat.getActionView\(\)`](#) and passing it the corresponding [`MenuItem`](#). For example, the search widget from the above sample is acquired like this:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main_activity_actions, menu);
    MenuItem searchItem = menu.findItem(R.id.action_search);
    SearchView searchView = (SearchView) MenuItemCompat.getActionView(searchItem);
    // Configure the search info and add any event listeners
    ...
    return super.onCreateOptionsMenu(menu);
}
```

## On API level 11 or higher

Get the action view by calling [`getActionView\(\)`](#) on the corresponding [`MenuItem`](#):

```
menu.findItem(R.id.action_search).getActionView()
```

For more information about using the search widget, see [Creating a Search Interface](#).

## Handling collapsible action views

To preserve the action bar space, you can collapse your action view into an action button. When collapsed, the system might place the action into the action overflow, but the action view still appears in the action bar when the user selects it. You can make your action view collapsible by adding `"collapseActionView"` to the `showAsAction` attribute, as shown in the XML above.

Because the system expands the action view when the user selects the action, you *do not* need to respond to the item in the [`onOptionsItemSelected\(\)`](#) callback. The system still calls [`onOptionsItemSelected\(\)`](#), but if you return `true` (indicating you've handled the event instead), then the action view will *not* expand.

The system also collapses your action view when the user presses the *Up* button or *Back* button.

If you need to update your activity based on the visibility of your action view, you can receive callbacks when the action is expanded and collapsed by defining an [`OnActionExpandListener`](#) and passing it to [`setOnActionExpandListener\(\)`](#). For example:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.options, menu);
    MenuItem menuItem = menu.findItem(R.id.actionItem);
    ...
}
```

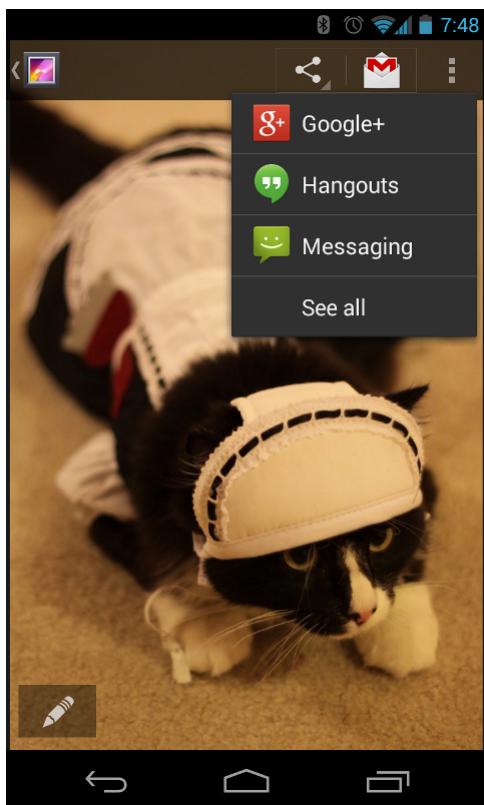
```

// When using the support library, the setOnActionExpandListener() method is
// static and accepts the MenuItem object as an argument
MenuItemCompat.setOnActionExpandListener(menuItem, new OnActionExpandListener()
    @Override
    public boolean onMenuItemActionCollapse(MenuItem item) {
        // Do something when collapsed
        return true; // Return true to collapse action view
    }

    @Override
    public boolean onMenuItemActionExpand(MenuItem item) {
        // Do something when expanded
        return true; // Return true to expand action view
    }
);
}

```

## Adding an Action Provider



**Figure 6.** An action bar with [ShareActionProvider](#) expanded to show share targets.

Similar to an [action view](#), an *action provider* replaces an action button with a customized layout. However, unlike an action view, an action provider takes control of all the action's behaviors and an action provider can display a submenu when pressed.

To declare an action provider, supply the `actionViewClass` attribute in the menu `<item>` tag with a fully-qualified class name for an [ActionProvider](#).

You can build your own action provider by extending the [ActionProvider](#) class, but Android provides some pre-built action providers such as [ShareActionProvider](#), which facilitates a "share" action by showing a list of possible apps for sharing directly in the action bar (as shown in figure 6).

Because each [ActionProvider](#) class defines its own action behaviors, you don't need to listen for the action in the [onOptionsItemSelected\(\)](#) method. If necessary though, you can still listen for the click event in the [onOptionsItemSelected\(\)](#) method in case you need to simultaneously perform another action. But be sure to return `false` so that the the action provider still receives the [onPerformDefaultAction\(\)](#) callback to perform its intended action.

However, if the action provider provides a submenu of actions, then your activity does not receive a call to [onOptionsItemSelected\(\)](#) when the user opens the list or selects one of the submenu items.

## Using the ShareActionProvider

To add a "share" action with [ShareActionProvider](#), define the `actionProviderClass` for an `<item>` tag with the [ShareActionProvider](#) class. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:yourapp="http://schemas.android.com/apk/res-auto" >
    <item android:id="@+id/action_share"
          android:title="@string/share"
          yourapp:showAsAction="ifRoom"
          yourapp:actionProviderClass="android.support.v7.widget.ShareActionProvider"
          />
    ...
</menu>
```

Now the action provider takes control of the action item and handles both its appearance and behavior. But you must still provide a title for the item to be used when it appears in the action overflow.

The only thing left to do is define the [Intent](#) you want to use for sharing. To do so, edit your [onCreateOptionsMenu\(\)](#) method to call [MenuItemCompat.getActionProvider\(\)](#) and pass it the [MenuItem](#) holding the action provider. Then call [setShareIntent\(\)](#) on the returned [ShareActionProvider](#) and pass it an [ACTION\\_SEND](#) intent with the appropriate content attached.

You should call [setShareIntent\(\)](#) once during [onCreateOptionsMenu\(\)](#) to initialize the share action, but because the user context might change, you must update the intent any time the shareable content changes by again calling [setShareIntent\(\)](#).

For example:

```
private ShareActionProvider mShareActionProvider;

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main_activity_actions, menu);

    // Set up ShareActionProvider's default share intent
    MenuItem shareItem = menu.findItem(R.id.action_share);
    mShareActionProvider = (ShareActionProvider)
        MenuItemCompat.getActionProvider(shareItem);
    mShareActionProvider.setShareIntent(getDefaultIntent());
```

```

        return super.onCreateOptionsMenu(menu);
    }

/** Defines a default (dummy) share intent to initialize the action provider.
 * However, as soon as the actual content to be used in the intent
 * is known or changes, you must update the share intent by again calling
 * mShareActionProvider.setShareIntent()
 */
private Intent getDefaultIntent() {
    Intent intent = new Intent(Intent.ACTION_SEND);
    intent.setType("image/*");
    return intent;
}

```

The [ShareActionProvider](#) now handles all user interaction with the item and you *do not* need to handle click events from the [onOptionsItemSelected\(\)](#) callback method.

By default, the [ShareActionProvider](#) retains a ranking for each share target based on how often the user selects each one. The share targets used more frequently appear at the top of the drop-down list and the target used most often appears directly in the action bar as the default share target. By default, the ranking information is saved in a private file with a name specified by [DEFAULT\\_SHARE\\_HISTORY\\_FILE\\_NAME](#). If you use the [ShareActionProvider](#) or an extension of it for only one type of action, then you should continue to use this default history file and there's nothing you need to do. However, if you use [ShareActionProvider](#) or an extension of it for multiple actions with semantically different meanings, then each [ShareActionProvider](#) should specify its own history file in order to maintain its own history. To specify a different history file for the [ShareActionProvider](#), call [setShareHistoryFileName\(\)](#) and provide an XML file name (for example, "custom\_share\_history.xml").

**Note:** Although the [ShareActionProvider](#) ranks share targets based on frequency of use, the behavior is extensible and extensions of [ShareActionProvider](#) can perform different behaviors and ranking based on the history file (if appropriate).

## Creating a custom action provider

Creating your own action provider allows you to re-use and manage dynamic action item behaviors in a self-contained module, rather than handle action item transformations and behaviors in your fragment or activity code. As shown in the previous section, Android already provides an implementation of [ActionProvider](#) for share actions: the [ShareActionProvider](#).

To create your own action provider for a different action, simply extend the [ActionProvider](#) class and implement its callback methods as appropriate. Most importantly, you should implement the following:

### [ActionProvider\(\)](#)

This constructor passes you the application [Context](#), which you should save in a member field to use in the other callback methods.

### [onCreateActionView\(MenuItem\)](#)

This is where you define the action view for the item. Use the [Context](#) acquired from the constructor to instantiate a [LayoutInflater](#) and inflate your action view layout from an XML resource, then hook up event listeners. For example:

```

public View onCreateActionView(MenuItem forIndexPath) {
    // Inflate the action view to be shown on the action bar.
    LayoutInflater layoutInflater = LayoutInflater.from(mContext);
    View view = layoutInflater.inflate(R.layout.action_provider, null);
    ImageButton button = (ImageButton) view.findViewById(R.id.button);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // Do something...
        }
    });
    return view;
}

```

### [onPerformDefaultAction\(\)](#)

The system calls this when the menu item is selected from the action overflow and the action provider should perform a default action for the menu item.

However, if your action provider provides a submenu, through the [onPrepareSubMenu\(\)](#) callback, then the submenu appears even when the action provider is placed in the action overflow. Thus, [onPerformDefaultAction\(\)](#) is never called when there is a submenu.

**Note:** An activity or a fragment that implements [onOptionsItemSelected\(\)](#) can override the action provider's default behavior (unless it uses a submenu) by handling the item-selected event (and returning true), in which case, the system does not call [onPerformDefaultAction\(\)](#).

For an example extension of [ActionProvider](#), see [ActionBarSettingsActionProviderActivity](#).

## Adding Navigation Tabs



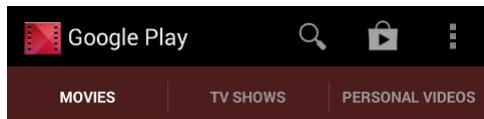
**Figure 7.** Action bar tabs on a wide screen.

## Design Guide

### Tabs

### Also read

#### [Creating Swipe Views with Tabs](#)



**Figure 8.** Tabs on a narrow screen.

Tabs in the action bar make it easy for users to explore and switch between different views in your app. The tabs provided by the [ActionBar](#) are ideal because they adapt to different screen sizes. For example, when the screen is wide enough the tabs appear in the action bar alongside the action buttons (such as when on a tablet,

shown in figure 7), while when on a narrow screen they appear in a separate bar (known as the "stacked action bar", shown in figure 8). In some cases, the Android system will instead show your tab items as a drop-down list to ensure the best fit in the action bar.

To get started, your layout must include a [ViewGroup](#) in which you place each [Fragment](#) associated with a tab. Be sure the [ViewGroup](#) has a resource ID so you can reference it from your code and swap the tabs within it. Alternatively, if the tab content will fill the activity layout, then your activity doesn't need a layout at all (you don't even need to call [setContentView\(\)](#)). Instead, you can place each fragment in the default root view, which you can refer to with the android.R.id.content ID.

Once you determine where the fragments appear in the layout, the basic procedure to add tabs is:

1. Implement the [ActionBar.TabListener](#) interface. This interface provides callbacks for tab events, such as when the user presses one so you can swap the tabs.
2. For each tab you want to add, instantiate an [ActionBar.Tab](#) and set the [ActionBar.TabListener](#) by calling [setTabListener\(\)](#). Also set the tab's title and with [setText\(\)](#) (and optionally, an icon with [setIcon\(\)](#)).
3. Then add each tab to the action bar by calling [addTab\(\)](#).

Notice that the [ActionBar.TabListener](#) callback methods don't specify which fragment is associated with the tab, but merely which [ActionBar.Tab](#) was selected. You must define your own association between each [ActionBar.Tab](#) and the appropriate [Fragment](#) that it represents. There are several ways you can define the association, depending on your design.

For example, here's how you might implement the [ActionBar.TabListener](#) such that each tab uses its own instance of the listener:

```
public static class TabListener<T extends Fragment> implements ActionBar.TabListener {
    private Fragment mFragment;
    private final Activity mActivity;
    private final String mTag;
    private final Class<T> mClass;

    /** Constructor used each time a new tab is created.
     * @param activity The host Activity, used to instantiate the fragment
     * @param tag The identifier tag for the fragment
     * @param clz The fragment's Class, used to instantiate the fragment
     */
    public TabListener(Activity activity, String tag, Class<T> clz) {
        mActivity = activity;
        mTag = tag;
        mClass = clz;
    }

    /* The following are each of the ActionBar.TabListener callbacks */

    public void onTabSelected(Tab tab, FragmentTransaction ft) {
        // Check if the fragment is already initialized
        if (mFragment == null) {
            // If not, instantiate and add it to the activity
            mFragment = Fragment.instantiate(mActivity, mClass.getName());
            ft.add(android.R.id.content, mFragment, mTag);
        } else {
            // If it exists, simply attach it in order to show it
        }
    }

    public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    }

    public void onTabReselected(Tab tab, FragmentTransaction ft) {
    }
}
```

```

        ft.attach(mFragment);
    }

}

public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    if (mFragment != null) {
        // Detach the fragment, because another one is being attached
        ft.detach(mFragment);
    }
}

public void onTabReselected(Tab tab, FragmentTransaction ft) {
    // User selected the already selected tab. Usually do nothing.
}
}

```

**Caution:** You **must not** call [commit\(\)](#) for the fragment transaction in each of these callbacks—the system calls it for you and it may throw an exception if you call it yourself. You also **cannot** add these fragment transactions to the back stack.

In this example, the listener simply attaches ([attach\(\)](#)) a fragment to the activity layout—or if not instantiated, creates the fragment and adds ([add\(\)](#)) it to the layout (as a child of the android.R.id.content view group)—when the respective tab is selected, and detaches ([detach\(\)](#)) it when the tab is unselected.

All that remains is to create each [ActionBar.Tab](#) and add it to the [ActionBar](#). Additionally, you must call [setNavigationMode\(NAVIGATION\\_MODE\\_TABS\)](#) to make the tabs visible.

For example, the following code adds two tabs using the listener defined above:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Notice that setContentView() is not used, because we use the root
    // android.R.id.content as the container for each fragment

    // setup action bar for tabs
    ActionBar actionBar = getSupportActionBar();
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
    actionBar.setDisplayShowTitleEnabled(false);

    Tab tab = actionBar.newTab()
        .setText(R.string.artist)
        .setTabListener(new TabListener<ArtistFragment>(
            this, "artist", ArtistFragment.class));
    actionBar.addTab(tab);

    tab = actionBar.newTab()
        .setText(R.string.album)
        .setTabListener(new TabListener<AlbumFragment>(
            this, "album", AlbumFragment.class));
    actionBar.addTab(tab);
}

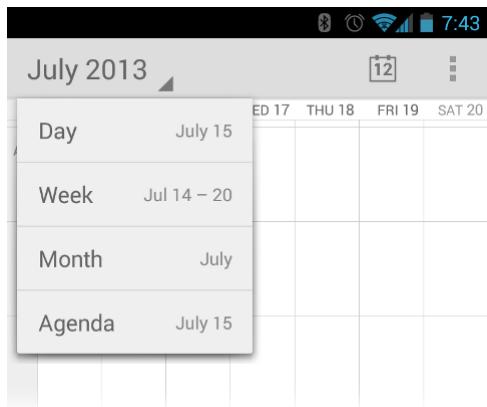
```

If your activity stops, you should retain the currently selected tab with the [saved instance state](#) so you can open the appropriate tab when the user returns. When it's time to save the state, you can query the currently selected tab with [getSelectedNavigationIndex\(\)](#). This returns the index position of the selected tab.

**Caution:** It's important that you save the state of each fragment so when users switch fragments with the tabs and then return to a previous fragment, it looks the way it did when they left. Some of the state is saved by default, but you may need to manually save state for customized views. For information about saving the state of your fragment, see the [Fragments API guide](#).

**Note:** The above implementation for [ActionBar.TabListener](#) is one of several possible techniques. Another popular option is to use [ViewPager](#) to manage the fragments so users can also use a swipe gesture to switch tabs. In this case, you simply tell the [ViewPager](#) the current tab position in the [onTabSelected\(\)](#) callback. For more information, read [Creating Swipe Views with Tabs](#).

## Adding Drop-down Navigation



**Figure 9.** A drop-down navigation list in the action bar.

As another mode of navigation (or filtering) for your activity, the action bar offers a built in drop-down list (also known as a "spinner"). For example, the drop-down list can offer different modes by which content in the activity is sorted.

Using the drop-down list is useful when changing the content is important but not necessarily a frequent occurrence. In cases where switching the content is more frequent, you should use [navigation tabs](#) instead.

The basic procedure to enable drop-down navigation is:

1. Create a [SpinnerAdapter](#) that provides the list of selectable items for the drop-down and the layout to use when drawing each item in the list.
2. Implement [ActionBar.OnNavigationListener](#) to define the behavior that occurs when the user selects an item from the list.
3. During your activity's [onCreate\(\)](#) method, enable the action bar's drop-down list by calling [setNavigationMode\(NAVIGATION\\_MODE\\_LIST\)](#).
4. Set the callback for the drop-down list with [setListNavigationCallbacks\(\)](#). For example:

```
actionBar.setListNavigationCallbacks(mSpinnerAdapter, mNavigationCallback)
```

This method takes your [SpinnerAdapter](#) and [ActionBar.OnNavigationListener](#).

This procedure is relatively short, but implementing the [SpinnerAdapter](#) and [ActionBar.OnNavigationListener](#) is where most of the work is done. There are many ways you can implement these to define the functionality for your drop-down navigation and implementing various types of

[SpinnerAdapter](#) is beyond the scope of this document (you should refer to the [SpinnerAdapter](#) class reference for more information). However, below is an example for a [SpinnerAdapter](#) and [ActionBar.OnNavigationListener](#) to get you started (click the title to reveal the sample).

## ► Example SpinnerAdapter and OnNavigationListener

[SpinnerAdapter](#) is an adapter that provides data for a spinner widget, such as the drop-down list in the action bar. [SpinnerAdapter](#) is an interface that you can implement, but Android includes some useful implementations that you can extend, such as [ArrayAdapter](#) and [SimpleCursorAdapter](#). For example, here's an easy way to create a [SpinnerAdapter](#) by using [ArrayAdapter](#) implementation, which uses a string array as the data source:

```
SpinnerAdapter mSpinnerAdapter = ArrayAdapter.createFromResource(this, R.array.  
    android.R.layout.simple_spinner_dropdown_item);
```

The [createFromResource\(\)](#) method takes three parameters: the application [Context](#), the resource ID for the string array, and the layout to use for each list item.

A [string array](#) defined in a resource looks like this:

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

The [ArrayAdapter](#) returned by [createFromResource\(\)](#) is complete and ready for you to pass it to [setListNavigationCallbacks\(\)](#) (in step 4 from above). Before you do, though, you need to create the [OnNavigationListener](#).

Your implementation of [ActionBar.OnNavigationListener](#) is where you handle fragment changes or other modifications to your activity when the user selects an item from the drop-down list. There's only one callback method to implement in the listener: [onNavigationItemSelected\(\)](#).

The [onNavigationItemSelected\(\)](#) method receives the position of the item in the list and a unique item ID provided by the [SpinnerAdapter](#).

Here's an example that instantiates an anonymous implementation of [OnNavigationListener](#), which inserts a [Fragment](#) into the layout container identified by `R.id.fragment_container`:

```
mOnNavigationListener = new OnNavigationListener() {  
    // Get the same strings provided for the drop-down's ArrayAdapter  
    String[] strings = getResources().getStringArray(R.array.action_list);  
  
    @Override  
    public boolean onNavigationItemSelected(int position, long itemId) {  
        // Create new fragment from our own Fragment class  
        ListContentFragment newFragment = new ListContentFragment();  
        FragmentTransaction ft = openFragmentTransaction();  
        // Replace whatever is in the fragment container with this fragment  
        // and give the fragment a tag name equal to the string at the position se
```

```

        ft.replace(R.id.fragment_container, newFragment, strings[position]);
        // Apply changes
        ft.commit();
        return true;
    }
} ;

```

This instance of [OnNavigationListener](#) is complete and you can now call [setListNavigationCallbacks\(\)](#) (in step 4), passing the [ArrayAdapter](#) and this [OnNavigationListener](#).

In this example, when the user selects an item from the drop-down list, a fragment is added to the layout (replacing the current fragment in the R.id.fragment\_container view). The fragment added is given a tag that uniquely identifies it, which is the same string used to identify the fragment in the drop-down list.

Here's a look at the `ListContentFragment` class that defines each fragment in this example:

```

public class ListContentFragment extends Fragment {
    private String mText;

    @Override
    public void onAttach(Activity activity) {
        // This is the first callback received; here we can set the text for
        // the fragment as defined by the tag specified during the fragment trans
        super.onAttach(activity);
        mText = getTag();
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
        // This is called to define the layout for the fragment;
        // we just create a TextView and set its text to be the fragment tag
        TextView text = new TextView(getActivity());
        text.setText(mText);
        return text;
    }
}

```

## Styling the Action Bar

If you want to implement a visual design that represents your app's brand, the action bar allows you to customize each detail of its appearance, including the action bar color, text colors, button styles, and more. To do so, you need to use Android's [style and theme](#) framework to restyle the action bar using special style properties.

**Caution:** For all background drawables you provide, be sure to use [Nine-Patch drawables](#) to allow stretching. The nine-patch image should be *smaller* than 40dp tall and 30dp wide.

### General appearance

#### [actionBarStyle](#)

Specifies a style resource that defines various style properties for the action bar.

The default for this style for this is [Widget.AppCompat.ActionBar](#), which is what you should use as the parent style.

Supported styles include:

#### **background**

Defines a drawable resource for the action bar background.

#### **backgroundStacked**

Defines a drawable resource for the stacked action bar (the [tabs](#)).

#### **backgroundSplit**

Defines a drawable resource for the [split action bar](#).

#### **actionButtonStyle**

Defines a style resource for action buttons.

The default for this style for this is [Widget.AppCompat.ActionButton](#), which is what you should use as the parent style.

#### **actionOverflowButtonStyle**

Defines a style resource for overflow action items.

The default for this style for this is [Widget.AppCompat.ActionButton.Overflow](#), which is what you should use as the parent style.

#### **displayOptions**

Defines one or more action bar display options, such as whether to use the app logo, show the activity title, or enable the *Up* action. See [displayOptions](#) for all possible values.

#### **divider**

Defines a drawable resource for the divider between action items.

#### **titleTextStyle**

Defines a style resource for the action bar title.

The default for this style for this is [TextAppearance-AppCompat.Widget.ActionBar.Title](#), which is what you should use as the parent style.

#### **windowActionBarOverlay**

Declares whether the action bar should overlay the activity layout rather than offset the activity's layout position (for example, the Gallery app uses overlay mode). This is `false` by default.

Normally, the action bar requires its own space on the screen and your activity layout fills in what's left over. When the action bar is in overlay mode, your activity layout uses all the available space and the system draws the action bar on top. Overlay mode can be useful if you want your content to keep a fixed size and position when the action bar is hidden and shown. You might also like to use it purely as a visual effect, because you can use a semi-transparent background for the action bar so the user can still see some of your activity layout behind the action bar.

**Note:** The [Holo](#) theme families draw the action bar with a semi-transparent background by default. However, you can modify it with your own styles and the [DeviceDefault](#) theme on different devices might use an opaque background by default.

When overlay mode is enabled, your activity layout has no awareness of the action bar lying on top of it. So, you must be careful not to place any important information or UI components in the area overlaid by the action bar. If appropriate, you can refer to the platform's value for  [actionBarSize](#) to determine the height of the action bar, by referencing it in your XML layout. For example:

```
<SomeView  
    ...  
    android:layout_marginTop="?android:attr/actionBarSize" />
```

You can also retrieve the action bar height at runtime with  [getHeight\(\)](#). This reflects the height of the action bar at the time it's called, which might not include the stacked action bar (due to navigation tabs) if called during early activity lifecycle methods. To see how you can determine the total height at runtime, including the stacked action bar, see the  [TitlesFragment](#) class in the  [Honeycomb Gallery](#) sample app.

## Action items

### [actionBarStyle](#)

Defines a style resource for the action item buttons.

The default for this style for this is  [Widget.AppCompat.ActionButton](#), which is what you should use as the parent style.

### [actionBarItemBackground](#)

Defines a drawable resource for each action item's background. This should be a  [state-list drawable](#) to indicate different selected states.

### [itemBackground](#)

Defines a drawable resource for each action overflow item's background. This should be a  [state-list drawable](#) to indicate different selected states.

### [actionBarDivider](#)

Defines a drawable resource for the divider between action items.

### [actionBarTextColor](#)

Defines a color for text that appears in an action item.

### [actionBarTextAppearance](#)

Defines a style resource for text that appears in an action item.

### [actionBarWidgetTheme](#)

Defines a theme resource for widgets that are inflated into the action bar as  [action views](#).

## Navigation tabs

### [actionBarTabStyle](#)

Defines a style resource for tabs in the action bar.

The default for this style for this is  [Widget.AppCompat.ActionBar.TabView](#), which is what you should use as the parent style.

### [actionBarTabBarStyle](#)

Defines a style resource for the thin bar that appears below the navigation tabs.

The default for this style for this is [Widget.AppCompat.ActionBar.TabBar](#), which is what you should use as the parent style.

#### actionBarTabTextStyle

Defines a style resource for text in the navigation tabs.

The default for this style for this is [Widget.AppCompat.ActionBar.TabText](#), which is what you should use as the parent style.

## Drop-down lists

#### actionDropDownStyle

Defines a style for the drop-down navigation (such as the background and text styles).

The default for this style for this is [Widget.AppCompat.Spinner.DropDown.ActionBar](#), which is what you should use as the parent style.

## Example theme

Here's an example that defines a custom theme for an activity, `CustomActivityTheme`, that includes several styles to customize the action bar.

Notice that there are two version for each action bar style property. The first one includes the `android:` prefix on the property name to support API levels 11 and higher that include these properties in the framework. The second version does *not* include the `android:` prefix and is for older versions of the platform, on which the system uses the style property from the support library. The effect for each is the same.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- the theme applied to the application or activity -->
    <style name="CustomActionBarTheme">
        parent="@style/Theme.AppCompat.Light"
        <item name="android: actionBarStyle">@style/MyActionBar</item>
        <item name="android: actionBarTabTextStyle">@style/TabTextStyle</item>
        <item name="android: actionMenuTextColor">@color/actionbar_text</item>

        <!-- Support library compatibility -->
        <item name="actionBarStyle">@style/MyActionBar</item>
        <item name="actionBarTabTextStyle">@style/TabTextStyle</item>
        <item name="actionMenuTextColor">@color/actionbar_text</item>
    </style>

    <!-- general styles for the action bar -->
    <style name="MyActionBar">
        parent="@style/Widget.AppCompat.ActionBar"
        <item name="android: titleTextStyle">@style/TitleTextStyle</item>
        <item name="android: background">@drawable/actionbar_background</item>
        <item name="android: backgroundStacked">@drawable/actionbar_background</item>
        <item name="android: backgroundSplit">@drawable/actionbar_background</item>

        <!-- Support library compatibility -->
        <item name="titleTextStyle">@style/TitleTextStyle</item>
        <item name="background">@drawable/actionbar_background</item>
        <item name="backgroundStacked">@drawable/actionbar_background</item>
        <item name="backgroundSplit">@drawable/actionbar_background</item>
    </style>
</resources>
```

```
<item name="backgroundSplit">@drawable/ActionBar_background</item>
</style>

<!-- action bar title text -->
<style name="TitleTextStyle"
    parent="@style/TextAppearance.AppCompat.Widget.ActionBar.Title">
    <item name="android:textColor">@color/ActionBar_text</item>
</style>

<!-- action bar tab text -->
<style name="TabTextStyle"
    parent="@style/Widget.AppCompat.ActionBar.TabText">
    <item name="android:textColor">@color/ActionBar_text</item>
</style>
</resources>
```

In your manifest file, you can apply the theme to your entire app:

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

Or to individual activities:

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

**Caution:** Be certain that each theme and style declares a parent theme in the `<style>` tag, from which it inherits all styles not explicitly declared by your theme. When modifying the action bar, using a parent theme is important so that you can simply override the action bar styles you want to change without re-implementing the styles you want to leave alone (such as text size or padding in action items).

For more information about using style and theme resources in your application, read [Styles and Themes](#).

# Settings

## In this document

1. [Overview](#)
  1. [Preferences](#)
2. [Defining Preferences in XML](#)
  1. [Creating setting groups](#)
  2. [Using intents](#)
3. [Creating a Preference Activity](#)
4. [Using Preference Fragments](#)
5. [Setting Default Values](#)
6. [Using Preference Headers](#)
  1. [Creating the headers file](#)
  2. [Displaying the headers](#)
  3. [Supporting older versions with preference headers](#)
7. [Reading Preferences](#)
  1. [Listening for preference changes](#)
8. [Managing Network Usage](#)
9. [Building a Custom Preference](#)
  1. [Specifying the user interface](#)
  2. [Saving the setting's value](#)
  3. [Initializing the current value](#)
  4. [Providing a default value](#)
  5. [Saving and restoring the Preference's state](#)

## Key classes

1. [Preference](#)
2. [PreferenceActivity](#)
3. [PreferenceFragment](#)

## See also

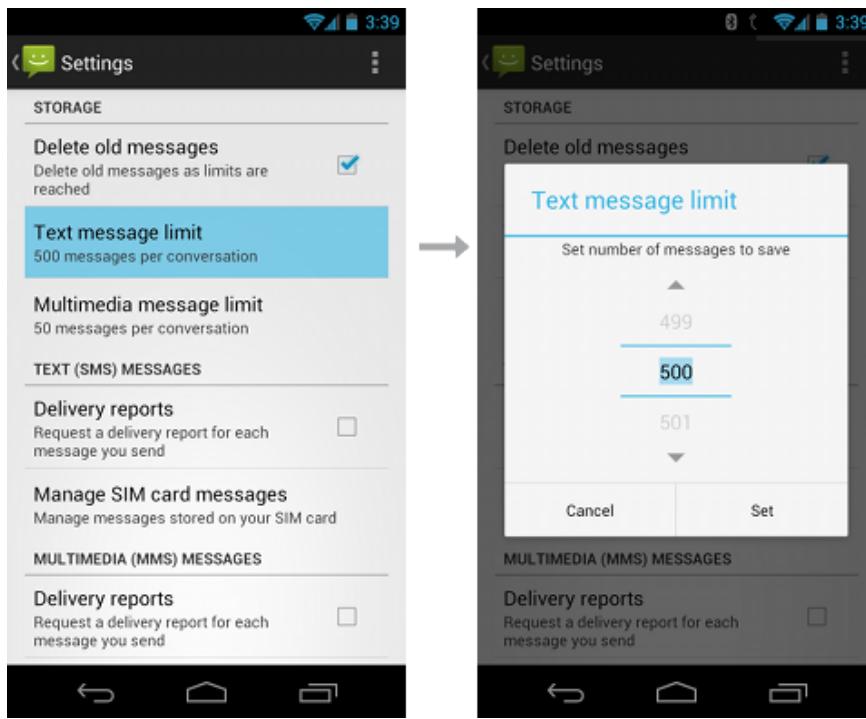
1. [Settings design guide](#)

Applications often include settings that allow users to modify app features and behaviors. For example, some apps allow users to specify whether notifications are enabled or specify how often the application syncs data with the cloud.

If you want to provide settings for your app, you should use Android's [Preference](#) APIs to build an interface that's consistent with the user experience in other Android apps (including the system settings). This document describes how to build your app settings using [Preference](#) APIs.

## Settings Design

For information about how to design your settings, read the [Settings](#) design guide.



**Figure 1.** Screenshots from the Android Messaging app's settings. Selecting an item defined by a [Preference](#) opens an interface to change the setting.

## Overview

Instead of using [View](#) objects to build the user interface, settings are built using various subclasses of the [Preference](#) class that you declare in an XML file.

A [Preference](#) object is the building block for a single setting. Each [Preference](#) appears as an item in a list and provides the appropriate UI for users to modify the setting. For example, a [CheckBoxPreference](#) creates a list item that shows a checkbox, and a [ListPreference](#) creates an item that opens a dialog with a list of choices.

Each [Preference](#) you add has a corresponding key-value pair that the system uses to save the setting in a default [SharedPreferences](#) file for your app's settings. When the user changes a setting, the system updates the corresponding value in the [SharedPreferences](#) file for you. The only time you should directly interact with the associated [SharedPreferences](#) file is when you need to read the value in order to determine your app's behavior based on the user's setting.

The value saved in [SharedPreferences](#) for each setting can be one of the following data types:

- Boolean
- Float
- Int
- Long
- String
- String [Set](#)

Because your app's settings UI is built using [Preference](#) objects instead of [View](#) objects, you need to use a specialized [Activity](#) or [Fragment](#) subclass to display the list settings:

- If your app supports versions of Android older than 3.0 (API level 10 and lower), you must build the activity as an extension of the [PreferenceActivity](#) class.

- On Android 3.0 and later, you should instead use a traditional [Activity](#) that hosts a [PreferenceFragment](#) that displays your app settings. However, you can also use [PreferenceActivity](#) to create a two-pane layout for large screens when you have multiple groups of settings.

How to set up your [PreferenceActivity](#) and instances of [PreferenceFragment](#) is discussed in the sections about [Creating a Preference Activity](#) and [Using Preference Fragments](#).

## Preferences

Every setting for your app is represented by a specific subclass of the [Preference](#) class. Each subclass includes a set of core properties that allow you to specify things such as a title for the setting and the default value. Each subclass also provides its own specialized properties and user interface. For instance, figure 1 shows a screenshot from the Messaging app's settings. Each list item in the settings screen is backed by a different [Preference](#) object.

A few of the most common preferences are:

### [CheckBoxPreference](#)

Shows an item with a checkbox for a setting that is either enabled or disabled. The saved value is a boolean (`true` if it's checked).

### [ListPreference](#)

Opens a dialog with a list of radio buttons. The saved value can be any one of the supported value types (listed above).

### [EditTextPreference](#)

Opens a dialog with an [EditText](#) widget. The saved value is a [String](#).

See the [Preference](#) class for a list of all other subclasses and their corresponding properties.

Of course, the built-in classes don't accommodate every need and your application might require something more specialized. For example, the platform currently does not provide a [Preference](#) class for picking a number or a date. So you might need to define your own [Preference](#) subclass. For help doing so, see the section about [Building a Custom Preference](#).

## Defining Preferences in XML

Although you can instantiate new [Preference](#) objects at runtime, you should define your list of settings in XML with a hierarchy of [Preference](#) objects. Using an XML file to define your collection of settings is preferred because the file provides an easy-to-read structure that's simple to update. Also, your app's settings are generally pre-determined, although you can still modify the collection at runtime.

Each [Preference](#) subclass can be declared with an XML element that matches the class name, such as `<CheckBoxPreference>`.

You must save the XML file in the `res/xml/` directory. Although you can name the file anything you want, it's traditionally named `preferences.xml`. You usually need only one file, because branches in the hierarchy (that open their own list of settings) are declared using nested instances of [PreferenceScreen](#).

**Note:** If you want to create a multi-pane layout for your settings, then you need separate XML files for each fragment.

The root node for the XML file must be a [`<PreferenceScreen>`](#) element. Within this element is where you add each [`Preference`](#). Each child you add within the [`<PreferenceScreen>`](#) element appears as a single item in the list of settings.

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <CheckBoxPreference
        android:key="pref_sync"
        android:title="@string/pref_sync"
        android:summary="@string/pref_sync_summ"
        android:defaultValue="true" />
    <ListPreference
        android:dependency="pref_sync"
        android:key="pref_syncConnectionType"
        android:title="@string/pref_syncConnectionType"
        android:dialogTitle="@string/pref_syncConnectionType"
        android:entries="@array/pref_syncConnectionTypes_entries"
        android:entryValues="@array/pref_syncConnectionTypes_values"
        android:defaultValue="@string/pref_syncConnectionTypes_default" />
</PreferenceScreen>
```

In this example, there's a [`CheckBoxPreference`](#) and a [`ListPreference`](#). Both items include the following three attributes:

#### **android:key**

This attribute is required for preferences that persist a data value. It specifies the unique key (a string) the system uses when saving this setting's value in the [`SharedPreferences`](#).

The only instances in which this attribute is *not required* is when the preference is a [`PreferenceCategory`](#) or [`PreferenceScreen`](#), or the preference specifies an [`Intent`](#) to invoke (with an [`<intent>`](#) element) or a [`Fragment`](#) to display (with an [`android:fragment`](#) attribute).

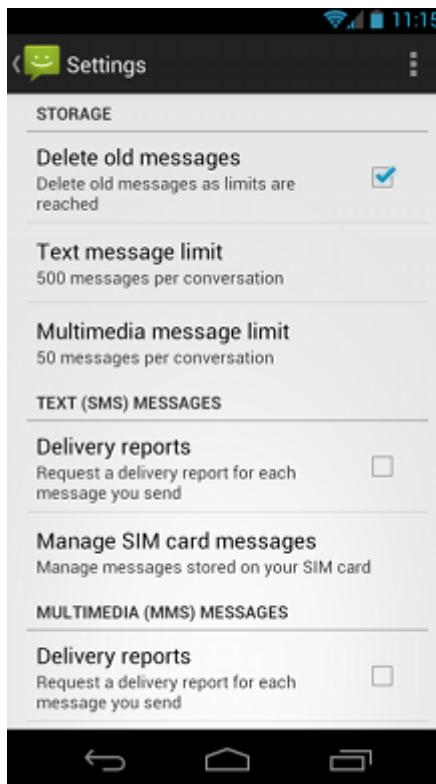
#### **android:title**

This provides a user-visible name for the setting.

#### **android.defaultValue**

This specifies the initial value that the system should set in the [`SharedPreferences`](#) file. You should supply a default value for all settings.

For information about all other supported attributes, see the [`Preference`](#) (and respective subclass) documentation.



**Figure 2.** Setting categories with titles.

1. The category is specified by the `<PreferenceCategory>` element.
2. The title is specified with the `android:title` attribute.

When your list of settings exceeds about 10 items, you might want to add titles to define groups of settings or display those groups in a separate screen. These options are described in the following sections.

## Creating setting groups

If you present a list of 10 or more settings, users may have difficulty scanning, comprehending, and processing them. You can remedy this by dividing some or all of the settings into groups, effectively turning one long list into multiple shorter lists. A group of related settings can be presented in one of two ways:

- [Using titles](#)
- [Using subscreens](#)

You can use one or both of these grouping techniques to organize your app's settings. When deciding which to use and how to divide your settings, you should follow the guidelines in Android Design's [Settings](#) guide.

### Using titles

If you want to provide dividers with headings between groups of settings (as shown in figure 2), place each group of `Preference` objects inside a `PreferenceCategory`.

For example:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory
        android:title="@string/pref_sms_storage_title"
        android:key="pref_key_storage_settings">
        <CheckBoxPreference
            android:key="pref_key_auto_delete">
```

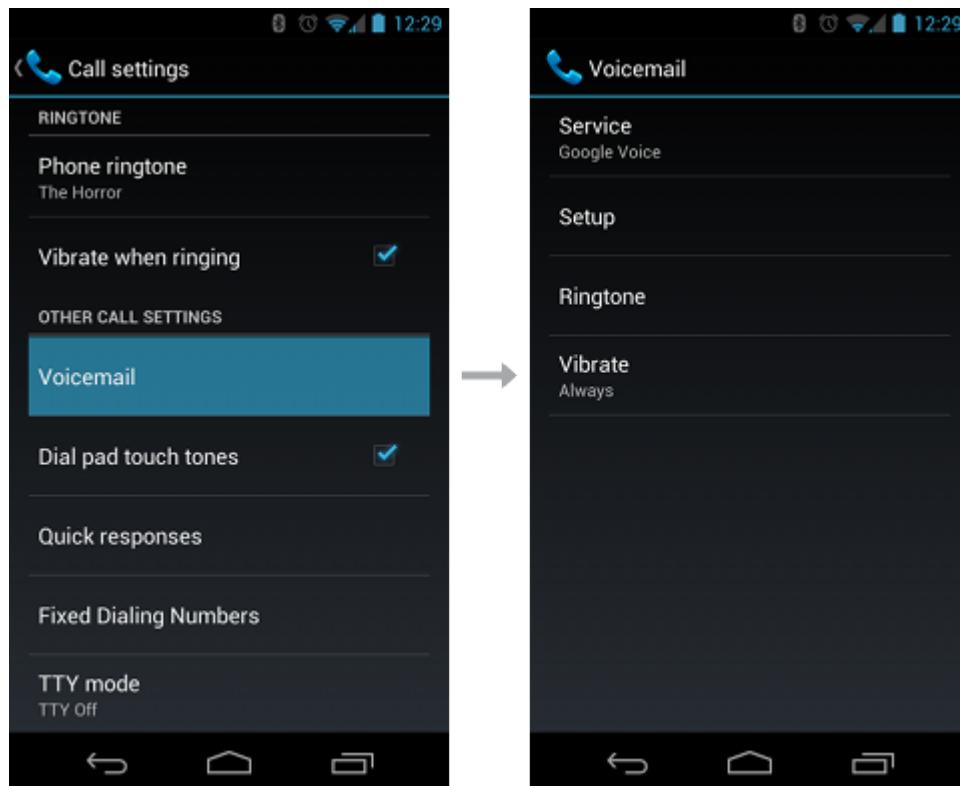
```

        android:summary="@string/pref_summary_auto_delete"
        android:title="@string/pref_title_auto_delete"
        android:defaultValue="false" ... />
    <Preference
        android:key="pref_key_sms_delete_limit"
        android:dependency="pref_key_auto_delete"
        android:summary="@string/pref_summary_delete_limit"
        android:title="@string/pref_title_sms_delete" ... />
    <Preference
        android:key="pref_key_mms_delete_limit"
        android:dependency="pref_key_auto_delete"
        android:summary="@string/pref_summary_delete_limit"
        android:title="@string/pref_title_mms_delete" ... />
</PreferenceCategory>
...
</PreferenceScreen>

```

## Using subscreens

If you want to place groups of settings into a subscreen (as shown in figure 3), place the group of [Preference](#) objects inside a [PreferenceScreen](#).



**Figure 3.** Setting subscreens. The `<PreferenceScreen>` element creates an item that, when selected, opens a separate list to display the nested settings.

For example:

```

<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- opens a subscreen of settings -->
    <PreferenceScreen
        android:key="button_voicemail_category_key"
        android:title="@string/voicemail"

```

```
    android:persistent="false">
    <ListPreference
        android:key="button_voicemail_provider_key"
        android:title="@string/voicemail_provider" ... />
    <!-- opens another nested subscreen -->
    <PreferenceScreen
        android:key="button_voicemail_setting_key"
        android:title="@string/voicemail_settings"
        android:persistent="false">
        ...
    </PreferenceScreen>
    <RingtonePreference
        android:key="button_voicemail_ringtone_key"
        android:title="@string/voicemail_ringtone_title"
        android:ringtoneType="notification" ... />
        ...
    </PreferenceScreen>
    ...
</PreferenceScreen>
```

## Using intents

In some cases, you might want a preference item to open a different activity instead of a settings screen, such as a web browser to view a web page. To invoke an [Intent](#) when the user selects a preference item, add an `<intent>` element as a child of the corresponding `<Preference>` element.

For example, here's how you can use a preference item to open a web page:

```
<Preference android:title="@string/prefs_web_page" >
    <intent android:action="android.intent.action.VIEW"
            android:data="http://www.example.com" />
</Preference>
```

You can create both implicit and explicit intents using the following attributes:

### **android:action**

The action to assign, as per the [setAction\(\)](#) method.

### **android:data**

The data to assign, as per the [setData\(\)](#) method.

### **android:mimeType**

The MIME type to assign, as per the [setType\(\)](#) method.

### **android:targetClass**

The class part of the component name, as per the [setComponent\(\)](#) method.

### **android:targetPackage**

The package part of the component name, as per the [setComponent\(\)](#) method.

# Creating a Preference Activity

To display your settings in an activity, extend the [PreferenceActivity](#) class. This is an extension of the traditional [Activity](#) class that displays a list of settings based on a hierarchy of [Preference](#) objects. The [PreferenceActivity](#) automatically persists the settings associated with each [Preference](#) when the user makes a change.

**Note:** If you're developing your application for Android 3.0 and higher, you should instead use [PreferenceFragment](#). Go to the next section about [Using Preference Fragments](#).

The most important thing to remember is that you do not load a layout of views during the [onCreate\(\)](#) callback. Instead, you call [addPreferencesFromResource\(\)](#) to add the preferences you've declared in an XML file to the activity. For example, here's the bare minimum code required for a functional [PreferenceActivity](#):

```
public class SettingsActivity extends PreferenceActivity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        addPreferencesFromResource(R.xml.preferences);  
    }  
}
```

This is actually enough code for some apps, because as soon as the user modifies a preference, the system saves the changes to a default [SharedPreferences](#) file that your other application components can read when you need to check the user's settings. Many apps, however, require a little more code in order to listen for changes that occur to the preferences. For information about listening to changes in the [SharedPreferences](#) file, see the section about [Reading Preferences](#).

## Using Preference Fragments

If you're developing for Android 3.0 (API level 11) and higher, you should use a [PreferenceFragment](#) to display your list of [Preference](#) objects. You can add a [PreferenceFragment](#) to any activity—you don't need to use [PreferenceActivity](#).

[Fragments](#) provide a more flexible architecture for your application, compared to using activities alone, no matter what kind of activity you're building. As such, we suggest you use [PreferenceFragment](#) to control the display of your settings instead of [PreferenceActivity](#) when possible.

Your implementation of [PreferenceFragment](#) can be as simple as defining the [onCreate\(\)](#) method to load a preferences file with [addPreferencesFromResource\(\)](#). For example:

```
public static class SettingsFragment extends PreferenceFragment {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        // Load the preferences from an XML resource  
        addPreferencesFromResource(R.xml.preferences);  
    }  
    ...  
}
```

You can then add this fragment to an [Activity](#) just as you would for any other [Fragment](#). For example:

```
public class SettingsActivity extends Activity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        // Display the fragment as the main content.  
        getSupportFragmentManager().beginTransaction()  
            .replace(android.R.id.content, new SettingsFragment())  
            .commit();  
    }  
}
```

**Note:** A [PreferenceFragment](#) doesn't have its own [Context](#) object. If you need a [Context](#) object, you can call [getActivity\(\)](#). However, be careful to call [getActivity\(\)](#) only when the fragment is attached to an activity. When the fragment is not yet attached, or was detached during the end of its lifecycle, [getActivity\(\)](#) will return null.

## Setting Default Values

The preferences you create probably define some important behaviors for your application, so it's necessary that you initialize the associated [SharedPreferences](#) file with default values for each [Preference](#) when the user first opens your application.

The first thing you must do is specify a default value for each [Preference](#) object in your XML file using the `android:defaultValue` attribute. The value can be any data type that is appropriate for the corresponding [Preference](#) object. For example:

```
<!-- default value is a boolean -->  
<CheckBoxPreference  
    android:defaultValue="true"  
    ... />  
  
<!-- default value is a string -->  
<ListPreference  
    android:defaultValue="@string/pref_syncConnectionTypes_default"  
    ... />
```

Then, from the [onCreate\(\)](#) method in your application's main activity—and in any other activity through which the user may enter your application for the first time—call [setDefaults\(\)](#):

```
PreferenceManager.setDefaultValues(this, R.xml.advanced_preferences, false);
```

Calling this during [onCreate\(\)](#) ensures that your application is properly initialized with default settings, which your application might need to read in order to determine some behaviors (such as whether to download data while on a cellular network).

This method takes three arguments:

- Your application [Context](#).
- The resource ID for the preference XML file for which you want to set the default values.
- A boolean indicating whether the default values should be set more than once.

When `false`, the system sets the default values only if this method has never been called in the past (or the [KEY HAS SET DEFAULT VALUES](#) in the default value shared preferences file is `false`).

As long as you set the third argument to `false`, you can safely call this method every time your activity starts without overriding the user's saved preferences by resetting them to the defaults. However, if you set it to `true`, you will override any previous values with the defaults.

## Using Preference Headers

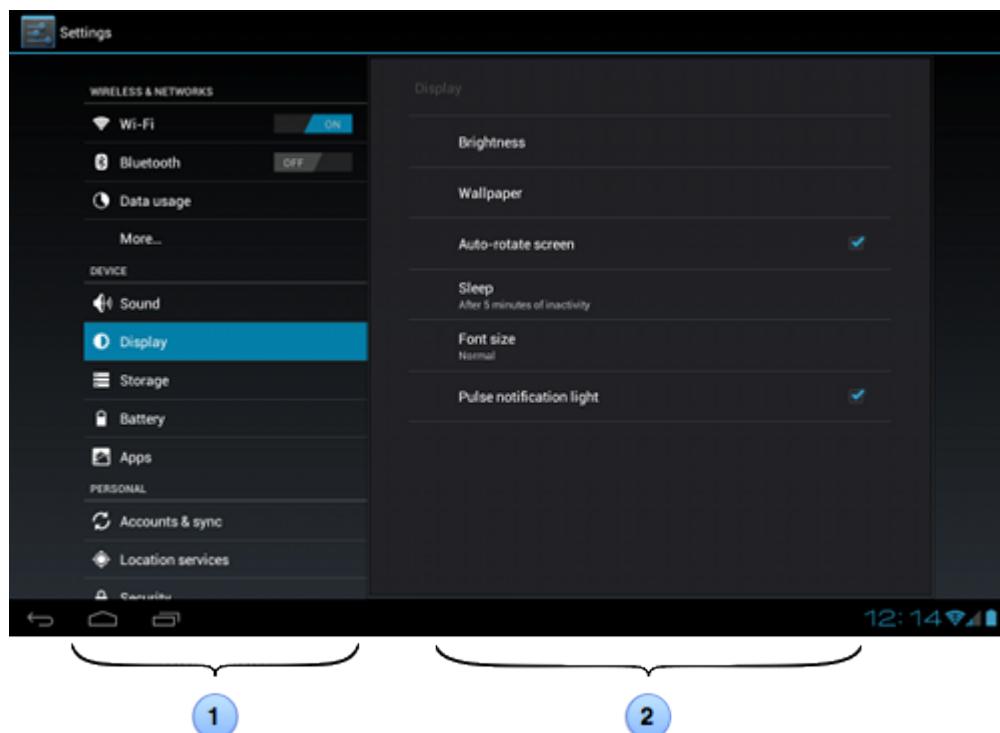
In rare cases, you might want to design your settings such that the first screen displays only a list of [subscreens](#) (such as in the system Settings app, as shown in figures 4 and 5). When you're developing such a design for Android 3.0 and higher, you should use a new "headers" feature in Android 3.0, instead of building subscreens with nested [PreferenceScreen](#) elements.

To build your settings with headers, you need to:

1. Separate each group of settings into separate instances of [PreferenceFragment](#). That is, each group of settings needs a separate XML file.
2. Create an XML headers file that lists each settings group and declares which fragment contains the corresponding list of settings.
3. Extend the [PreferenceActivity](#) class to host your settings.
4. Implement the [onBuildHeaders\(\)](#) callback to specify the headers file.

A great benefit to using this design is that [PreferenceActivity](#) automatically presents the two-pane layout shown in figure 4 when running on large screens.

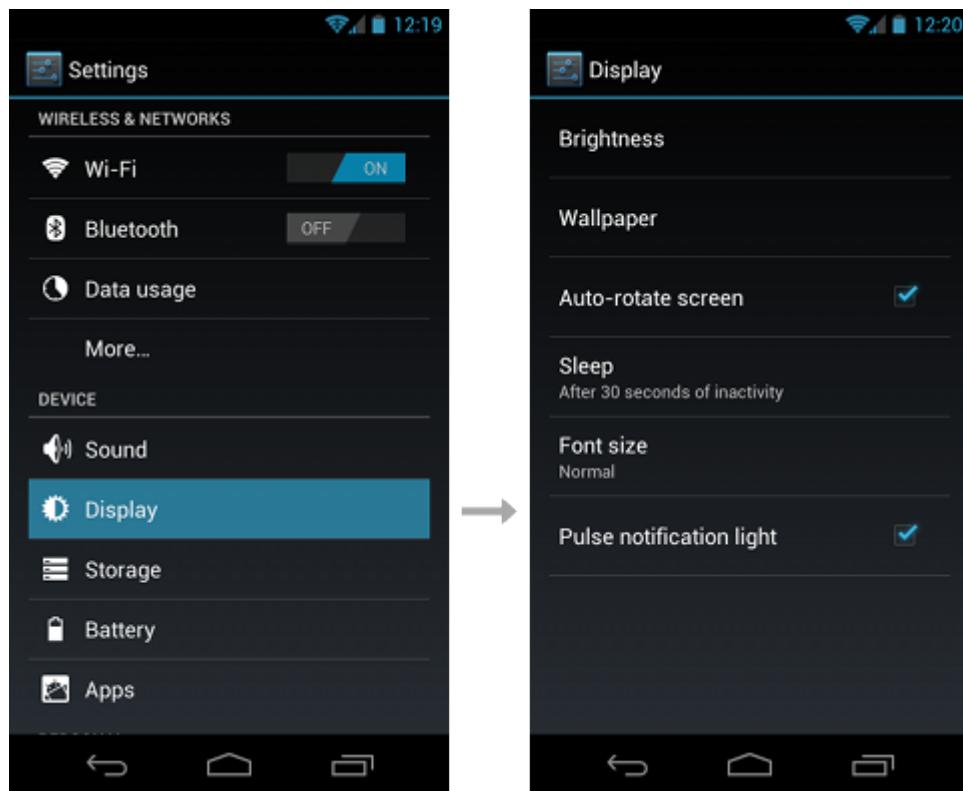
Even if your application supports versions of Android older than 3.0, you can build your application to use [PreferenceFragment](#) for a two-pane presentation on newer devices while still supporting a traditional multi-screen hierarchy on older devices (see the section about [Supporting older versions with preference headers](#)).



**Figure 4.** Two-pane layout with headers.

1. The headers are defined with an XML headers file.

2. Each group of settings is defined by a [PreferenceFragment](#) that's specified by a <header> element in the headers file.



**Figure 5.** A handset device with setting headers. When an item is selected, the associated [PreferenceFragment](#) replaces the headers.

## Creating the headers file

Each group of settings in your list of headers is specified by a single <header> element inside a root <preference-headers> element. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
    <header
        android:fragment="com.example.prefs.SettingsActivity$SettingsFragmentOne"
        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one" />
    <header
        android:fragment="com.example.prefs.SettingsActivity$SettingsFragmentTwo"
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" >
        <!-- key/value pairs can be included as arguments for the fragment. -->
        <extra android:name="someKey" android:value="someHeaderValue" />
    </header>
</preference-headers>
```

With the `android:fragment` attribute, each header declares an instance of [PreferenceFragment](#) that should open when the user selects the header.

The <extras> element allows you to pass key-value pairs to the fragment in a [Bundle](#). The fragment can retrieve the arguments by calling [getArguments\(\)](#). You might pass arguments to the fragment for a variety of

reasons, but one good reason is to reuse the same subclass of [PreferenceFragment](#) for each group and use the argument to specify which preferences XML file the fragment should load.

For example, here's a fragment that can be reused for multiple settings groups, when each header defines an `<extra>` argument with the "settings" key:

```
public static class SettingsFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        String settings = getArguments().getString("settings");
        if ("notifications".equals(settings)) {
            addPreferencesFromResource(R.xml.settings_wifi);
        } else if ("sync".equals(settings)) {
            addPreferencesFromResource(R.xml.settings_sync);
        }
    }
}
```

## Displaying the headers

To display the preference headers, you must implement the [onBuildHeaders\(\)](#) callback method and call [loadHeadersFromResource\(\)](#). For example:

```
public class SettingsActivity extends PreferenceActivity {
    @Override
    public void onBuildHeaders(List<Header> target) {
        loadHeadersFromResource(R.xml.preference_headers, target);
    }
}
```

When the user selects an item from the list of headers, the system opens the associated [PreferenceFragment](#).

**Note:** When using preference headers, your subclass of [PreferenceActivity](#) doesn't need to implement the [onCreate\(\)](#) method, because the only required task for the activity is to load the headers.

## Supporting older versions with preference headers

If your application supports versions of Android older than 3.0, you can still use headers to provide a two-pane layout when running on Android 3.0 and higher. All you need to do is create an additional preferences XML file that uses basic [<Preference>](#) elements that behave like the header items (to be used by the older Android versions).

Instead of opening a new [PreferenceScreen](#), however, each of the [<Preference>](#) elements sends an [Intent](#) to the [PreferenceActivity](#) that specifies which preference XML file to load.

For example, here's an XML file for preference headers that is used on Android 3.0 and higher (`res/xml/preference_headers.xml`):

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
    <header
        android:fragment="com.example.prefs.SettingsFragmentOne">
```

```

        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one" />
<header
        android:fragment="com.example.prefs.SettingsFragmentTwo"
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" />
</preference-headers>
```

And here is a preference file that provides the same headers for versions older than Android 3.0 (`res/xml/preference_headers_legacy.xml`):

```

<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <Preference
        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one" >
        <intent
            android:targetPackage="com.example.prefs"
            android:targetClass="com.example.prefs.SettingsActivity"
            android:action="com.example.prefs.PREFS_ONE" />
    </Preference>
    <Preference
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" >
        <intent
            android:targetPackage="com.example.prefs"
            android:targetClass="com.example.prefs.SettingsActivity"
            android:action="com.example.prefs.PREFS_TWO" />
    </Preference>
</PreferenceScreen>
```

Because support for `<preference-headers>` was added in Android 3.0, the system calls [onBuildHeaders\(\)](#) in your [PreferenceActivity](#) only when running on Androd 3.0 or higher. In order to load the "legacy" headers file (`preference_headers_legacy.xml`), you must check the Android version and, if the version is older than Android 3.0 ([HONEYCOMB](#)), call [addPreferencesFromResource\(\)](#) to load the legacy header file. For example:

```

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

    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
        // Load the legacy preferences headers
        addPreferencesFromResource(R.xml.preference_headers_legacy);
    }
}

// Called only on Honeycomb and later
@Override
public void onBuildHeaders(List<Header> target) {
    loadHeadersFromResource(R.xml.preference_headers, target);
}
```

The only thing left to do is handle the [Intent](#) that's passed into the activity to identify which preference file to load. So retrieve the intent's action and compare it to known action strings that you've used in the preference XML's <intent> tags:

```
final static String ACTION_PREFS_ONE = "com.example.prefs.PREFS_ONE";
...
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String action = getIntent().getAction();
    if (action != null && action.equals(ACTION_PREFS_ONE)) {
        addPreferencesFromResource(R.xml.preferences);
    }
    ...
}

else if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
    // Load the legacy preferences headers
    addPreferencesFromResource(R.xml.preference_headers_legacy);
}
}
```

Beware that consecutive calls to [addPreferencesFromResource\(\)](#) will stack all the preferences in a single list, so be sure that it's only called once by chaining the conditions with else-if statements.

## Reading Preferences

By default, all your app's preferences are saved to a file that's accessible from anywhere within your application by calling the static method [PreferenceManager.getDefaultSharedPreferences\(\)](#). This returns the [SharedPreferences](#) object containing all the key-value pairs that are associated with the [Preference](#) objects used in your [PreferenceActivity](#).

For example, here's how you can read one of the preference values from any other activity in your application:

```
SharedPreferences sharedPref = PreferenceManager.getDefaultSharedPreferences();
String syncConnPref = sharedPref.getString(SettingsActivity.KEY_PREF_SYNC_CONN,
```

## Listening for preference changes

There are several reasons you might want to be notified as soon as the user changes one of the preferences. In order to receive a callback when a change happens to any one of the preferences, implement the [SharedPreference.OnSharedPreferenceChangeListener](#) interface and register the listener for the [SharedPreferences](#) object by calling [registerOnSharedPreferenceChangeListener\(\)](#).

The interface has only one callback method, [onSharedPreferenceChanged\(\)](#), and you might find it easiest to implement the interface as a part of your activity. For example:

```
public class SettingsActivity extends PreferenceActivity
    implements OnSharedPreferenceChangeListener {
    public static final String KEY_PREF_SYNC_CONN = "pref_syncConnectionType";
    ...
}
```

```

public void onSharedPreferenceChanged(SharedPreferences sharedPreferences,
    if (key.equals(KEY_PREF_SYNC_CONN)) {
        Preference connectionPref = findPreference(key);
        // Set summary to be the user-description for the selected value
        connectionPref.setSummary(sharedPreferences.getString(key, ""));
    }
}
}

```

In this example, the method checks whether the changed setting is for a known preference key. It calls [findPreference\(\)](#) to get the [Preference](#) object that was changed so it can modify the item's summary to be a description of the user's selection. That is, when the setting is a [ListPreference](#) or other multiple choice setting, you should call [setSummary\(\)](#) when the setting changes to display the current status (such as the Sleep setting shown in figure 5).

**Note:** As described in the Android Design document about [Settings](#), we recommend that you update the summary for a [ListPreference](#) each time the user changes the preference in order to describe the current setting.

For proper lifecycle management in the activity, we recommend that you register and unregister your [SharedPreference.OnSharedPreferenceChangeListener](#) during the [onResume\(\)](#) and [onPause\(\)](#) callbacks, respectively:

```

@Override
protected void onResume() {
    super.onResume();
    getPreferenceScreen().getSharedPreferences()
        .registerOnSharedPreferenceChangeListener(this);
}

@Override
protected void onPause() {
    super.onPause();
    getPreferenceScreen().getSharedPreferences()
        .unregisterOnSharedPreferenceChangeListener(this);
}

```

## Managing Network Usage

Beginning with Android 4.0, the system's Settings application allows users to see how much network data their applications are using while in the foreground and background. Users can then disable the use of background data for individual apps. In order to avoid users disabling your app's access to data from the background, you should use the data connection efficiently and allow users to refine your app's data usage through your application settings.

For example, you might allow the user to control how often your app syncs data, whether your app performs uploads/downloads only when on Wi-Fi, whether your app uses data while roaming, etc. With these controls available to them, users are much less likely to disable your app's access to data when they approach the limits they set in the system Settings, because they can instead precisely control how much data your app uses.

Once you've added the necessary preferences in your [PreferenceActivity](#) to control your app's data habits, you should add an intent filter for [ACTION\\_MANAGE\\_NETWORK\\_USAGE](#) in your manifest file. For example:

```
<activity android:name="SettingsActivity" ... >
    <intent-filter>
        <action android:name="android.intent.action.MANAGE_NETWORK_USAGE" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

This intent filter indicates to the system that this is the activity that controls your application's data usage. Thus, when the user inspects how much data your app is using from the system's Settings app, a *View application settings* button is available that launches your [PreferenceActivity](#) so the user can refine how much data your app uses.

## Building a Custom Preference

The Android framework includes a variety of [Preference](#) subclasses that allow you to build a UI for several different types of settings. However, you might discover a setting you need for which there's no built-in solution, such as a number picker or date picker. In such a case, you'll need to create a custom preference by extending the [Preference](#) class or one of the other subclasses.

When you extend the [Preference](#) class, there are a few important things you need to do:

- Specify the user interface that appears when the user selects the settings.
- Save the setting's value when appropriate.
- Initialize the [Preference](#) with the current (or default) value when it comes into view.
- Provide the default value when requested by the system.
- If the [Preference](#) provides its own UI (such as a dialog), save and restore the state to handle lifecycle changes (such as when the user rotates the screen).

The following sections describe how to accomplish each of these tasks.

### Specifying the user interface

If you directly extend the [Preference](#) class, you need to implement [onClick\(\)](#) to define the action that occurs when the user selects the item. However, most custom settings extend [DialogPreference](#) to show a dialog, which simplifies the procedure. When you extend [DialogPreference](#), you must call [setDialogLayoutResource\(\)](#) during in the class constructor to specify the layout for the dialog.

For example, here's the constructor for a custom [DialogPreference](#) that declares the layout and specifies the text for the default positive and negative dialog buttons:

```
public class NumberPickerPreference extends DialogPreference {
    public NumberPickerPreference(Context context, AttributeSet attrs) {
        super(context, attrs);

        setDialogLayoutResource(R.layout.numberpicker_dialog);
        setPositiveButtonText(android.R.string.ok);
        setNegativeButtonText(android.R.string.cancel);

        setDialogIcon(null);
    }
    ...
}
```

## Saving the setting's value

You can save a value for the setting at any time by calling one of the [Preference](#) class's `persist*` () methods, such as [persistInt\(\)](#) if the setting's value is an integer or [persistBoolean\(\)](#) to save a boolean.

**Note:** Each [Preference](#) can save only one data type, so you must use the `persist*` () method appropriate for the data type used by your custom [Preference](#).

When you choose to persist the setting can depend on which [Preference](#) class you extend. If you extend [DialogPreference](#), then you should persist the value only when the dialog closes due to a positive result (the user selects the "OK" button).

When a [DialogPreference](#) closes, the system calls the [onDialogClosed\(\)](#) method. The method includes a boolean argument that specifies whether the user result is "positive"—if the value is `true`, then the user selected the positive button and you should save the new value. For example:

```
@Override  
protected void onDialogClosed(boolean positiveResult) {  
    // When the user selects "OK", persist the new value  
    if (positiveResult) {  
        persistInt(mnewValue);  
    }  
}
```

In this example, `mnewValue` is a class member that holds the setting's current value. Calling [persistInt\(\)](#) saves the value to the [SharedPreferences](#) file (automatically using the key that's specified in the XML file for this [Preference](#)).

## Initializing the current value

When the system adds your [Preference](#) to the screen, it calls [onSetInitialValue\(\)](#) to notify you whether the setting has a persisted value. If there is no persisted value, this call provides you the default value.

The [onSetInitialValue\(\)](#) method passes a boolean, `restorePersistedValue`, to indicate whether a value has already been persisted for the setting. If it is `true`, then you should retrieve the persisted value by calling one of the [Preference](#) class's `getPersisted*` () methods, such as [getPersistedInt\(\)](#) for an integer value. You'll usually want to retrieve the persisted value so you can properly update the UI to reflect the previously saved value.

If `restorePersistedValue` is `false`, then you should use the default value that is passed in the second argument.

```
@Override  
protected void onSetInitialValue(boolean restorePersistedValue, Object defaultValue)  
{  
    if (restorePersistedValue) {  
        // Restore existing state  
        mCurrentValue = this.getPersistedInt(DEFAULT_VALUE);  
    } else {  
        // Set default state from the XML attribute  
        mCurrentValue = (Integer) defaultValue;  
        persistInt(mCurrentValue);  
    }  
}
```

```
    }  
}
```

Each `getPersisted*()` method takes an argument that specifies the default value to use in case there is actually no persisted value or the key does not exist. In the example above, a local constant is used to specify the default value in case `getPersistedInt()` can't return a persisted value.

**Caution:** You **cannot** use the `defaultValue` as the default value in the `getPersisted*` () method, because its value is always null when `restorePersistedValue` is true.

## Providing a default value

If the instance of your `Preference` class specifies a default value (with the `android:defaultValue` attribute), then the system calls `onGetDefaultValue()` when it instantiates the object in order to retrieve the value. You must implement this method in order for the system to save the default value in the `SharedPreferences`. For example:

```
@Override  
protected Object onGetDefaultValue(TypedArray a, int index) {  
    return a.getInteger(index, DEFAULT_VALUE);  
}
```

The method arguments provide everything you need: the array of attributes and the index position of the `android:defaultValue`, which you must retrieve. The reason you must implement this method to extract the default value from the attribute is because you must specify a local default value for the attribute in case the value is undefined.

## Saving and restoring the Preference's state

Just like a `View` in a layout, your `Preference` subclass is responsible for saving and restoring its state in case the activity or fragment is restarted (such as when the user rotates the screen). To properly save and restore the state of your `Preference` class, you must implement the lifecycle callback methods `onSaveInstanceState()` and `onRestoreInstanceState()`.

The state of your `Preference` is defined by an object that implements the `Parcelable` interface. The Android framework provides such an object for you as a starting point to define your state object: the `Preference.BaseSavedState` class.

To define how your `Preference` class saves its state, you should extend the `Preference.BaseSavedState` class. You need to override just a few methods and define the `CREATOR` object.

For most apps, you can copy the following implementation and simply change the lines that handle the `value` if your `Preference` subclass saves a data type other than an integer.

```
private static class SavedState extends BaseSavedState {  
    // Member that holds the setting's value  
    // Change this data type to match the type saved by your Preference  
    int value;  
  
    public SavedState(Parcelable superState) {  
        super(superState);  
    }
```

```

public SavedState(Parcel source) {
    super(source);
    // Get the current preference's value
    value = source.readInt(); // Change this to read the appropriate data
}

@Override
public void writeToParcel(Parcel dest, int flags) {
    super.writeToParcel(dest, flags);
    // Write the preference's value
    dest.writeInt(value); // Change this to write the appropriate data type
}

// Standard creator object using an instance of this class
public static final Parcelable.Creator<SavedState> CREATOR =
    new Parcelable.Creator<SavedState>() {

        public SavedState createFromParcel(Parcel in) {
            return new SavedState(in);
        }

        public SavedState[] newArray(int size) {
            return new SavedState[size];
        }
    };
}

```

With the above implementation of [Preference.BaseSavedState](#) added to your app (usually as a subclass of your [Preference](#) subclass), you then need to implement the [onSaveInstanceState\(\)](#) and [onRestoreInstanceState\(\)](#) methods for your [Preference](#) subclass.

For example:

```

@Override
protected Parcelable onSaveInstanceState() {
    final Parcelable superState = super.onSaveInstanceState();
    // Check whether this Preference is persistent (continually saved)
    if (isPersistent()) {
        // No need to save instance state since it's persistent, use superclass
        return superState;
    }

    // Create instance of custom BaseSavedState
    final SavedState myState = new SavedState(superState);
    // Set the state's value with the class member that holds current setting value
    myState.value = mNewValue;
    return myState;
}

@Override
protected void onRestoreInstanceState(Parcelable state) {
    // Check whether we saved the state in onSaveInstanceState
    if (state == null || !state.getClass().equals(SavedState.class)) {
        // Didn't save the state, so call superclass
    }
}

```

```
super.onRestoreInstanceState(state);
return;
}

// Cast state to custom BaseSavedState and pass to superclass
SavedState myState = (SavedState) state;
super.onRestoreInstanceState(myState.getSuperState());

// Set this Preference's widget to reflect the restored state
mNumberPicker.setValue(myState.value);
}
```

# Dialogs

## In this document

1. [Creating a Dialog Fragment](#)
2. [Building an Alert Dialog](#)
  1. [Adding buttons](#)
  2. [Adding a list](#)
  3. [Creating a Custom Layout](#)
3. [Passing Events Back to the Dialog's Host](#)
4. [Showing a Dialog](#)
5. [Showing a Dialog Fullscreen or as an Embedded Fragment](#)
  1. [Showing an activity as a dialog on large screens](#)
6. [Dismissing a Dialog](#)

## Key classes

1. [DialogFragment](#)
2. [AlertDialog](#)

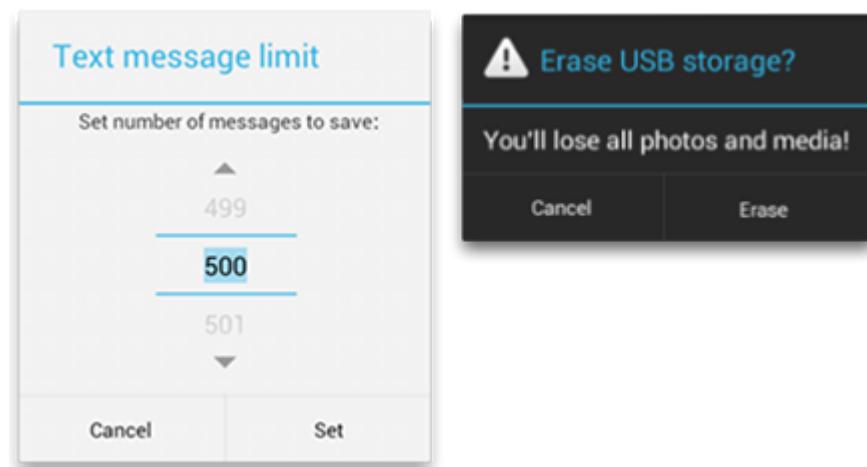
## See also

1. [Dialogs design guide](#)
2. [Pickers](#) (Date/Time dialogs)

A dialog is a small window that prompts the user to make a decision or enter additional information. A dialog does not fill the screen and is normally used for modal events that require users to take an action before they can proceed.

## Dialog Design

For information about how to design your dialogs, including recommendations for language, read the [Dialogs design guide](#).



The [Dialog](#) class is the base class for dialogs, but you should avoid instantiating [Dialog](#) directly. Instead, use one of the following subclasses:

## [AlertDialog](#)

A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.

## [DatePickerDialog or TimePickerDialog](#)

A dialog with a pre-defined UI that allows the user to select a date or time.

# Avoid ProgressDialog

Android includes another dialog class called [ProgressDialog](#) that shows a dialog with a progress bar. However, if you need to indicate loading or indeterminate progress, you should instead follow the design guidelines for [Progress & Activity](#) and use a [ProgressBar](#) in your layout.

These classes define the style and structure for your dialog, but you should use a [DialogFragment](#) as a container for your dialog. The [DialogFragment](#) class provides all the controls you need to create your dialog and manage its appearance, instead of calling methods on the [Dialog](#) object.

Using [DialogFragment](#) to manage the dialog ensures that it correctly handles lifecycle events such as when the user presses the *Back* button or rotates the screen. The [DialogFragment](#) class also allows you to reuse the dialog's UI as an embeddable component in a larger UI, just like a traditional [Fragment](#) (such as when you want the dialog UI to appear differently on large and small screens).

The following sections in this guide describe how to use a [DialogFragment](#) in combination with an [AlertDialog](#) object. If you'd like to create a date or time picker, you should instead read the [Pickers](#) guide.

**Note:** Because the [DialogFragment](#) class was originally added with Android 3.0 (API level 11), this document describes how to use the [DialogFragment](#) class that's provided with the [Support Library](#). By adding this library to your app, you can use [DialogFragment](#) and a variety of other APIs on devices running Android 1.6 or higher. If the minimum version your app supports is API level 11 or higher, then you can use the framework version of [DialogFragment](#), but be aware that the links in this document are for the support library APIs. When using the support library, be sure that you import `android.support.v4.app.DialogFragment` class and *not* `android.app.DialogFragment`.

# Creating a Dialog Fragment

You can accomplish a wide variety of dialog designs—including custom layouts and those described in the [Dialogs](#) design guide—by extending [DialogFragment](#) and creating a [AlertDialog](#) in the [onCreateDialog\(\)](#) callback method.

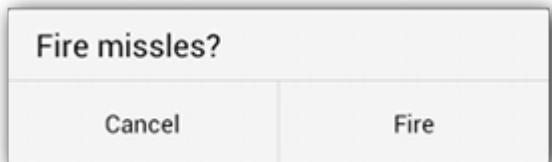
For example, here's a basic [AlertDialog](#) that's managed within a [DialogFragment](#):

```
public class FireMissilesDialogFragment extends DialogFragment {  
    @Override  
    public Dialog onCreateDialog(Bundle savedInstanceState) {  
        // Use the Builder class for convenient dialog construction  
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());  
        builder.setMessage(R.string.dialog_fire_missiles)  
            .setPositiveButton(R.string.fire, new DialogInterface.OnClickListener() {  
                public void onClick(DialogInterface dialog, int id) {  
                    // FIRE ZE MISSILES!  
                }  
            })  
            .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {  
                public void onClick(DialogInterface dialog, int id) {  
                    // CANCEL  
                }  
            })  
    }  
}
```

```

        public void onClick(DialogInterface dialog, int id) {
            // User cancelled the dialog
        }
    }) ;
    // Create the AlertDialog object and return it
    return builder.create();
}
}

```



**Figure 1.** A dialog with a message and two action buttons.

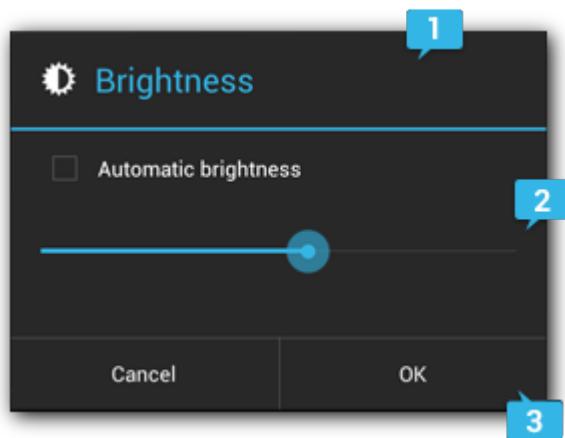
Now, when you create an instance of this class and call [show\(\)](#) on that object, the dialog appears as shown in figure 1.

The next section describes more about using the [AlertDialog.Builder](#) APIs to create the dialog.

Depending on how complex your dialog is, you can implement a variety of other callback methods in the [DialogFragment](#), including all the basic [fragment lifecycle methods](#).

## Building an Alert Dialog

The [AlertDialog](#) class allows you to build a variety of dialog designs and is often the only dialog class you'll need. As shown in figure 2, there are three regions of an alert dialog:



**Figure 2.** The layout of a dialog.

### 1. Title

This is optional and should be used only when the content area is occupied by a detailed message, a list, or custom layout. If you need to state a simple message or question (such as the dialog in figure 1), you don't need a title.

### 2. Content area

This can display a message, a list, or other custom layout.

### 3. Action buttons

There should be no more than three action buttons in a dialog.

The [AlertDialog.Builder](#) class provides APIs that allow you to create an [AlertDialog](#) with these kinds of content, including a custom layout.

To build an [AlertDialog](#):

```
// 1. Instantiate an AlertDialog.Builder with its constructor
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());

// 2. Chain together various setter methods to set the dialog characteristics
builder.setMessage(R.string.dialog_message)
    .setTitle(R.string.dialog_title);

// 3. Get the AlertDialog from create\(\)
AlertDialog dialog = builder.create();
```

The following topics show how to define various dialog attributes using the [AlertDialog.Builder](#) class.

## Adding buttons

To add action buttons like those in figure 2, call the [setPositiveButton\(\)](#) and [setNegativeButton\(\)](#) methods:

```
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
// Add the buttons
builder.setPositiveButton(R.string.ok, new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
        // User clicked OK button
    }
});
builder.setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
        // User cancelled the dialog
    }
});
// Set other dialog properties
...
// Create the AlertDialog
AlertDialog dialog = builder.create();
```

The [set...Button\(\)](#) methods require a title for the button (supplied by a [string resource](#)) and a [DialogInterface.OnClickListener](#) that defines the action to take when the user presses the button.

There are three different action buttons you can add:

### Positive

You should use this to accept and continue with the action (the "OK" action).

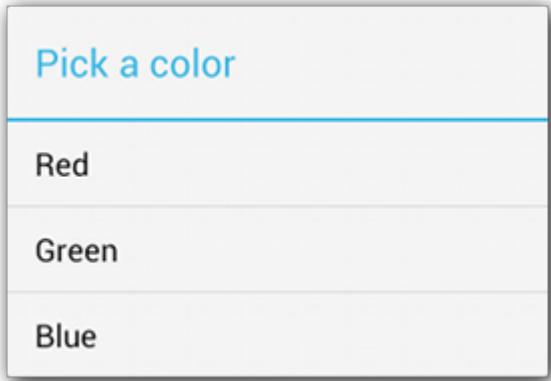
## Negative

You should use this to cancel the action.

## Neutral

You should use this when the user may not want to proceed with the action, but doesn't necessarily want to cancel. It appears between the positive and negative buttons. For example, the action might be "Remind me later."

You can add only one of each button type to an [AlertDialog](#). That is, you cannot have more than one "positive" button.



**Figure 3.** A dialog with a title and list.

## Adding a list

There are three kinds of lists available with the [AlertDialog](#) APIs:

- A traditional single-choice list
- A persistent single-choice list (radio buttons)
- A persistent multiple-choice list (checkboxes)

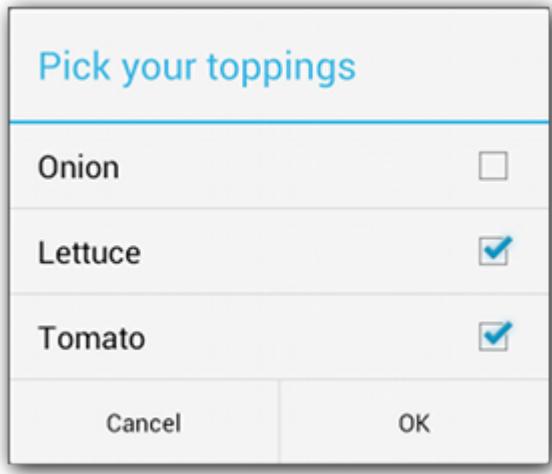
To create a single-choice list like the one in figure 3, use the [setItems\(\)](#) method:

```
@Override  
public Dialog onCreateDialog(Bundle savedInstanceState) {  
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());  
    builder.setTitle(R.string.pick_color);  
    builder.setItems(R.array.colors_array, new DialogInterface.OnClickListener()  
        public void onClick(DialogInterface dialog, int which) {  
            // The 'which' argument contains the index position  
            // of the selected item  
        }  
    );  
    return builder.create();  
}
```

Because the list appears in the dialog's content area, the dialog cannot show both a message and a list and you should set a title for the dialog with [setTitle\(\)](#). To specify the items for the list, call [setItems\(\)](#), passing an array. Alternatively, you can specify a list using [setAdapter\(\)](#). This allows you to back the list with dynamic data (such as from a database) using a [ListAdapter](#).

If you choose to back your list with a [ListAdapter](#), always use a [Loader](#) so that the content loads asynchronously. This is described further in [Building Layouts with an Adapter](#) and the [Loaders](#) guide.

**Note:** By default, touching a list item dismisses the dialog, unless you're using one of the following persistent choice lists.



**Figure 4.** A list of multiple-choice items.

### Adding a persistent multiple-choice or single-choice list

To add a list of multiple-choice items (checkboxes) or single-choice items (radio buttons), use the [setMultiChoiceItems\(\)](#) or [setSingleChoiceItems\(\)](#) methods, respectively.

For example, here's how you can create a multiple-choice list like the one shown in figure 4 that saves the selected items in an [ArrayList](#):

```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    mSelectedItems = new ArrayList(); // Where we track the selected items
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    // Set the dialog title
    builder.setTitle(R.string.pick_toppings)
    // Specify the list array, the items to be selected by default (null for no
    // and the listener through which to receive callbacks when items are selected
    .setMultiChoiceItems(R.array.toppings, null,
        new DialogInterface.OnMultiChoiceClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which,
                boolean isChecked) {
                if (isChecked) {
                    // If the user checked the item, add it to the selected
                    mSelectedItems.add(which);
                } else if (mSelectedItems.contains(which)) {
                    // Else, if the item is already in the array, remove it
                    mSelectedItems.remove(Integer.valueOf(which));
                }
            }
        })
    // Set the action buttons
    .setPositiveButton(R.string.ok, new DialogInterface.OnClickListener()
```

```

        @Override
        public void onClick(DialogInterface dialog, int id) {
            // User clicked OK, so save the mSelectedItems results somewhere
            // or return them to the component that opened the dialog
            ...
        }
    }

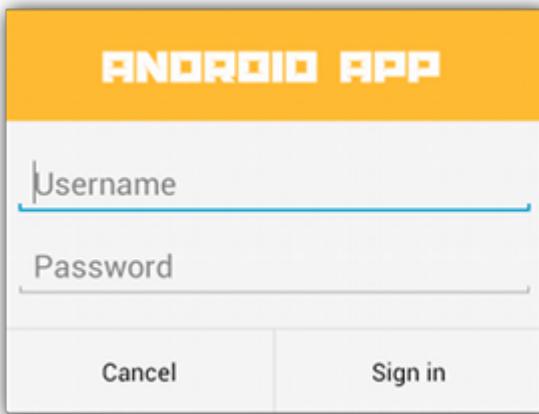
    .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int id) {
            ...
        }
    });
}

return builder.create();
}

```

Although both a traditional list and a list with radio buttons provide a "single choice" action, you should use [setSingleChoiceItems\(\)](#) if you want to persist the user's choice. That is, if opening the dialog again later should indicate what the user's current choice is, then you create a list with radio buttons.

## Creating a Custom Layout



**Figure 5.** A custom dialog layout.

If you want a custom layout in a dialog, create a layout and add it to an [AlertDialog](#) by calling [setView\(\)](#) on your [AlertDialog.Builder](#) object.

By default, the custom layout fills the dialog window, but you can still use [AlertDialog.Builder](#) methods to add buttons and a title.

For example, here's the layout file for the dialog in Figure 5:

res/layout/dialog\_signin.xml

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <ImageView
        android:src="@drawable/header_logo"

```

```

        android:layout_width="match_parent"
        android:layout_height="64dp"
        android:scaleType="center"
        android:background="#FFFFBB33"
        android:contentDescription="@string/app_name" />
<EditText
    android:id="@+id/username"
    android:inputType="textEmailAddress"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:layout_marginLeft="4dp"
    android:layout_marginRight="4dp"
    android:layout_marginBottom="4dp"
    android:hint="@string/username" />
<EditText
    android:id="@+id/password"
    android:inputType="textPassword"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="4dp"
    android:layout_marginLeft="4dp"
    android:layout_marginRight="4dp"
    android:layout_marginBottom="16dp"
    android:fontFamily="sans-serif"
    android:hint="@string/password"/>
</LinearLayout>

```

**Tip:** By default, when you set an [EditText](#) element to use the "textPassword" input type, the font family is set to monospace, so you should change its font family to "sans-serif" so that both text fields use a matching font style.

To inflate the layout in your [DialogFragment](#), get a [LayoutInflater](#) with [getLayoutInflater\(\)](#) and call [inflate\(\)](#), where the first parameter is the layout resource ID and the second parameter is a parent view for the layout. You can then call [setView\(\)](#) to place the layout in the dialog.

```

@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    // Get the layout inflator
    LayoutInflator inflater = getActivity().getLayoutInflater();

    // Inflate and set the layout for the dialog
    // Pass null as the parent view because its going in the dialog layout
    builder.setView(inflater.inflate(R.layout.dialog_signin, null))
    // Add action buttons
        .setPositiveButton(R.string.signin, new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int id) {
                // sign in the user ...
            }
        })
        .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) {

```

```

        LoginDialogFragment.this.getDialog().cancel();
    }
})
return builder.create();
}

```

**Tip:** If you want a custom dialog, you can instead display an [Activity](#) as a dialog instead of using the [Dialog](#) APIs. Simply create an activity and set its theme to [Theme.Holo.Dialog](#) in the [<activity>](#) manifest element:

```
<activity android:theme="@android:style/Theme.Holo.Dialog" >
```

That's it. The activity now displays in a dialog window instead of fullscreen.

## Passing Events Back to the Dialog's Host

When the user touches one of the dialog's action buttons or selects an item from its list, your [DialogFragment](#) might perform the necessary action itself, but often you'll want to deliver the event to the activity or fragment that opened the dialog. To do this, define an interface with a method for each type of click event. Then implement that interface in the host component that will receive the action events from the dialog.

For example, here's a [DialogFragment](#) that defines an interface through which it delivers the events back to the host activity:

```

public class NoticeDialogFragment extends DialogFragment {

    /* The activity that creates an instance of this dialog fragment must
     * implement this interface in order to receive event callbacks.
     * Each method passes the DialogFragment in case the host needs to query it
    */
    public interface NoticeDialogListener {
        public void onDialogPositiveClick(DialogFragment dialog);
        public void onDialogNegativeClick(DialogFragment dialog);
    }

    // Use this instance of the interface to deliver action events
    NoticeDialogListener mListener;

    // Override the Fragment.onAttach() method to instantiate the NoticeDialogI
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        // Verify that the host activity implements the callback interface
        try {
            // Instantiate the NoticeDialogListener so we can send events to th
            mListener = (NoticeDialogListener) activity;
        } catch (ClassCastException e) {
            // The activity doesn't implement the interface, throw exception
            throw new ClassCastException(activity.toString()
                + " must implement NoticeDialogListener");
        }
    }
    ...
}

```

The activity hosting the dialog creates an instance of the dialog with the dialog fragment's constructor and receives the dialog's events through an implementation of the `NoticeDialogListener` interface:

```
public class MainActivity extends FragmentActivity
    implements NoticeDialogFragment.NoticeDialogListener{
    ...
    public void showNoticeDialog() {
        // Create an instance of the dialog fragment and show it
        DialogFragment dialog = new NoticeDialogFragment();
        dialog.show(getSupportFragmentManager(), "NoticeDialogFragment");
    }
    ...
    // The dialog fragment receives a reference to this Activity through the
    // Fragment.onAttach() callback, which it uses to call the following method
    // defined by the NoticeDialogFragment.NoticeDialogListener interface
    @Override
    public void onDialogPositiveClick(DialogFragment dialog) {
        // User touched the dialog's positive button
        ...
    }
    @Override
    public void onDialogNegativeClick(DialogFragment dialog) {
        // User touched the dialog's negative button
        ...
    }
}
```

Because the host activity implements the `NoticeDialogListener`—which is enforced by the [onAt-tach\(\)](#) callback method shown above—the dialog fragment can use the interface callback methods to deliver click events to the activity:

```
public class NoticeDialogFragment extends DialogFragment {
    ...
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Build the dialog and set up the button click handlers
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setMessage(R.string.dialog_fire_missiles)
            .setPositiveButton(R.string.fire, new DialogInterface.OnClickListener {
                public void onClick(DialogInterface dialog, int id) {
                    // Send the positive button event back to the host activity
                    mListener.onDialogPositiveClick(NoticeDialogFragment.this);
                }
            })
            .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener {
                public void onClick(DialogInterface dialog, int id) {
                    // Send the negative button event back to the host activity
                    mListener.onDialogPositiveClick(NoticeDialogFragment.this);
                }
            });
        return builder.create();
    }
}
```

```
    }  
}
```

## Showing a Dialog

When you want to show your dialog, create an instance of your [DialogFragment](#) and call [show\(\)](#), passing the [FragmentManager](#) and a tag name for the dialog fragment.

You can get the [FragmentManager](#) by calling [getSupportFragmentManager\(\)](#) from the [FragmentActivity](#) or [getFragmentManager\(\)](#) from a [Fragment](#). For example:

```
public void confirmFireMissiles() {  
    DialogFragment newFragment = new FireMissilesDialogFragment();  
    newFragment.show(getSupportFragmentManager(), "missiles");  
}
```

The second argument, "missiles", is a unique tag name that the system uses to save and restore the fragment state when necessary. The tag also allows you to get a handle to the fragment by calling [findFragmentByTag\(\)](#).

## Showing a Dialog Fullscreen or as an Embedded Fragment

You might have a UI design in which you want a piece of the UI to appear as a dialog in some situations, but as a full screen or embedded fragment in others (perhaps depending on whether the device is a large screen or small screen). The [DialogFragment](#) class offers you this flexibility because it can still behave as an embeddable [Fragment](#).

However, you cannot use [AlertDialog.Builder](#) or other [Dialog](#) objects to build the dialog in this case. If you want the [DialogFragment](#) to be embeddable, you must define the dialog's UI in a layout, then load the layout in the [onCreateView\(\)](#) callback.

Here's an example [DialogFragment](#) that can appear as either a dialog or an embeddable fragment (using a layout named `purchase_items.xml`):

```
public class CustomDialogFragment extends DialogFragment {  
    /** The system calls this to get the DialogFragment's layout, regardless  
     * of whether it's being displayed as a dialog or an embedded fragment. */  
    @Override  
    public View onCreateView(LayoutInflater inflater, ViewGroup container,  
        Bundle savedInstanceState) {  
        // Inflate the layout to use as dialog or embedded fragment  
        return inflater.inflate(R.layout.purchase_items, container, false);  
    }  
  
    /** The system calls this only when creating the layout in a dialog. */  
    @Override  
    public Dialog onCreateDialog(Bundle savedInstanceState) {  
        // The only reason you might override this method when using onCreateView  
        // to modify any dialog characteristics. For example, the dialog includes  
        // a title by default, but your custom layout might not need it. So here  
        // remove the dialog title, but you must call the superclass to get the  
        Dialog dialog = super.onCreateDialog(savedInstanceState);  
        return dialog;  
    }  
}
```

```

        dialog.requestWindowFeature(Window.FEATURE_NO_TITLE);
        return dialog;
    }
}

```

And here's some code that decides whether to show the fragment as a dialog or a fullscreen UI, based on the screen size:

```

public void showDialog() {
    FragmentManager fragmentManager = getSupportFragmentManager();
    CustomDialogFragment newFragment = new CustomDialogFragment();

    if (mIsLargeLayout) {
        // The device is using a large layout, so show the fragment as a dialog
        newFragment.show(fragmentManager, "dialog");
    } else {
        // The device is smaller, so show the fragment fullscreen
        FragmentTransaction transaction = fragmentManager.beginTransaction();
        // For a little polish, specify a transition animation
        transaction.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN);
        // To make it fullscreen, use the 'content' root view as the container
        // for the fragment, which is always the root view for the activity
        transaction.add(android.R.id.content, newFragment)
            .addToBackStack(null).commit();
    }
}

```

For more information about performing fragment transactions, see the [Fragments](#) guide.

In this example, the `mIsLargeLayout` boolean specifies whether the current device should use the app's large layout design (and thus show this fragment as a dialog, rather than fullscreen). The best way to set this kind of boolean is to declare a [bool resource value](#) with an [alternative resource](#) value for different screen sizes. For example, here are two versions of the bool resource for different screen sizes:

`res/values/bools.xml`

```

<!-- Default boolean values -->
<resources>
    <bool name="large_layout">false</bool>
</resources>

```

`res/values-large/bools.xml`

```

<!-- Large screen boolean values -->
<resources>
    <bool name="large_layout">true</bool>
</resources>

```

Then you can initialize the `mIsLargeLayout` value during the activity's [onCreate\(\)](#) method:

```

boolean mIsLargeLayout;

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

```

```
setContentView(R.layout.activity_main);  
  
mIsLargeLayout = getResources().getBoolean(R.bool.large_layout);  
}
```

## Showing an activity as a dialog on large screens

Instead of showing a dialog as a fullscreen UI when on small screens, you can accomplish the same result by showing an [Activity](#) as a dialog when on large screens. Which approach you choose depends on your app design, but showing an activity as a dialog is often useful when your app is already designed for small screens and you'd like to improve the experience on tablets by showing a short-lived activity as a dialog.

To show an activity as a dialog only when on large screens, apply the [Theme.Holo.DialogWhenLarge](#) theme to the [`<activity>`](#) manifest element:

```
<activity android:theme="@android:style/Theme.Holo.DialogWhenLarge" >
```

For more information about styling your activities with themes, see the [Styles and Themes](#) guide.

## Dismissing a Dialog

When the user touches any of the action buttons created with an [AlertDialog.Builder](#), the system dismisses the dialog for you.

The system also dismisses the dialog when the user touches an item in a dialog list, except when the list uses radio buttons or checkboxes. Otherwise, you can manually dismiss your dialog by calling [dismiss\(\)](#) on your [DialogFragment](#).

In case you need to perform certain actions when the dialog goes away, you can implement the [onDismiss\(\)](#) method in your [DialogFragment](#).

You can also *cancel* a dialog. This is a special event that indicates the user explicitly left the dialog without completing the task. This occurs if the user presses the *Back* button, touches the screen outside the dialog area, or if you explicitly call [cancel\(\)](#) on the [Dialog](#) (such as in response to a "Cancel" button in the dialog).

As shown in the example above, you can respond to the cancel event by implementing [onCancel\(\)](#) in your [DialogFragment](#) class.

**Note:** The system calls [onDismiss\(\)](#) upon each event that invokes the [onCancel\(\)](#) callback. However, if you call [Dialog.dismiss\(\)](#) or [DialogFragment.dismiss\(\)](#), the system calls [onDismiss\(\)](#) but not [onCancel\(\)](#). So you should generally call [dismiss\(\)](#) when the user presses the *positive* button in your dialog in order to remove the dialog from view.

# Notifications

## In this document

1. [Notification Display Elements](#)
  1. [Normal view](#)
  2. [Big view](#)
2. [Creating a Notification](#)
  1. [Required notification contents](#)
  2. [Optional notification contents and settings](#)
  3. [Notification actions](#)
  4. [Creating a simple notification](#)
  5. [Applying a big view style to a notification](#)
  6. [Handling compatibility](#)
3. [Managing Notifications](#)
  1. [Updating notifications](#)
  2. [Removing notifications](#)
4. [Preserving Navigation when Starting an Activity](#)
  1. [Setting up a regular activity PendingIntent](#)
  2. [Setting up a special activity PendingIntent](#)
5. [Displaying Progress in a Notification](#)
  1. [Displaying a fixed-duration progress indicator](#)
  2. [Displaying a continuing activity indicator](#)
6. [Custom Notification Layouts](#)

## Key classes

1. [NotificationManager](#)
2. [NotificationCompat](#)

## Videos

1. [Notifications in 4.1](#)

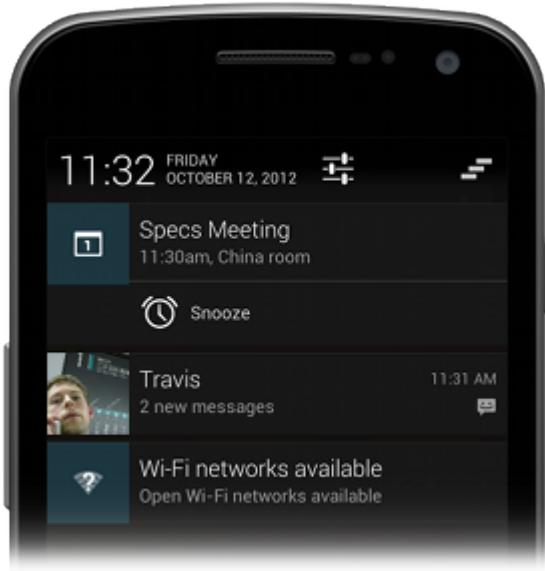
## See also

1. [Android Design: 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.



**Figure 1.** Notifications in the notification area.



**Figure 2.** Notifications in the notification drawer.

## Notification Design

Notifications, as an important part of the Android UI, have their own design guidelines. To learn how to design notifications and their interactions, read the [Android Design Guide Notifications](#) topic.

**Note:** Except where noted, this guide refers to the [NotificationCompat.Builder](#) class in the version 4 [Support Library](#). The class [Notification.Builder](#) was added in Android 3.0.

# Notification Display Elements

Notifications in the notification drawer can appear in one of two visual styles, depending on the version and the state of the drawer:

### Normal view

The standard view of the notifications in the notification drawer.

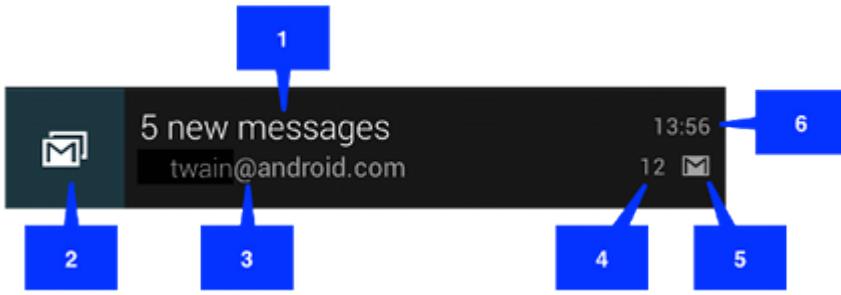
### Big view

A large view that's visible when the notification is expanded. Big view is part of the expanded notification feature available as of Android 4.1.

These styles are described in the following sections.

### Normal view

A notification in normal view appears in an area that's up to 64 dp tall. Even if you create a notification with a big view style, it will appear in normal view until it's expanded. This is an example of a normal view:



**Figure 3.** Notification in normal view.

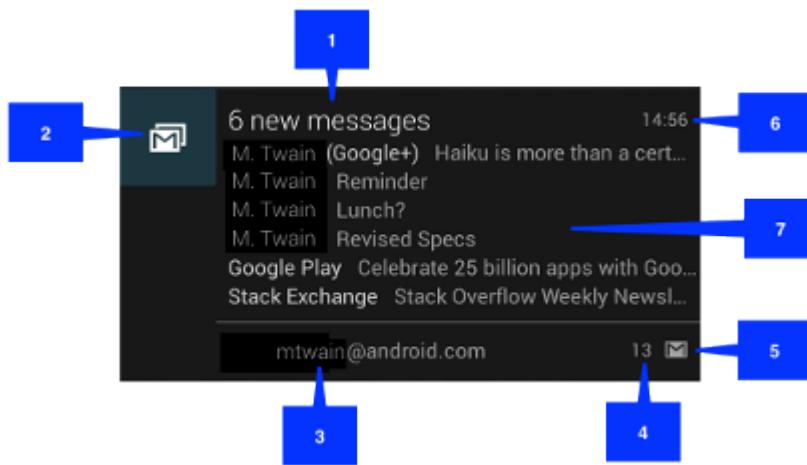
The callouts in the illustration refer to the following:

1. Content title
2. Large icon
3. Content text
4. Content info
5. Small icon
6. Time that the notification was issued. You can set an explicit value with [`setWhen\(\)`](#); if you don't it defaults to the time that the system received the notification.

## Big view

A notification's big view appears only when the notification is expanded, which happens when the notification is at the top of the notification drawer, or when the user expands the notification with a gesture. Expanded notifications are available starting with Android 4.1.

The following screenshot shows an inbox-style notification:



**Figure 4.** Big view notification.

Notice that the big view shares most of its visual elements with the normal view. The only difference is callout number 7, the details area. Each big view style sets this area in a different way. The available styles are:

### Big picture style

The details area contains a bitmap up to 256 dp tall in its detail section.

### Big text style

Displays a large text block in the details section.

## Inbox style

Displays lines of text in the details section.

All of the big view styles also have the following content options that aren't available in normal view:

## Big content title

Allows you to override the normal view's content title with a title that appears only in the expanded view.

## Summary text

Allows you to add a line of text below the details area.

Applying a big view style to a notification is described in the section [Applying a big view style to a notification](#).

# Creating a Notification

You specify the UI information and actions for a notification in a [NotificationCompat.Builder](#) object. To create the notification itself, you 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\(\)](#).

## Required notification contents

A [Notification](#) object *must* contain the following:

- A small icon, set by [setSmallIcon\(\)](#)
- A title, set by [setContentTitle\(\)](#)
- Detail text, set by [setContentText\(\)](#)

## Optional notification contents and settings

All other notification settings and contents are optional. To learn more about them, see the reference documentation for [NotificationCompat.Builder](#).

## Notification actions

Although they're optional, you should add at least one action to your 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.

A notification can provide multiple actions. You should always define the action that's triggered when the user clicks the notification; usually this action opens an [Activity](#) in your application. You can also add buttons to the notification that perform additional actions such as snoozing an alarm or responding immediately to a text message; this feature is available as of Android 4.1. If you use additional action buttons, you must also make their functionality available in an [Activity](#) in your app; see the section [Handling compatibility](#) for more details.

Inside a [Notification](#), the action itself 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\(\)](#).

Starting an [Activity](#) when the user clicks the notification is the most common action scenario. You can also start an [Activity](#) when the user dismisses an [Activity](#). In Android 4.1 and later, you can start an [Activity](#) from an action button. To learn more, read the reference guide for [NotificationCompat.Builder](#).

## Creating a simple notification

The following snippet illustrates a simple notification that specifies an activity to open when the user clicks the notification. Notice that the code creates a [TaskStackBuilder](#) object and uses it to create the [PendingIntent](#) for the action. This pattern is explained in more detail in the section [Preserving Navigation when Starting an Activity](#):

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("My notification")
        .setContentText("Hello World!");
// Creates an explicit intent for an Activity in your app
Intent resultIntent = new Intent(this, ResultActivity.class);

// The stack builder object 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.
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
// Adds the back stack for the Intent (but not the Intent itself)
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);
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// mId allows you to update the notification later on.
mNotificationManager.notify(mId, mBuilder.build());
```

That's it. Your user has now been notified.

## Applying a big view style to a notification

To have a notification appear in a big view when it's expanded, first create a [NotificationCompat.Builder](#) object with the normal view options you want. Next, call [Builder.setStyle\(\)](#) with a big view style object as its argument.

Remember that expanded notifications are not available on platforms prior to Android 4.1. To learn how to handle notifications for Android 4.1 and for earlier platforms, read the section [Handling compatibility](#).

For example, the following code snippet demonstrates how to alter the notification created in the previous snippet to use the Inbox big view style:

```

NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("Event tracker")
    .setContentText("Events received")
NotificationCompat.InboxStyle inboxStyle =
    new NotificationCompat.InboxStyle();
String[] events = new String[6];
// Sets a title for the Inbox style big view
inboxStyle.setBigContentTitle("Event tracker details:");
...
// Moves events into the big view
for (int i=0; i < events.length; i++) {

    inboxStyle.addLine(events[i]);
}
// Moves the big view style object into the notification object.
mBuilder.setStyle(inboxStyle);
...
// Issue the notification here.

```

## Handling compatibility

Not all notification features are available for a particular version, even though the methods to set them are in the support library class [NotificationCompat.Builder](#). For example, action buttons, which depend on expanded notifications, only appear on Android 4.1 and higher, because expanded notifications themselves are only available on Android 4.1 and higher.

To ensure the best compatibility, create notifications with [NotificationCompat](#) and its subclasses, particularly [NotificationCompat.Builder](#). In addition, follow this process when you implement a notification:

1. Provide all of the notification's functionality to all users, regardless of the version they're using. To do this, verify that all of the functionality is available from an [Activity](#) in your app. You may want to add a new [Activity](#) to do this.

For example, if you want to use [addAction\(\)](#) to provide a control that stops and starts media playback, first implement this control in an [Activity](#) in your app.

2. Ensure that all users can get to the functionality in the [Activity](#), by having it start when users click the notification. To do this, create a [PendingIntent](#) for the [Activity](#). Call [setContentIntent\(\)](#) to add the [PendingIntent](#) to the notification.
3. Now add the expanded notification features you want to use to the notification. Remember that any functionality you add also has to be available in the [Activity](#) that starts when users click the notification.

## Managing Notifications

When you need to issue a notification multiple times for the same type of event, you should avoid making a completely new notification. Instead, you should consider updating a previous notification, either by changing some of its values or by adding to it, or both.

For example, Gmail notifies the user that new emails have arrived by increasing its count of unread messages and by adding a summary of each email to the notification. This is called "stacking" the notification; it's described in more detail in the [Notifications](#) Design guide.

**Note:** This Gmail feature requires the "inbox" big view style, which is part of the expanded notification feature available starting in Android 4.1.

The following section describes how to update notifications and also how to remove them.

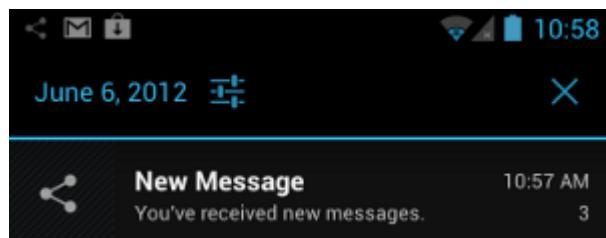
## Updating notifications

To set up a notification so it can be updated, issue it with a notification ID by calling [NotificationManager.notify\(ID, notification\)](#). To update this notification once you've issued it, update or create a [NotificationCompat.Builder](#) object, build a [Notification](#) object from it, and issue the [Notification](#) with the same ID you used previously. If the previous notification is still visible, the system updates it from the contents of the [Notification](#) object. If the previous notification has been dismissed, a new notification is created instead.

The following snippet demonstrates a notification that is updated to reflect the number of events that have occurred. It stacks the notification, showing a summary:

```
mNotificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// Sets an ID for the notification, so it can be updated
int notifyID = 1;
mNotifyBuilder = new NotificationCompat.Builder(this)
        .setContentTitle("New Message")
        .setContentText("You've received new messages.")
        .setSmallIcon(R.drawable.ic_notify_status)
numMessages = 0;
// Start of a loop that processes data and then notifies the user
...
    mNotifyBuilder.setContentText(currentText)
        .setNumber(++numMessages);
// Because the ID remains unchanged, the existing notification is
// updated.
mNotificationManager.notify(
        notifyID,
        mNotifyBuilder.build());
...
```

This produces a notification that looks like this:



**Figure 5.** Updated notification displayed in the notification drawer.

## Removing notifications

Notifications remain visible until one of the following happens:

- The user dismisses the notification either individually or by using "Clear All" (if the notification can be cleared).
- The user clicks the notification, and you called `setAutoCancel()` when you created the notification.
- You call `cancel()` for a specific notification ID. This method also deletes ongoing notifications.
- You call `cancelAll()`, which removes all of the notifications you previously issued.

## Preserving Navigation when Starting an Activity

When you start an [Activity](#) from a notification, you must preserve the user's expected navigation experience. Clicking **Back** should take the user back through the application's normal work flow to the Home screen, and clicking **Recents** should show the [Activity](#) as a separate task. To preserve the navigation experience, you should start the [Activity](#) in a fresh task. How you set up the [PendingIntent](#) to give you a fresh task depends on the nature of the [Activity](#) you're starting. There are two general situations:

### Regular activity

You're starting an [Activity](#) that's part of the application's normal workflow. In this situation, set up the [PendingIntent](#) to start a fresh task, and provide the [PendingIntent](#) with a back stack that reproduces the application's normal **Back** behavior.

Notifications from the Gmail app demonstrate this. When you click a notification for a single email message, you see the message itself. Touching **Back** takes you backwards through Gmail to the Home screen, just as if you had entered Gmail from the Home screen rather than entering it from a notification.

This happens regardless of the application you were in when you touched the notification. For example, if you're in Gmail composing a message, and you click a notification for a single email, you go immediately to that email. Touching **Back** takes you to the inbox and then the Home screen, rather than taking you to the message you were composing.

### Special activity

The user only sees this [Activity](#) if it's started from a notification. In a sense, the [Activity](#) extends the notification by providing information that would be hard to display in the notification itself. For this situation, set up the [PendingIntent](#) to start in a fresh task. There's no need to create a back stack, though, because the started [Activity](#) isn't part of the application's activity flow. Clicking **Back** will still take the user to the Home screen.

## Setting up a regular activity PendingIntent

To set up a [PendingIntent](#) that starts a direct entry [Activity](#), follow these steps:

1. Define your application's [Activity](#) hierarchy in the manifest.
  1. Add support for Android 4.0.3 and earlier. To do this, specify the parent of the [Activity](#) you're starting by adding a `<meta-data>` element as the child of the `<activity>`.

For this element, set `android:name="android.support.PARENT_ACTIVITY"`. Set `android:value="<parent_activity_name>"` where `<parent_activity_name>` is the value of `android:name` for the parent `<activity>` element. See the following XML for an example.

2. Also add support for Android 4.1 and later. To do this, add the `an-droid:parentActivityName` attribute to the `<activity>` element of the `Activity` you're starting.

The final XML should look like this:

```
<activity
    android:name=".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=".ResultActivity"
    android:parentActivityName=".MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

2. Create a back stack based on the `Intent` that starts the `Activity`:

1. Create the `Intent` to start the `Activity`.
2. Create a stack builder by calling `TaskStackBuilder.create()`.
3. Add the back stack to the stack builder by calling `addParentStack()`. For each `Activity` in the hierarchy you've defined in the manifest, the back stack contains an `Intent` object that starts the `Activity`. This method also adds flags that start the stack in a fresh task.

**Note:** Although the argument to `addParentStack()` is a reference to the started `Activity`, the method call doesn't add the `Intent` that starts the `Activity`. Instead, that's taken care of in the next step.

4. Add the `Intent` that starts the `Activity` from the notification, by calling `addNextIntent()`. Pass the `Intent` you created in the first step as the argument to `addNextIntent()`.
5. If you need to, add arguments to `Intent` objects on the stack by calling `TaskStackBuilder.editIntentAt()`. This is sometimes necessary to ensure that the target `Activity` displays meaningful data when the user navigates to it using *Back*.
6. Get a `PendingIntent` for this back stack by calling `getPendingIntent()`. You can then use this `PendingIntent` as the argument to `setContentIntent()`.

The following code snippet demonstrates the process:

```
...
Intent resultIntent = new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
// Adds the back stack
stackBuilder.addParentStack(ResultActivity.class);
// Adds the Intent to the top of the stack
stackBuilder.addNextIntent(resultIntent);
// Gets a PendingIntent containing the entire back stack
PendingIntent resultPendingIntent =
```

```

stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);
...
NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
builder.setContentIntent(resultPendingIntent);
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mNotificationManager.notify(id, builder.build());

```

## Setting up a special activity PendingIntent

The following section describes how to set up a special activity [PendingIntent](#).

A special [Activity](#) doesn't need a back stack, so you don't have to define its [Activity](#) hierarchy in the manifest, and you don't have to call [addParentStack\(\)](#) to build a back stack. Instead, use the manifest to set up the [Activity](#) task options, and create the [PendingIntent](#) by calling [getActivity\(\)](#):

1. In your manifest, add the following attributes to the [`<activity>`](#) element for the [Activity](#) [`android:name="activityclass"`](#)

The activity's fully-qualified class name.

### [`android:taskAffinity=""`](#)

Combined with the [FLAG\\_ACTIVITY\\_NEW\\_TASK](#) flag that you set in code, this ensures that this [Activity](#) doesn't go into the application's default task. Any existing tasks that have the application's default affinity are not affected.

### [`android:excludeFromRecents="true"`](#)

Excludes the new task from *Recents*, so that the user can't accidentally navigate back to it.

This snippet shows the element:

```

<activity
    android:name=".ResultActivity"
    ...
    android:launchMode="singleTask"
    android:taskAffinity=""
    android:excludeFromRecents="true">
</activity>
...

```

2. Build and issue the notification:

1. Create an [Intent](#) that starts the [Activity](#).
2. Set the [Activity](#) to start in a new, empty task by calling [setFlags\(\)](#) with the flags [FLAG\\_ACTIVITY\\_NEW\\_TASK](#) and [FLAG\\_ACTIVITY\\_CLEAR\\_TASK](#).
3. Set any other options you need for the [Intent](#).
4. Create a [PendingIntent](#) from the [Intent](#) by calling [getActivity\(\)](#). You can then use this [PendingIntent](#) as the argument to [setContentIntent\(\)](#).

The following code snippet demonstrates the process:

```

// Instantiate a Builder object.
NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
// Creates an Intent for the Activity
Intent notifyIntent =
    new Intent(new ComponentName(this, ResultActivity.class));

```

```

// Sets the Activity to start in a new, empty task
notifyIntent.setFlags(FLAG_ACTIVITY_NEW_TASK | FLAG_ACTIVITY_CLEAR_TASK);
// Creates the PendingIntent
PendingIntent notifyIntent =
    PendingIntent.getActivity(
        this,
        0,
        notifyIntent
        PendingIntent.FLAG_UPDATE_CURRENT
    );

// Puts the PendingIntent into the notification builder
builder.setContentIntent(notifyIntent);
// Notifications are issued by sending them to the
// NotificationManager system service.
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// Builds an anonymous Notification object from the builder, and
// passes it to the NotificationManager
mNotificationManager.notify(id, builder.build());

```

## Displaying Progress in a Notification

Notifications can include an animated progress indicator that shows users the status of an ongoing operation. If you can estimate how long the operation takes and how much of it is complete at any time, use the "determinate" form of the indicator (a progress bar). If you can't estimate the length of the operation, use the "ineterminate" form of the indicator (an activity indicator).

Progress indicators are displayed with the platform's implementation of the [ProgressBar](#) class.

To use a progress indicator on platforms starting with Android 4.0, call [setProgress\(\)](#). For previous versions, you must create your own custom notification layout that includes a [ProgressBar](#) view.

The following sections describe how to display progress in a notification using [setProgress\(\)](#).

### Displaying a fixed-duration progress indicator

To display a determinate progress bar, add the bar to your notification by calling [setProgress\(\) setProgress\(max, progress, false\)](#) and then issue the notification. As your operation proceeds, increment progress, and update the notification. At the end of the operation, progress should equal max. A common way to call [setProgress\(\)](#) is to set max to 100 and then increment progress as a "percent complete" value for the operation.

You can either leave the progress bar showing when the operation is done, or remove it. In either case, remember to update the notification text to show that the operation is complete. To remove the progress bar, call [setProgress\(\) setProgress\(0, 0, false\)](#). For example:

```

...
mNotifyManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mBuilder = new NotificationCompat.Builder(this);
mBuilder.setContentTitle("Picture Download")
    .setContentText("Download in progress")

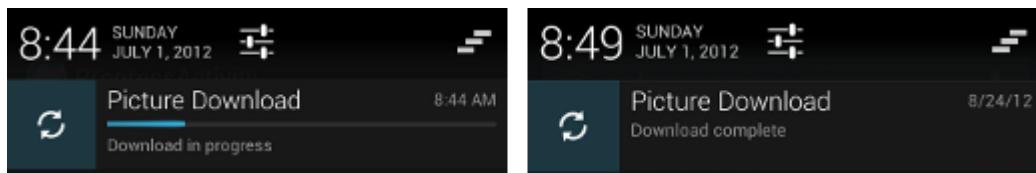
```

```

.setSmallIcon(R.drawable.ic_notification);
// Start a lengthy operation in a background thread
new Thread(
    new Runnable() {
        @Override
        public void run() {
            int incr;
            // Do the "lengthy" operation 20 times
            for (incr = 0; incr <= 100; incr+=5) {
                // Sets the progress indicator to a max value, the
                // current completion percentage, and "determinate"
                // state
                mBuilder.setProgress(100, incr, false);
                // Displays the progress bar for the first time.
                mNotifyManager.notify(0, mBuilder.build());
                // Sleeps the thread, simulating an operation
                // that takes time
                try {
                    // Sleep for 5 seconds
                    Thread.sleep(5*1000);
                } catch (InterruptedException e) {
                    Log.d(TAG, "sleep failure");
                }
            }
            // When the loop is finished, updates the notification
            mBuilder.setContentText("Download complete")
            // Removes the progress bar
            .setProgress(0,0,false);
            mNotifyManager.notify(ID, mBuilder.build());
        }
    }
)
// Starts the thread by calling the run() method in its Runnable
).start();

```

The resulting notifications are shown in figure 6. On the left side is a snapshot of the notification during the operation; on the right side is a snapshot of it after the operation has finished.



**Figure 6.** The progress bar during and after the operation.

## Displaying a continuing activity indicator

To display an indeterminate activity indicator, add it to your notification with [`setProgress\(0, 0, true\)`](#) (the first two arguments are ignored), and issue the notification. The result is an indicator that has the same style as a progress bar, except that its animation is ongoing.

Issue the notification at the beginning of the operation. The animation will run until you modify your notification. When the operation is done, call [`setProgress\(\)`](#) [`setProgress\(0, 0, false\)`](#) and then update the notification to remove the activity indicator. Always do this; otherwise, the animation will run even when

the operation is complete. Also remember to change the notification text to indicate that the operation is complete.

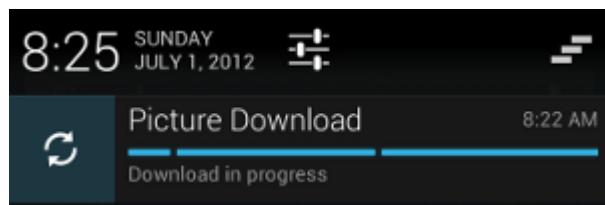
To see how activity indicators work, refer to the preceding snippet. Locate the following lines:

```
// Sets the progress indicator to a max value, the current completion  
// percentage, and "determinate" state  
mBuilder.setProgress(100, incr, false);  
// Issues the notification  
mNotifyManager.notify(0, mBuilder.build());
```

Replace the lines you've found with the following lines:

```
// Sets an activity indicator for an operation of indeterminate length  
mBuilder.setProgress(0, 0, true);  
// Issues the notification  
mNotifyManager.notify(0, mBuilder.build());
```

The resulting indicator is shown in figure 7:



**Figure 7.** An ongoing activity indicator.

## Custom Notification Layouts

The notifications framework allows you to define a custom notification layout, which defines the notification's appearance in a [RemoteViews](#) object. Custom layout notifications are similar to normal notifications, but they're based on a [RemoteViews](#) defined in a XML layout file.

The height available for a custom notification layout depends on the notification view. Normal view layouts are limited to 64 dp, and expanded view layouts are limited to 256 dp.

To define a custom notification layout, start by instantiating a [RemoteViews](#) object that inflates an XML layout file. Then, instead of calling methods such as [setContentTitle\(\)](#), call [setContent\(\)](#). To set content details in the custom notification, use the methods in [RemoteViews](#) to set the values of the view's children:

1. Create an XML layout for the notification in a separate file. You can use any file name you wish, but you must use the extension .xml
2. In your app, use [RemoteViews](#) methods to define your notification's icons and text. Put this [RemoteViews](#) object into your [NotificationCompat.Builder](#) by calling [setContent\(\)](#). Avoid setting a background [Drawable](#) on your [RemoteViews](#) object, because your text color may become unreadable.

The [RemoteViews](#) class also includes methods that you can use to easily add a [Chronometer](#) or [ProgressBar](#) to your notification's layout. For more information about creating custom layouts for your notification, refer to the [RemoteViews](#) reference documentation.

**Caution:** When you use a custom notification layout, take special care to ensure that your custom layout works with different device orientations and resolutions. While this advice applies to all View layouts, it's especially important for notifications because the space in the notification drawer is very restricted. Don't make your custom layout too complex, and be sure to test it in various configurations.

### Using style resources for custom notification text

Always use style resources for the text of a custom notification. The background color of the notification can vary across different devices and versions, and using style resources helps you account for this. Starting in Android 2.3, the system defined a style for the standard notification layout text. If you use the same style in applications that target Android 2.3 or higher, you'll ensure that your text is visible against the display background.

# Toasts

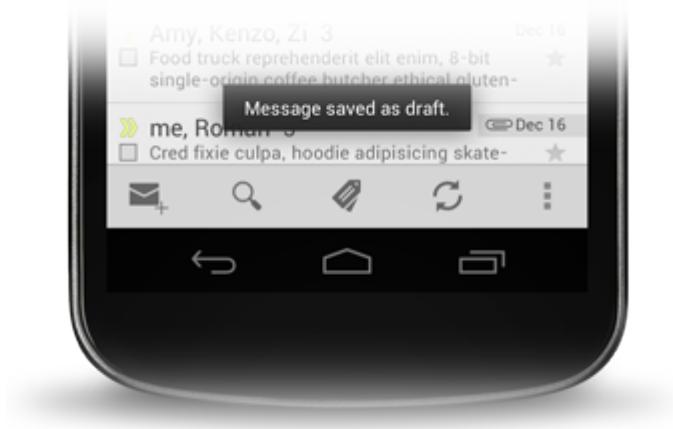
## In this document

1. [The Basics](#)
2. [Positioning your Toast](#)
3. [Creating a Custom Toast View](#)

## Key classes

1. [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.



If user response to a status message is required, consider instead using a [Notification](#).

## The Basics

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.

## Creating a Custom Toast View

If a simple text message isn't enough, you can create a customized layout for your toast notification. To create a custom layout, define a View layout, in XML or in your application code, and pass the root [View](#) object to the [setView\(View\)](#) method.

For example, you can create the layout for the toast visible in the screenshot to the right with the following XML (saved as *toast\_layout.xml*):

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/toast_layout_root"  
    android:orientation="horizontal"  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent"  
    android:padding="8dp"  
    android:background="#DAAA"  
    >  
    <ImageView android:src="@drawable/droid"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_marginRight="8dp"  
        />  
    <TextView android:id="@+id/text"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:textColor="#FFF"  
        />  
</LinearLayout>
```

Notice that the ID of the LinearLayout element is "toast\_layout\_root". You must use this ID to inflate the layout from the XML, as shown here:

```
LayoutInflater inflater = getLayoutInflater();  
View layout = inflater.inflate(R.layout.custom_toast,  
                               ( ViewGroup ) findViewById(R.id.toast_layout_root))  
  
TextView text = ( TextView ) layout.findViewById(R.id.text);  
text.setText("This is a custom toast");
```

```
Toast toast = new Toast(getApplicationContext());
toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(layout);
toast.show();
```

First, retrieve the [LayoutInflater](#) with [getLayoutInflater\(\)](#) (or [getSystemService\(\)](#)), and then inflate the layout from XML using [inflate\(int, ViewGroup\)](#). The first parameter is the layout resource ID and the second is the root View. You can use this inflated layout to find more View objects in the layout, so now capture and define the content for the ImageView and TextView elements. Finally, create a new Toast with [Toast\(Context\)](#) and set some properties of the toast, such as the gravity and duration. Then call [setView\(View\)](#) and pass it the inflated layout. You can now display the toast with your custom layout by calling [show\(\)](#).

**Note:** Do not use the public constructor for a Toast unless you are going to define the layout with [setView\(View\)](#). If you do not have a custom layout to use, you must use [makeText\(Context, int, int\)](#) to create the Toast.

# Search Overview

## Topics

1. [Creating a Search Interface](#)
2. [Adding Recent Query Suggestions](#)
3. [Adding Custom Suggestions](#)

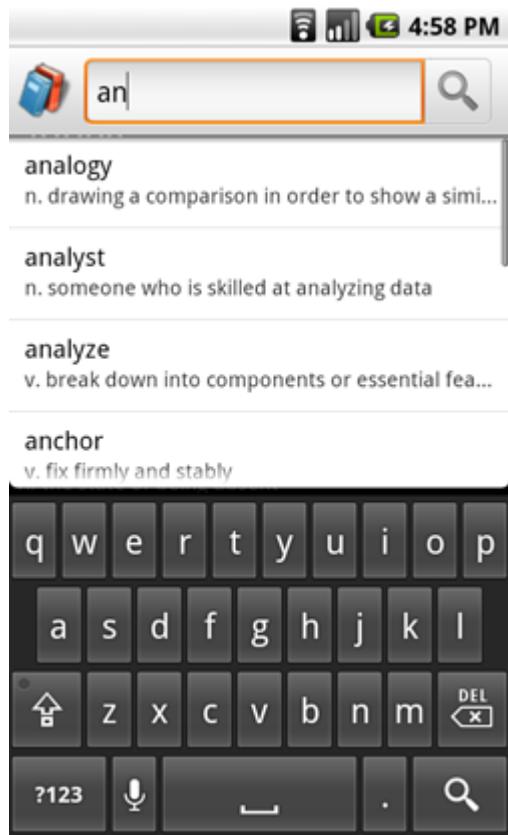
## Reference

1. [Searchable Configuration](#)

## Related samples

1. [Searchable Dictionary](#)

Search is a core user feature on Android. Users should be able to search any data that is available to them, whether the content is located on the device or the Internet. To help create a consistent search experience for users, Android provides a search framework that helps you implement search for your application.



**Figure 1.** Screenshot of a search dialog with custom search suggestions.

The search framework offers two modes of search input: a search dialog at the top of the screen or a search widget ([SearchView](#)) that you can embed in your activity layout. In either case, the Android system will assist your search implementation by delivering search queries to a specific activity that performs searches. You can also enable either the search dialog or widget to provide search suggestions as the user types. Figure 1 shows an example of the search dialog with optional search suggestions.

Once you've set up either the search dialog or the search widget, you can:

- Enable voice search
- Provide search suggestions based on recent user queries
- Provide custom search suggestions that match actual results in your application data
- Offer your application's search suggestions in the system-wide Quick Search Box

**Note:** The search framework does *not* provide APIs to search your data. To perform a search, you need to use APIs appropriate for your data. For example, if your data is stored in an SQLite database, you should use the [android.database.sqlite](#) APIs to perform searches.

Also, there is no guarantee that a device provides a dedicated SEARCH button that invokes the search interface in your application. When using the search dialog or a custom interface, you must provide a search button in your UI that activates the search interface. For more information, see [Invoking the search dialog](#).

The following documents show you how to use Android's framework to implement search:

### [Creating a Search Interface](#)

How to set up your application to use the search dialog or search widget.

### [Adding Recent Query Suggestions](#)

How to provide suggestions based on queries previously used.

### [Adding Custom Suggestions](#)

How to provide suggestions based on custom data from your application and also offer them in the system-wide Quick Search Box.

### [Searchable Configuration](#)

A reference document for the searchable configuration file (though the other documents also discuss the configuration file in terms of specific behaviors).

## Protecting User Privacy

When you implement search in your application, take steps to protect the user's privacy. Many users consider their activities on the phone—including searches—to be private information. To protect each user's privacy, you should abide by the following principles:

- **Don't send personal information to servers, but if you must, do not log it.**

Personal information is any information that can personally identify your users, such as their names, email addresses, billing information, or other data that can be reasonably linked to such information. If your application implements search with the assistance of a server, avoid sending personal information along with the search queries. For example, if you are searching for businesses near a zip code, you don't need to send the user ID as well; send only the zip code to the server. If you must send the personal information, you should not log it. If you must log it, protect that data very carefully and erase it as soon as possible.

- **Provide users with a way to clear their search history.**

The search framework helps your application provide context-specific suggestions while the user types. Sometimes these suggestions are based on previous searches or other actions taken by the user in an earlier session. A user might not wish for previous searches to be revealed to other device users, for instance, if the user shares the device with a friend. If your application provides suggestions that can reveal previous search activities, you should implement the ability for the user to clear the search history. If you are using [SearchRecentSuggestions](#), you can simply call the [clearHistory\(\)](#)

method. If you are implementing custom suggestions, you'll need to provide a similar "clear history" method in your content provider that the user can execute.

# Creating a Search Interface

## In this document

1. [The Basics](#)
2. [Creating a Searchable Configuration](#)
3. [Creating a Searchable Activity](#)
  1. [Declaring a searchable activity](#)
  2. [Performing a search](#)
4. [Using the Search Dialog](#)
  1. [Invoking the search dialog](#)
  2. [The impact of the search dialog on your activity lifecycle](#)
  3. [Passing search context data](#)
5. [Using the Search Widget](#)
  1. [Configuring the search widget](#)
  2. [Other search widget features](#)
  3. [Using both the widget and the dialog](#)
6. [Adding Voice Search](#)
7. [Adding Search Suggestions](#)

## Key classes

1. [SearchManager](#)
2. [SearchView](#)

## Related samples

1. [Searchable Dictionary](#)
2. [SearchView in the Action Bar](#)
3. [SearchView filter mode](#)

## Downloads

1. [ActionBar Icon Pack](#)

When you're ready to add search functionality to your application, Android helps you implement the user interface with either a search dialog that appears at the top of the activity window or a search widget that you can insert in your layout. Both the search dialog and the widget can deliver the user's search query to a specific activity in your application. This way, the user can initiate a search from any activity where the search dialog or widget is available, and the system starts the appropriate activity to perform the search and present results.

Other features available for the search dialog and widget include:

- Voice search
- Search suggestions based on recent queries
- Search suggestions that match actual results in your application data

This guide shows you how to set up your application to provide a search interface that's assisted by the Android system to deliver search queries, using either the search dialog or the search widget.

# The Basics



**Figure 1.** Screenshot of an application's search dialog.

Before you begin, you should decide whether you'll implement your search interface using the search dialog or the search widget. Both provide the same search features, but in slightly different ways:

- The **search dialog** is a UI component that's controlled by the Android system. When activated by the user, the search dialog appears at the top of the activity, as shown in figure 1.

The Android system controls all events in the search dialog. When the user submits a query, the system delivers the query to the activity that you specify to handle searches. The dialog can also provide search suggestions while the user types.

- The **search widget** is an instance of [SearchView](#) that you can place anywhere in your layout. By default, the search widget behaves like a standard [EditText](#) widget and doesn't do anything, but you can configure it so that the Android system handles all input events, delivers queries to the appropriate activity, and provides search suggestions (just like the search dialog). However, the search widget is available only in Android 3.0 (API Level 11) and higher.

**Note:** If you want, you can handle all user input into the search widget yourself, using various callback methods and listeners. This document, however, focuses on how to integrate the search widget with the system for an assisted search implementation. If you want to handle all user input yourself, read the reference documentation for [SearchView](#) and its nested interfaces.

When the user executes a search from the search dialog or a search widget, the system creates an [Intent](#) and stores the user query in it. The system then starts the activity that you've declared to handle searches (the "searchable activity") and delivers it the intent. To set up your application for this kind of assisted search, you need the following:

- A searchable configuration

An XML file that configures some settings for the search dialog or widget. It includes settings for features such as voice search, search suggestion, and hint text for the search box.

- A searchable activity

The [Activity](#) that receives the search query, searches your data, and displays the search results.

- A search interface, provided by either:
  - The search dialog

By default, the search dialog is hidden, but appears at the top of the screen when you call [on-SearchRequested\(\)](#) (when the user presses your Search button).

- Or, a [SearchView](#) widget

Using the search widget allows you to put the search box anywhere in your activity. Instead of putting it in your activity layout, you should usually use [SearchView](#) as an [action view in the Action Bar](#).

The rest of this document shows you how to create the searchable configuration, searchable activity, and implement a search interface with either the search dialog or search widget.

## Creating a Searchable Configuration

The first thing you need is an XML file called the searchable configuration. It configures certain UI aspects of the search dialog or widget and defines how features such as suggestions and voice search behave. This file is traditionally named `searchable.xml` and must be saved in the `res/xml/` project directory.

**Note:** The system uses this file to instantiate a [SearchableInfo](#) object, but you cannot create this object yourself at runtime—you must declare the searchable configuration in XML.

The searchable configuration file must include the [`<searchable>`](#) element as the root node and specify one or more attributes. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_label"
    android:hint="@string/search_hint" >
</searchable>
```

The `android:label` attribute is the only required attribute. It points to a string resource, which should be the application name. This label isn't actually visible to the user until you enable search suggestions for Quick Search Box. At that point, this label is visible in the list of Searchable items in the system Settings.

Though it's not required, we recommend that you always include the `android:hint` attribute, which provides a hint string in the search box before users enters a query. The hint is important because it provides important clues to users about what they can search.

**Tip:** For consistency among other Android applications, you should format the string for `android:hint` as "Search <content-or-product>". For example, "Search songs and artists" or "Search YouTube".

The [`<searchable>`](#) element accepts several other attributes. However, you don't need most attributes until you add features such as [search suggestions](#) and [voice search](#). For detailed information about the searchable configuration file, see the [Searchable Configuration](#) reference document.

# Creating a Searchable Activity

A searchable activity is the [Activity](#) in your application that performs searches based on a query string and presents the search results.

When the user executes a search in the search dialog or widget, the system starts your searchable activity and delivers it the search query in an [Intent](#) with the [ACTION\\_SEARCH](#) action. Your searchable activity retrieves the query from the intent's [QUERY](#) extra, then searches your data and presents the results.

Because you may include the search dialog or widget in any other activity in your application, the system must know which activity is your searchable activity, so it can properly deliver the search query. So, you must first declare your searchable activity in the Android manifest file.

## Declaring a searchable activity

If you don't have one already, create an [Activity](#) that will perform searches and present results. You don't need to implement the search functionality yet—just create an activity that you can declare in the manifest. Inside the manifest's [activity](#) element:

1. Declare the activity to accept the [ACTION\\_SEARCH](#) intent, in an [<intent-filter>](#) element.
2. Specify the searchable configuration to use, in a [<meta-data>](#) element.

For example:

```
<application ... >
    <activity android:name=".SearchableActivity" >
        <intent-filter>
            <action android:name="android.intent.action.SEARCH" />
        </intent-filter>
        <meta-data android:name="android.app.searchable"
                  android:resource="@xml/searchable"/>
    </activity>
    ...
</application>
```

The [<meta-data>](#) element must include the `android:name` attribute with a value of "an-  
droid.app.searchable" and the `android:resource` attribute with a reference to the searchable  
configuration file (in this example, it refers to the `res/xml/searchable.xml` file).

**Note:** The [<intent-filter>](#) does not need a [<category>](#) with the `DEFAULT` value (which you usually see in [activity](#) elements), because the system delivers the [ACTION\\_SEARCH](#) intent explicitly to your searchable activity, using its component name.

## Performing a search

Once you have declared your searchable activity in the manifest, performing a search in your searchable activity involves three steps:

1. [Receiving the query](#)
2. [Searching your data](#)
3. [Presenting the results](#)

Traditionally, your search results should be presented in a [ListView](#), so you might want your searchable activity to extend [ListActivity](#). It includes a default layout with a single [ListView](#) and provides several convenience methods for working with the [ListView](#).

## Receiving the query

When a user executes a search from the search dialog or widget, the system starts your searchable activity and sends it a [ACTION\\_SEARCH](#) intent. This intent carries the search query in the [QUERY](#) string extra. You must check for this intent when the activity starts and extract the string. For example, here's how you can get the search query when your searchable activity starts:

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.search);  
  
    // Get the intent, verify the action and get the query  
    Intent intent = getIntent();  
    if (Intent.ACTION_SEARCH.equals(intent.getAction())) {  
        String query = intent.getStringExtra(SearchManager.QUERY);  
        doMySearch(query);  
    }  
}
```

The [QUERY](#) string is always included with the [ACTION\\_SEARCH](#) intent. In this example, the query is retrieved and passed to a local `doMySearch()` method where the actual search operation is done.

## Searching your data

The process of storing and searching your data is unique to your application. You can store and search your data in many ways, but this guide does not show you how to store your data and search it. Storing and searching your data is something you should carefully consider in terms of your needs and your data format. However, here are some tips you might be able to apply:

- If your data is stored in a SQLite database on the device, performing a full-text search (using FTS3, rather than a `LIKE` query) can provide a more robust search across text data and can produce results significantly faster. See [sqlite.org](#) for information about FTS3 and the [SQLiteDatabase](#) class for information about SQLite on Android. Also look at the [Searchable Dictionary](#) sample application to see a complete SQLite implementation that performs searches with FTS3.
- If your data is stored online, then the perceived search performance might be inhibited by the user's data connection. You might want to display a spinning progress wheel until your search returns. See [android.net](#) for a reference of network APIs and [Creating a Progress Dialog](#) for information about how to display a progress wheel.

## About Adapters

An [Adapter](#) binds each item from a set of data into a [View](#) object. When the [Adapter](#) is applied to a [ListView](#), each piece of data is inserted as an individual view into the list. [Adapter](#) is just an interface, so implementations such as [CursorAdapter](#) (for binding data from a [Cursor](#)) are needed. If none of the existing implementations work for your data, then you can implement your own from [BaseAdapter](#). Install the SDK Samples package for API Level 4 to see the original version of the Searchable Dictionary, which creates a custom adapter to read data from a file.

Regardless of where your data lives and how you search it, we recommend that you return search results to your searchable activity with an [Adapter](#). This way, you can easily present all the search results in a [ListView](#). If your data comes from a SQLite database query, you can apply your results to a [ListView](#) using a [CursorAdapter](#). If your data comes in some other type of format, then you can create an extension of [BaseAdapter](#).

## Presenting the results

As discussed above, the recommended UI for your search results is a [ListView](#), so you might want your searchable activity to extend [ListActivity](#). You can then call `setListAdapter()`, passing it an [Adapter](#) that is bound to your data. This injects all the search results into the activity [ListView](#).

For more help presenting your results in a list, see the [ListActivity](#) documentation.

Also see the [Searchable Dictionary](#) sample for a complete demonstration of how to search an SQLite database and use an [Adapter](#) to provide results in a [ListView](#).

## Using the Search Dialog

### Should I use the search dialog or the widget?

The answer depends mostly on whether you are developing for Android 3.0 (API Level 11 or higher), because the [SearchView](#) widget was introduced in Android 3.0. So, if you are developing your application for a version of Android lower than 3.0, the search widget is not an option and you should use the search dialog to implement your search interface.

If you *are* developing for Android 3.0 or higher, then the decision depends more on your needs. In most cases, we recommend that you use the search widget as an "action view" in the Action Bar. However, it might not be an option for you to put the search widget in the Action Bar for some reason (perhaps there's not enough space or you don't use the Action Bar). So, you might instead want to put the search widget somewhere in your activity layout. And if all else fails, you can still use the search dialog if you prefer to keep the search box hidden. In fact, you might want to offer both the dialog and the widget in some cases. For more information about the widget, skip to [Using the Search Widget](#).

The search dialog provides a floating search box at the top of the screen, with the application icon on the left. The search dialog can provide search suggestions as the user types and, when the user executes a search, the system sends the search query to a searchable activity that performs the search. However, if you are developing your application for devices running Android 3.0, you should consider using the search widget instead (see the side box).

The search dialog is always hidden by default, until the user activates it. Your application can activate the search dialog by calling `onSearchRequested()`. However, this method doesn't work until you enable the search dialog for the activity.

To enable the search dialog, you must indicate to the system which searchable activity should receive search queries from the search dialog, in order to perform searches. For example, in the previous section about [Creating a Searchable Activity](#), a searchable activity named `SearchableActivity` was created. If you want a separate activity, named `OtherActivity`, to show the search dialog and deliver searches to `SearchableActivity`, you must declare in the manifest that `SearchableActivity` is the searchable activity to use for the search dialog in `OtherActivity`.

To declare the searchable activity for an activity's search dialog, add a `<meta-data>` element inside the respective activity's `<activity>` element. The `<meta-data>` element must include the `android:value` attribute that specifies the searchable activity's class name and the `android:name` attribute with a value of `"android.app.default_searchable"`.

For example, here is the declaration for both a searchable activity, `SearchableActivity`, and another activity, `OtherActivity`, which uses `SearchableActivity` to perform searches executed from its search dialog:

```
<application ... >
    <!-- this is the searchable activity; it performs searches -->
    <activity android:name=".SearchableActivity" >
        <intent-filter>
            <action android:name="android.intent.action.SEARCH" />
        </intent-filter>
        <meta-data android:name="android.app.searchable"
                  android:resource="@xml/searchable"/>
    </activity>

    <!-- this activity enables the search dialog to initiate searches
         in the SearchableActivity -->
    <activity android:name=".OtherActivity" ... >
        <!-- enable the search dialog to send searches to SearchableActivity -->
        <meta-data android:name="android.app.default_searchable"
                  android:value=".SearchableActivity" />
    </activity>
    ...
</application>
```

Because the `OtherActivity` now includes a `<meta-data>` element to declare which searchable activity to use for searches, the activity has enabled the search dialog. While the user is in this activity, the `onSearchRequested()` method activates the search dialog. When the user executes the search, the system starts `SearchableActivity` and delivers it the `ACTION_SEARCH` intent.

**Note:** The searchable activity itself provides the search dialog by default, so you don't need to add this declaration to `SearchableActivity`.

If you want every activity in your application to provide the search dialog, insert the above `<meta-data>` element as a child of the `<application>` element, instead of each `<activity>`. This way, every activity inherits the value, provides the search dialog, and delivers searches to the same searchable activity. (If you have multiple searchable activities, you can override the default searchable activity by placing a different `<meta-data>` declaration inside individual activities.)

With the search dialog now enabled for your activities, your application is ready to perform searches.

## Invoking the search dialog

Although some devices provide a dedicated Search button, the behavior of the button may vary between devices and many devices do not provide a Search button at all. So when using the search dialog, you **must provide a search button in your UI** that activates the search dialog by calling `onSearchRequested()`.

For instance, you should add a Search button in your [Options Menu](#) or UI layout that calls [onSearchRequested\(\)](#). For consistency with the Android system and other apps, you should label your button with the Android Search icon that's available from the [Action Bar Icon Pack](#).

**Note:** If your app uses the [action bar](#), then you should not use the search dialog for your search interface. Instead, use the [search widget](#) as a collapsible view in the action bar.

You can also enable "type-to-search" functionality, which activates the search dialog when the user starts typing on the keyboard—the keystrokes are inserted into the search dialog. You can enable type-to-search in your activity by calling [setDefaultKeyMode\(DEFAULT\\_KEYS\\_SEARCH\\_LOCAL\)](#) during your activity's [onCreate\(\)](#) method.

## The impact of the search dialog on your activity lifecycle

The search dialog is a [Dialog](#) that floats at the top of the screen. It does not cause any change in the activity stack, so when the search dialog appears, no lifecycle methods (such as [onPause\(\)](#)) are called. Your activity just loses input focus, as input focus is given to the search dialog.

If you want to be notified when the search dialog is activated, override the [onSearchRequested\(\)](#) method. When the system calls this method, it is an indication that your activity has lost input focus to the search dialog, so you can do any work appropriate for the event (such as pause a game). Unless you are [passing search context data](#) (discussed below), you should end the method by calling the super class implementation. For example:

```
@Override  
public boolean onSearchRequested() {  
    pauseSomeStuff();  
    return super.onSearchRequested();  
}
```

If the user cancels search by pressing the *Back* button, the search dialog closes and the activity regains input focus. You can register to be notified when the search dialog is closed with [setOnDismissListener\(\)](#) and/or [setOnCancelListener\(\)](#). You should need to register only the [OnDismissListener](#), because it is called every time the search dialog closes. The [OnCancelListener](#) only pertains to events in which the user explicitly exited the search dialog, so it is not called when a search is executed (in which case, the search dialog naturally disappears).

If the current activity is not the searchable activity, then the normal activity lifecycle events are triggered once the user executes a search (the current activity receives [onPause\(\)](#) and so forth, as described in the [Activities](#) document). If, however, the current activity is the searchable activity, then one of two things happens:

- a. By default, the searchable activity receives the [ACTION\\_SEARCH](#) intent with a call to [onCreate\(\)](#) and a new instance of the activity is brought to the top of the activity stack. There are now two instances of your searchable activity in the activity stack (so pressing the *Back* button goes back to the previous instance of the searchable activity, rather than exiting the searchable activity).
- b. If you set `android:launchMode` to "`singleTop`", then the searchable activity receives the [ACTION\\_SEARCH](#) intent with a call to [onNewIntent\(Intent\)](#), passing the new [ACTION\\_SEARCH](#) intent here. For example, here's how you might handle this case, in which the searchable activity's launch mode is "`singleTop`":

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.search);  
    handleIntent(getIntent());
```

```

}

@Override
protected void onNewIntent(Intent intent) {
    setIntent(intent);
    handleIntent(intent);
}

private void handleIntent(Intent intent) {
    if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
        String query = intent.getStringExtra(SearchManager.QUERY);
        doMySearch(query);
    }
}

```

Compared to the example code in the section about [Performing a Search](#), all the code to handle the search intent is now in the `handleIntent()` method, so that both [onCreate\(\)](#) and [onNewIntent\(\)](#) can execute it.

When the system calls [onNewIntent\(Intent\)](#), the activity has not been restarted, so the [getIntent\(\)](#) method returns the same intent that was received with [onCreate\(\)](#). This is why you should call [setIntent\(Intent\)](#) inside [onNewIntent\(Intent\)](#) (so that the intent saved by the activity is updated in case you call [getIntent\(\)](#) in the future).

The second scenario using "singleTop" launch mode is usually ideal, because chances are good that once a search is done, the user will perform additional searches and it's a bad experience if your application creates multiple instances of the searchable activity. So, we recommend that you set your searchable activity to "singleTop" launch mode in the application manifest. For example:

```

<activity android:name=".SearchableActivity"
          android:launchMode="singleTop" >
    <intent-filter>
        <action android:name="android.intent.action.SEARCH" />
    </intent-filter>
    <meta-data android:name="android.app.searchable"
              android:resource="@xml/searchable"/>
</activity>

```

## Passing search context data

In some cases, you can make necessary refinements to the search query inside the searchable activity, for every search made. However, if you want to refine your search criteria based on the activity from which the user is performing a search, you can provide additional data in the intent that the system sends to your searchable activity. You can pass the additional data in the [APP DATA Bundle](#), which is included in the [ACTION\\_SEARCH](#) intent.

To pass this kind of data to your searchable activity, override the [onSearchRequested\(\)](#) method for the activity from which the user can perform a search, create a [Bundle](#) with the additional data, and call [startSearch\(\)](#) to activate the search dialog. For example:

```

@Override
public boolean onSearchRequested() {
    Bundle appData = new Bundle();

```

```
        appData.putBoolean(SearchableActivity.JARGON, true);
        startSearch(null, false, appData, false);
        return true;
    }
```

Returning "true" indicates that you have successfully handled this callback event and called [startSearch\(\)](#) to activate the search dialog. Once the user submits a query, it's delivered to your searchable activity along with the data you've added. You can extract the extra data from the [APP\\_DATA Bundle](#) to refine the search. For example:

```
Bundle appData = getIntent().getBundleExtra(SearchManager.APP_DATA);
if (appData != null) {
    boolean jargon = appData.getBoolean(SearchableActivity.JARGON);
}
```

**Caution:** Never call the [startSearch\(\)](#) method from outside the [onSearchRequested\(\)](#) callback method. To activate the search dialog in your activity, always call [onSearchRequested\(\)](#). Otherwise, [onSearchRequested\(\)](#) is not called and customizations (such as the addition of appData in the above example) are missed.

## Using the Search Widget



**Figure 2.** The [SearchView](#) widget as an "action view" in the Action Bar.

The [SearchView](#) widget is available in Android 3.0 and higher. If you're developing your application for Android 3.0 and have decided to use the search widget, we recommend that you insert the search widget as an [action view in the Action Bar](#), instead of using the search dialog (and instead of placing the search widget in your activity layout). For example, figure 2 shows the search widget in the Action Bar.

The search widget provides the same functionality as the search dialog. It starts the appropriate activity when the user executes a search, and it can provide search suggestions and perform voice search.

**Note:** When you use the search widget as an action view, you still might need to support using the search dialog, for cases in which the search widget does not fit in the Action Bar. See the following section about [Using both the widget and the dialog](#).

## Configuring the search widget

After you've created a [searchable configuration](#) and a [searchable activity](#), as discussed above, you need to enable assisted search for each [SearchView](#). You can do so by calling [setSearchableInfo\(\)](#) and passing it the [SearchableInfo](#) object that represents your searchable configuration.

You can get a reference to the [SearchableInfo](#) by calling [getSearchableInfo\(\)](#) on [SearchManager](#).

For example, if you're using a [SearchView](#) as an action view in the [Action Bar](#), you should enable the widget during the [onCreateOptionsMenu\(\)](#) callback:

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the options menu from XML
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);

    // Get the SearchView and set the searchable configuration
    SearchManager searchManager = (SearchManager) getSystemService(Context.SEARCH_SERVICE);
    SearchView searchView = (SearchView) menu.findItem(R.id.menu_search).getActionView();
    // Assumes current activity is the searchable activity
    searchView.setSearchableInfo(searchManager.getSearchableInfo(getApplicationContext()));
    searchView.setIconifiedByDefault(false); // Do not iconify the widget; expand it by default

    return true;
}

```

That's all you need. The search widget is now configured and the system will deliver search queries to your searchable activity. You can also enable [search suggestions](#) for the search widget.

**Note:** If you want to handle all user input yourself, you can do so with some callback methods and event listeners. For more information, see the reference documentation for [SearchView](#) and its nested interfaces for the appropriate event listeners.

For more information about action views in the Action Bar, read the [Action Bar](#) developer guide (which includes sample code for adding a search widget as an action view).

## Other search widget features

The [SearchView](#) widget allows for a few additional features you might want:

### A submit button

By default, there's no button to submit a search query, so the user must press the "Return" key on the keyboard to initiate a search. You can add a "submit" button by calling [setSubmitButtonEnabled\(true\)](#).

### Query refinement for search suggestions

When you've enabled search suggestions, you usually expect users to simply select a suggestion, but they might also want to refine the suggested search query. You can add a button alongside each suggestion that inserts the suggestion in the search box for refinement by the user, by calling [setQueryRefinementEnabled\(true\)](#).

### The ability to toggle the search box visibility

By default, the search widget is "iconified," meaning that it is represented only by a search icon (a magnifying glass), and expands to show the search box when the user touches it. As shown above, you can show the search box by default, by calling [setIconifiedByDefault\(false\)](#). You can also toggle the search widget appearance by calling [setIconified\(\)](#).

There are several other APIs in the [SearchView](#) class that allow you to customize the search widget. However, most of them are used only when you handle all user input yourself, instead of using the Android system to deliver search queries and display search suggestions.

## Using both the widget and the dialog

If you insert the search widget in the Action Bar as an [action view](#), and you enable it to appear in the Action Bar "if there is room" (by setting `android:showAsAction="ifRoom"`), then there is a chance that the search widget will not appear as an action view, but the menu item will appear in the overflow menu. For example, when your application runs on a smaller screen, there might not be enough room in the Action Bar to display the search widget along with other action items or navigation elements, so the menu item will instead appear in the overflow menu. When placed in the overflow menu, the item works like an ordinary menu item and does not display the action view (the search widget).

To handle this situation, the menu item to which you've attached the search widget should activate the search dialog when the user selects it from the overflow menu. In order for it to do so, you must implement [onOptionsItemSelected\(\)](#) to handle the "Search" menu item and open the search dialog by calling [onSearchRequested\(\)](#).

For more information about how items in the Action Bar work and how to handle this situation, see the [ActionBar](#) developer guide.

Also see the [Searchable Dictionary](#) for an example implementation using both the dialog and the widget.

## Adding Voice Search

You can add voice search functionality to your search dialog or widget by adding the `android:voiceSearchMode` attribute to your searchable configuration. This adds a voice search button that launches a voice prompt. When the user has finished speaking, the transcribed search query is sent to your searchable activity.

For example:

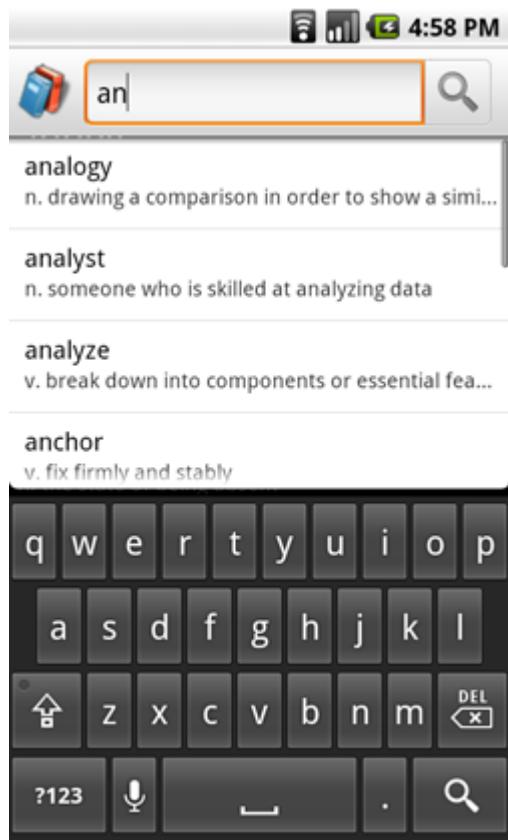
```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/search_label"
    android:hint="@string/search_hint"
    android:voiceSearchMode="showVoiceSearchButton|launchRecognizer" >
</searchable>
```

The value `showVoiceSearchButton` is required to enable voice search, while the second value, `launchRecognizer`, specifies that the voice search button should launch a recognizer that returns the transcribed text to the searchable activity.

You can provide additional attributes to specify the voice search behavior, such as the language to be expected and the maximum number of results to return. See the [Searchable Configuration](#) reference for more information about the available attributes.

**Note:** Carefully consider whether voice search is appropriate for your application. All searches performed with the voice search button are immediately sent to your searchable activity without a chance for the user to review the transcribed query. Sufficiently test the voice recognition and ensure that it understands the types of queries that the user might submit inside your application.

# Adding Search Suggestions



**Figure 3.** Screenshot of a search dialog with custom search suggestions.

Both the search dialog and the search widget can provide search suggestions as the user types, with assistance from the Android system. The system manages the list of suggestions and handles the event when the user selects a suggestion.

You can provide two kinds of search suggestions:

## Recent query search suggestions

These suggestions are simply words that the user previously used as search queries in your application.

See [Adding Recent Query Suggestions](#).

## Custom search suggestions

These are search suggestions that you provide from your own data source, to help users immediately select the correct spelling or item they are searching for. Figure 3 shows an example of custom suggestions for a dictionary application—the user can select a suggestion to instantly go to the definition.

See [Adding Custom Suggestions](#)

# Adding Recent Query Suggestions

## In this document

1. [The Basics](#)
2. [Creating a Content Provider](#)
3. [Modifying the Searchable Configuration](#)
4. [Saving Queries](#)
5. [Clearing the Suggestion Data](#)

## Key classes

1. [SearchRecentSuggestions](#)
2. [SearchRecentSuggestionsProvider](#)

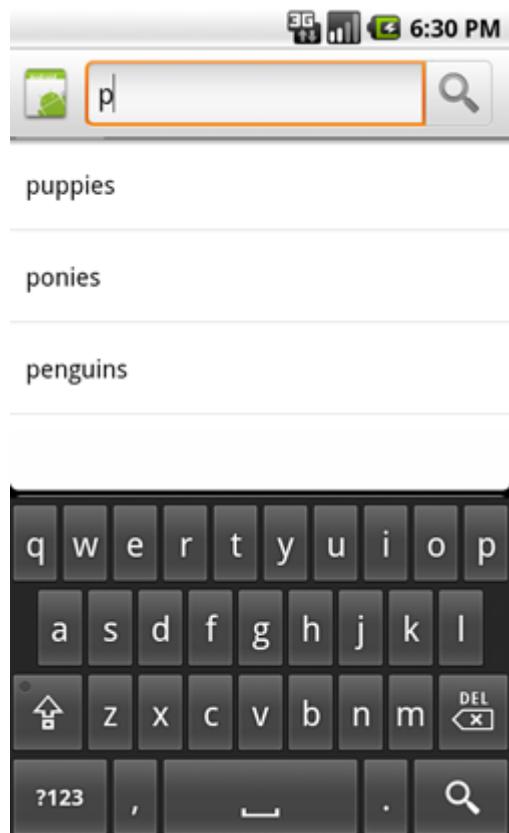
## See also

1. [Searchable Configuration](#)

When using the Android search dialog or search widget, you can provide search suggestions based on recent search queries. For example, if a user previously searched for "puppies," then that query appears as a suggestion once he or she begins typing the same query. Figure 1 shows an example of a search dialog with recent query suggestions.

Before you begin, you need to implement the search dialog or a search widget for basic searches in your application. If you haven't, see [Creating a Search Interface](#).

## The Basics



## Figure 1.

Screenshot of a search dialog with recent query suggestions.

Recent query suggestions are simply saved searches. When the user selects one of the suggestions, your searchable activity receives a [ACTION\\_SEARCH](#) intent with the suggestion as the search query, which your searchable activity already handles (as described in [Creating a Search Interface](#)).

To provide recent queries suggestions, you need to:

- Implement a searchable activity, as described in [Creating a Search Interface](#).
- Create a content provider that extends [SearchRecentSuggestionsProvider](#) and declare it in your application manifest.
- Modify the searchable configuration with information about the content provider that provides search suggestions.
- Save queries to your content provider each time a search is executed.

Just as the Android system displays the search dialog, it also displays the search suggestions below the dialog or search widget. All you need to do is provide a source from which the system can retrieve suggestions.

When the system identifies that your activity is searchable and provides search suggestions, the following procedure takes place as soon as the user begins typing a query:

1. The system takes the search query text (whatever has been typed so far) and performs a query to the content provider that contains your suggestions.
2. Your content provider returns a [Cursor](#) that points to all suggestions that match the search query text.
3. The system displays the list of suggestions provided by the Cursor.

Once the recent query suggestions are displayed, the following might happen:

- If the user types another key, or changes the query in any way, the aforementioned steps are repeated and the suggestion list is updated.
- If the user executes the search, the suggestions are ignored and the search is delivered to your searchable activity using the normal [ACTION\\_SEARCH](#) intent.
- If the user selects a suggestion, an [ACTION\\_SEARCH](#) intent is delivered to your searchable activity using the suggested text as the query.

The [SearchRecentSuggestionsProvider](#) class that you extend for your content provider automatically does the work described above, so there's actually very little code to write.

## Creating a Content Provider

The content provider that you need for recent query suggestions must be an implementation of [SearchRecentSuggestionsProvider](#). This class does practically everything for you. All you have to do is write a class constructor that executes one line of code.

For example, here's a complete implementation of a content provider for recent query suggestions:

```
public class MySuggestionProvider extends SearchRecentSuggestionsProvider {  
    public final static String AUTHORITY = "com.example.MySuggestionProvider";  
    public final static int MODE = DATABASE_MODE_QUERIES;  
  
    public MySuggestionProvider() {  
        setupSuggestions(AUTHORITY, MODE);  
    }  
}
```

The call to `setupSuggestions()` passes the name of the search authority and a database mode. The search authority can be any unique string, but the best practice is to use a fully qualified name for your content provider (package name followed by the provider's class name; for example, "com.example.MySuggestionProvider"). The database mode must include `DATABASE_MODE_QUERIES` and can optionally include `DATABASE_MODE_2LINES`, which adds another column to the suggestions table that allows you to provide a second line of text with each suggestion. For example, if you want to provide two lines in each suggestion:

```
public final static int MODE = DATABASE_MODE_QUERIES | DATABASE_MODE_2LINES;
```

Now declare the content provider in your application manifest with the same authority string used in your `SearchRecentSuggestionsProvider` class (and in the searchable configuration). For example:

```
<application>
    <provider android:name=".MySuggestionProvider"
              android:authorities="com.example.MySuggestionProvider" />
    ...
</application>
```

## Modifying the Searchable Configuration

To configure the system to use your suggestions provider, you need to add the `android:searchSuggestAuthority` and `android:searchSuggestSelection` attributes to the `<searchable>` element in your searchable configuration file. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_label"
    android:hint="@string/search_hint"
    android:searchSuggestAuthority="com.example.MySuggestionProvider"
    android:searchSuggestSelection=" ?" >
</searchable>
```

The value for `android:searchSuggestAuthority` should be a fully qualified name for your content provider that exactly matches the authority used in the content provider (the AUTHORITY string in the above example).

The value for `android:searchSuggestSelection` must be a single question mark, preceded by a space (" ?"), which is simply a placeholder for the SQLite selection argument (which is automatically replaced by the query text entered by the user).

## Saving Queries

To populate your collection of recent queries, add each query received by your searchable activity to your `SearchRecentSuggestionsProvider`. To do this, create an instance of `SearchRecentSuggestions` and call `saveRecentQuery()` each time your searchable activity receives a query. For example, here's how you can save the query during your activity's `onCreate()` method:

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

```
Intent intent = getIntent();

if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
    String query = intent.getStringExtra(SearchManager.QUERY);
    SearchRecentSuggestions suggestions = new SearchRecentSuggestions(this,
        MySuggestionProvider.AUTHORITY, MySuggestionProvider.MODE);
    suggestions.saveRecentQuery(query, null);
}
}
```

The [SearchRecentSuggestionsProvider](#) constructor requires the same authority and database mode declared by your content provider.

The [saveRecentQuery\(\)](#) method takes the search query string as the first parameter and, optionally, a second string to include as the second line of the suggestion (or null). The second parameter is only used if you've enabled two-line mode for the search suggestions with [DATABASE\\_MODE\\_2LINES](#). If you have enabled two-line mode, then the query text is also matched against this second line when the system looks for matching suggestions.

## Clearing the Suggestion Data

To protect the user's privacy, you should always provide a way for the user to clear the recent query suggestions. To clear the query history, call [clearHistory\(\)](#). For example:

```
SearchRecentSuggestions suggestions = new SearchRecentSuggestions(this,
    HelloSuggestionProvider.AUTHORITY, HelloSuggestionProvider.MODE);
suggestions.clearHistory();
```

Execute this from your choice of a "Clear Search History" menu item, preference item, or button. You should also provide a confirmation dialog to verify that the user wants to delete their search history.

# Adding Custom Suggestions

## In this document

1. [The Basics](#)
2. [Modifying the Searchable Configuration](#)
3. [Creating a Content Provider](#)
  1. [Handling a suggestion query](#)
  2. [Building a suggestion table](#)
4. [Declaring an Intent for Suggestions](#)
  1. [Declaring the intent action](#)
  2. [Declaring the intent data](#)
5. [Handling the Intent](#)
6. [Rewriting the Query Text](#)
7. [Exposing Search Suggestions to Quick Search Box](#)

## Key classes

1. [SearchManager](#)
2. [SearchRecentSuggestionsProvider](#)
3. [ContentProvider](#)

## Related samples

1. [Searchable Dictionary](#)

## See also

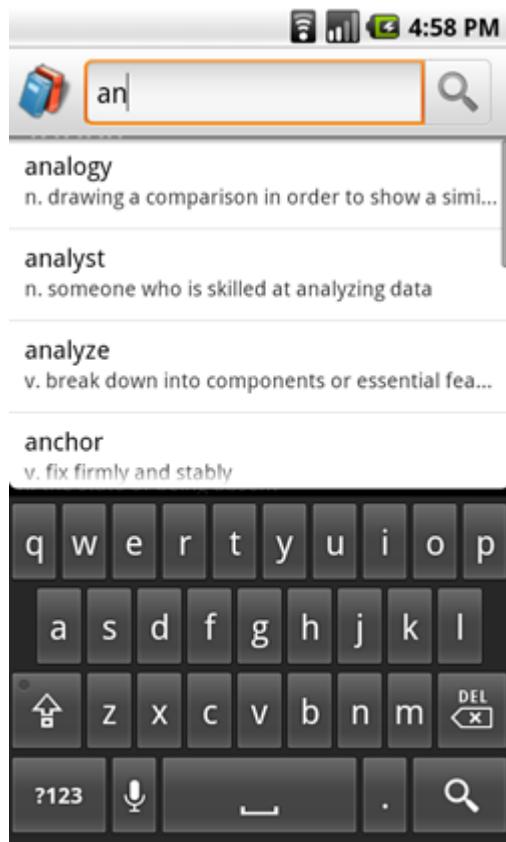
1. [Searchable Configuration](#)
2. [Content Providers](#)

When using the Android search dialog or search widget, you can provide custom search suggestions that are created from data in your application. For example, if your application is a word dictionary, you can suggest words from the dictionary that match the text entered so far. These are the most valuable suggestions, because you can effectively predict what the user wants and provide instant access to it. Figure 1 shows an example of a search dialog with custom suggestions.

Once you provide custom suggestions, you can also make them available to the system-wide Quick Search Box, providing access to your content from outside your application.

Before you begin with this guide to add custom suggestions, you need to have implemented the Android search dialog or a search widget for searches in your application. If you haven't, see [Creating a Search Interface](#).

# The Basics



**Figure 1.** Screenshot of a search dialog with custom search suggestions.

When the user selects a custom suggestion, the Android system sends an [Intent](#) to your searchable activity. Whereas a normal search query sends an intent with the [ACTION\\_SEARCH](#) action, you can instead define your custom suggestions to use [ACTION\\_VIEW](#) (or any other intent action), and also include data that's relevant to the selected suggestion. Continuing the dictionary example, when the user selects a suggestion, your application can immediately open the definition for that word, instead of searching the dictionary for matches.

To provide custom suggestions, do the following:

- Implement a basic searchable activity, as described in [Creating a Search Interface](#).
- Modify the searchable configuration with information about the content provider that provides custom suggestions.
- Build a table (such as in an [SQLiteDatabase](#)) for your suggestions and format the table with required columns.
- Create a [Content Provider](#) that has access to your suggestions table and declare the provider in your manifest.
- Declare the type of [Intent](#) to be sent when the user selects a suggestion (including a custom action and custom data).

Just as the Android system displays the search dialog, it also displays your search suggestions. All you need is a content provider from which the system can retrieve your suggestions. If you're not familiar with creating content providers, read the [Content Providers](#) developer guide before you continue.

When the system identifies that your activity is searchable and provides search suggestions, the following procedure takes place when the user types a query:

1. The system takes the search query text (whatever has been typed so far) and performs a query to your content provider that manages your suggestions.

2. Your content provider returns a [Cursor](#) that points to all suggestions that are relevant to the search query text.
3. The system displays the list of suggestions provided by the Cursor.

Once the custom suggestions are displayed, the following might happen:

- If the user types another key, or changes the query in any way, the above steps are repeated and the suggestion list is updated as appropriate.
- If the user executes the search, the suggestions are ignored and the search is delivered to your searchable activity using the normal [ACTION\\_SEARCH](#) intent.
- If the user selects a suggestion, an intent is sent to your searchable activity, carrying a custom action and custom data so that your application can open the suggested content.

## Modifying the searchable configuration

To add support for custom suggestions, add the `android:searchSuggestAuthority` attribute to the `<searchable>` element in your searchable configuration file. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_label"
    android:hint="@string/search_hint"
    android:searchSuggestAuthority="com.example.MyCustomSuggestionProvider">
</searchable>
```

You might need some additional attributes, depending on the type of intent you attach to each suggestion and how you want to format queries to your content provider. The other optional attributes are discussed in the following sections.

## Creating a Content Provider

Creating a content provider for custom suggestions requires previous knowledge about content providers that's covered in the [Content Provider](#) developer guide. For the most part, a content provider for custom suggestions is the same as any other content provider. However, for each suggestion you provide, the respective row in the [Cursor](#) must include specific columns that the system understands and uses to format the suggestions.

When the user starts typing into the search dialog or search widget, the system queries your content provider for suggestions by calling [query\(\)](#) each time a letter is typed. In your implementation of [query\(\)](#), your content provider must search your suggestion data and return a [Cursor](#) that points to the rows you have determined to be good suggestions.

Details about creating a content provider for custom suggestions are discussed in the following two sections:

### [Handling the suggestion query](#)

How the system sends requests to your content provider and how to handle them

### [Building a suggestion table](#)

How to define the columns that the system expects in the [Cursor](#) returned with each query

## Handling the suggestion query

When the system requests suggestions from your content provider, it calls your content provider's [query\(\)](#) method. You must implement this method to search your suggestion data and return a [Cursor](#) pointing to the suggestions you deem relevant.

Here's a summary of the parameters that the system passes to your [query\(\)](#) method (listed in order):

### **uri**

Always a content [Uri](#), formatted as:

`content://your.authority/optional.suggest.path/SUGGEST\_URI\_PATH\_QUERY`

The default behavior is for system to pass this URI and append it with the query text. For example:

`content://your.authority/optional.suggest.path/SUGGEST\_URI\_PATH\_QUERY/puppies`

The query text on the end is encoded using URI encoding rules, so you might need to decode it before performing a search.

The `optional.suggest.path` portion is only included in the URI if you have set such a path in your searchable configuration file with the `android:searchSuggestPath` attribute. This is only needed if you use the same content provider for multiple searchable activities, in which case, you need to disambiguate the source of the suggestion query.

**Note:** [SUGGEST\\_URI\\_PATH\\_QUERY](#) is not the literal string provided in the URI, but a constant that you should use if you need to refer to this path.

### **projection**

Always null

### **selection**

The value provided in the `android:searchSuggestSelection` attribute of your searchable configuration file, or null if you have not declared the `android:searchSuggestSelection` attribute.

More about using this to [get the query](#) below.

### **selectionArgs**

Contains the search query as the first (and only) element of the array if you have declared the `android:searchSuggestSelection` attribute in your searchable configuration. If you have not declared `android:searchSuggestSelection`, then this parameter is null. More about using this to [get the query](#) below.

### **sortOrder**

Always null

The system can send you the search query text in two ways. The default manner is for the query text to be included as the last path of the content URI passed in the `uri` parameter. However, if you include a selection value in your searchable configuration's `android:searchSuggestSelection` attribute, then the query text is instead passed as the first element of the `selectionArgs` string array. Both options are summarized next.

## Get the query in the Uri

By default, the query is appended as the last segment of the `uri` parameter (a [Uri](#) object). To retrieve the query text in this case, simply use [getLastPathSegment\(\)](#). For example:

```
String query = uri.getLastPathSegment().toLowerCase();
```

This returns the last segment of the [Uri](#), which is the query text entered by the user.

## Get the query in the selection arguments

Instead of using the URI, you might decide it makes more sense for your [query\(\)](#) method to receive everything it needs to perform the look-up and you want the `selection` and `selectionArgs` parameters to carry the appropriate values. In such a case, add the `android:searchSuggestSelection` attribute to your searchable configuration with your SQLite selection string. In the selection string, include a question mark ("?") as a placeholder for the actual search query. The system calls [query\(\)](#) with the selection string as the `selection` parameter and the search query as the first element in the `selectionArgs` array.

For example, here's how you might form the `android:searchSuggestSelection` attribute to create a full-text search statement:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_label"
    android:hint="@string/search_hint"
    android:searchSuggestAuthority="com.example.MyCustomSuggestionProvider"
    android:searchSuggestIntentAction="android.intent.action.VIEW"
    android:searchSuggestSelection="word MATCH ?">
</searchable>
```

With this configuration, your [query\(\)](#) method delivers the `selection` parameter as "word MATCH ?" and the `selectionArgs` parameter as the search query. When you pass these to an SQLite [query\(\)](#) method, as their respective arguments, they are synthesized together (the question mark is replaced with the query text). If you chose to receive suggestion queries this way and need to add wildcards to the query text, append (and/or prefix) them to the `selectionArgs` parameter, because this value is wrapped in quotes and inserted in place of the question mark.

Another new attribute in the example above is `android:searchSuggestIntentAction`, which defines the intent action sent with each intent when the user selects a suggestion. It is discussed further in the section about [Declaring an Intent for Suggestions](#).

**Tip:** If you don't want to define a selection clause in the `android:searchSuggestSelection` attribute, but would still like to receive the query text in the `selectionArgs` parameter, simply provide a non-null value for the `android:searchSuggestSelection` attribute. This triggers the query to be passed in `selectionArgs` and you can ignore the `selection` parameter. In this way, you can instead define the actual selection clause at a lower level so that your content provider doesn't have to handle it.

## Building a suggestion table

## Creating a Cursor without a table

If your search suggestions are not stored in a table format (such as an SQLite table) using the columns required by the system, then you can search your suggestion data for matches and then format them into the necessary table on each request. To do so, create a [MatrixCursor](#) using the required column names and then add a row for each suggestion using [addRow\(Object\[\]\)](#). Return the final product from your Content Provider's [query\(\)](#) method.

When you return suggestions to the system with a [Cursor](#), the system expects specific columns in each row. So, regardless of whether you decide to store your suggestion data in an SQLite database on the device, a database on a web server, or another format on the device or web, you must format the suggestions as rows in a table and present them with a [Cursor](#). The system understands several columns, but only two are required:

#### ID

A unique integer row ID for each suggestion. The system requires this in order to present suggestions in a [ListView](#).

#### SUGGEST COLUMN TEXT 1

The string that is presented as a suggestion.

The following columns are all optional (and most are discussed further in the following sections):

#### SUGGEST COLUMN TEXT 2

A string. If your Cursor includes this column, then all suggestions are provided in a two-line format. The string in this column is displayed as a second, smaller line of text below the primary suggestion text. It can be null or empty to indicate no secondary text.

#### SUGGEST COLUMN ICON 1

A drawable resource, content, or file URI string. If your Cursor includes this column, then all suggestions are provided in an icon-plus-text format with the drawable icon on the left side. This can be null or zero to indicate no icon in this row.

#### SUGGEST COLUMN ICON 2

A drawable resource, content, or file URI string. If your Cursor includes this column, then all suggestions are provided in an icon-plus-text format with the icon on the right side. This can be null or zero to indicate no icon in this row.

#### SUGGEST COLUMN INTENT ACTION

An intent action string. If this column exists and contains a value at the given row, the action defined here is used when forming the suggestion's intent. If the element is not provided, the action is taken from the `android:searchSuggestIntentAction` field in your searchable configuration. If your action is the same for all suggestions, it is more efficient to specify the action using `android:searchSuggestIntentAction` and omit this column.

#### SUGGEST COLUMN INTENT DATA

A data URI string. If this column exists and contains a value at the given row, this is the data that is used when forming the suggestion's intent. If the element is not provided, the data is taken from the `android:searchSuggestIntentData` field in your searchable configuration. If neither source is provided, the intent's data field is null. If your data is the same for all suggestions, or can be described using a constant part and a specific ID, it is more efficient to specify it using `android:searchSuggestIntentData` and omit this column.

#### SUGGEST COLUMN INTENT DATA ID

A URI path string. If this column exists and contains a value at the given row, then "/" and this value is appended to the data field in the intent. This should only be used if the data field specified by the `android:searchSuggestIntentData` attribute in the searchable configuration has already been set to an appropriate base string.

#### SUGGEST COLUMN INTENT EXTRA DATA

Arbitrary data. If this column exists and contains a value at a given row, this is the *extra* data used when forming the suggestion's intent. If not provided, the intent's extra data field is null. This column allows suggestions to provide additional data that is included as an extra in the intent's [EXTRA DATA KEY](#) key.

### SUGGEST COLUMN QUERY

If this column exists and this element exists at the given row, this is the data that is used when forming the suggestion's query, included as an extra in the intent's [QUERY](#) key. Required if suggestion's action is [ACTION\\_SEARCH](#), optional otherwise.

### SUGGEST COLUMN SHORTCUT ID

Only used when providing suggestions for Quick Search Box. This column indicates whether a search suggestion should be stored as a shortcut and whether it should be validated. Shortcuts are usually formed when the user clicks a suggestion from Quick Search Box. If missing, the result is stored as a shortcut and never refreshed. If set to [SUGGEST\\_NEVER\\_MAKE\\_SHORTCUT](#), the result is not stored as a shortcut. Otherwise, the shortcut ID is used to check back for an up to date suggestion using [SUGGEST\\_URI\\_PATH\\_SHORTCUT](#).

### SUGGEST COLUMN SPINNER WHILE REFRESHING

Only used when providing suggestions for Quick Search Box. This column specifies that a spinner should be shown instead of an icon from [SUGGEST\\_COLUMN\\_ICON\\_2](#) while the shortcut of this suggestion is being refreshed in Quick Search Box.

Some of these columns are discussed more in the following sections.

## Declaring an Intent for Suggestions

When the user selects a suggestion from the list that appears below the search dialog or widget, the system sends a custom [Intent](#) to your searchable activity. You must define the action and data for the intent.

### Declaring the intent action

The most common intent action for a custom suggestion is [ACTION\\_VIEW](#), which is appropriate when you want to open something, like the definition for a word, a person's contact information, or a web page. However, the intent action can be any other action and can even be different for each suggestion.

Depending on whether you want all suggestions to use the same intent action, you can define the action in two ways:

- a. Use the `android:searchSuggestIntentAction` attribute of your searchable configuration file to define the action for all suggestions.

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_label"
    android:hint="@string/search_hint"
    android:searchSuggestAuthority="com.example.MyCustomSuggestionProvider"
    android:searchSuggestIntentAction="android.Intent.action.VIEW" >
</searchable>
```

- b. Use the [SUGGEST\\_COLUMN\\_INTENT\\_ACTION](#) column to define the action for individual suggestions.

Add the [SUGGEST\\_COLUMN\\_INTENT\\_ACTION](#) column to your suggestions table and, for each suggestion, place in it the action to use (such as "android.Intent.action.VIEW").

You can also combine these two techniques. For instance, you can include the `android:searchSuggestIntentAction` attribute with an action to be used with all suggestions by default, then override this action for some suggestions by declaring a different action in the [SUGGEST\\_COLUMN\\_INTENT\\_ACTION](#) column. If you do not include a value in the [SUGGEST\\_COLUMN\\_INTENT\\_ACTION](#) column, then the intent provided in the `android:searchSuggestIntentAction` attribute is used.

**Note:** If you do not include the `android:searchSuggestIntentAction` attribute in your searchable configuration, then you *must* include a value in the [SUGGEST\\_COLUMN\\_INTENT\\_ACTION](#) column for every suggestion, or the intent will fail.

## Declaring intent data

When the user selects a suggestion, your searchable activity receives the intent with the action you've defined (as discussed in the previous section), but the intent must also carry data in order for your activity to identify which suggestion was selected. Specifically, the data should be something unique for each suggestion, such as the row ID for the suggestion in your SQLite table. When the intent is received, you can retrieve the attached data with [getData\(\)](#) or [getDataString\(\)](#).

You can define the data included with the intent in two ways:

- a. Define the data for each suggestion inside the [SUGGEST\\_COLUMN\\_INTENT\\_DATA](#) column of your suggestions table.

Provide all necessary data information for each intent in the suggestions table by including the [SUGGEST\\_COLUMN\\_INTENT\\_DATA](#) column and then populating it with unique data for each row. The data from this column is attached to the intent exactly as you define it in this column. You can then retrieve it with with [getData\(\)](#) or [getDataString\(\)](#).

**Tip:** It's usually easiest to use the table's row ID as the Intent data, because it's always unique. And the easiest way to do that is by using the [SUGGEST\\_COLUMN\\_INTENT\\_DATA](#) column name as an alias for the row ID column. See the [Searchable Dictionary sample app](#) for an example in which [SQLiteQueryBuilder](#) creates a projection map of column names to aliases.

- b. Fragment a data URI into two pieces: the portion common to all suggestions and the portion unique to each suggestion. Place these parts into the `android:searchSuggestIntentData` attribute of the searchable configuration and the [SUGGEST\\_COLUMN\\_INTENT\\_DATA\\_ID](#) column of your suggestions table, respectively.

Declare the piece of the URI that is common to all suggestions in the `android:searchSuggestIntentData` attribute of your searchable configuration. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_label"
    android:hint="@string/search_hint"
    android:searchSuggestAuthority="com.example.MyCustomSuggestionProvider"
    android:searchSuggestIntentAction="android.intent.action.VIEW"
    android:searchSuggestIntentData="content://com.example/datatable" >
</searchable>
```

Then include the final path for each suggestion (the unique part) in the [SUGGEST\\_COLUMN\\_INTENT\\_DATA\\_ID](#) column of your suggestions table. When the user selects a sug-

gestion, the system takes the string from `android:searchSuggestIntentData`, appends a slash ("/") and then adds the respective value from the `SUGGEST_COLUMN_INTENT_DATA_ID` column to form a complete content URI. You can then retrieve the `Uri` with `getData()`.

## Add more data

If you need to express even more information with your intent, you can add another table column, `SUGGEST_COLUMN_INTENT_EXTRA_DATA`, which can store additional information about the suggestion. The data saved in this column is placed in `EXTRA_DATA_KEY` of the intent's extra Bundle.

## Handling the Intent

Now that you provide custom search suggestions with custom intents, you need your searchable activity to handle these intents when the user selects a suggestion. This is in addition to handling the `ACTION_SEARCH` intent, which your searchable activity already does. Here's an example of how you can handle the intents during your activity `onCreate()` callback:

```
Intent intent = getIntent();
if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
    // Handle the normal search query case
    String query = intent.getStringExtra(SearchManager.QUERY);
    doSearch(query);
} else if (Intent.ACTION_VIEW.equals(intent.getAction())) {
    // Handle a suggestions click (because the suggestions all use ACTION_VIEW)
    Uri data = intent.getData();
    showResult(data);
}
```

In this example, the intent action is `ACTION_VIEW` and the data carries a complete URI pointing to the suggested item, as synthesized by the `android:searchSuggestIntentData` string and `SUGGEST_COLUMN_INTENT_DATA_ID` column. The URI is then passed to the local `showResult()` method that queries the content provider for the item specified by the URI.

**Note:** You do *not* need to add an intent filter to your Android manifest file for the intent action you defined with the `android:searchSuggestIntentAction` attribute or `SUGGEST_COLUMN_INTENT_ACTION` column. The system opens your searchable activity by name to deliver the suggestion's intent, so the activity does not need to declare the accepted action.

## Rewriting the query text

If the user navigates through the suggestions list using the directional controls (such as with a trackball or d-pad), the query text does not update, by default. However, you can temporarily rewrite the user's query text as it appears in the text box with a query that matches the suggestion currently in focus. This enables the user to see what query is being suggested (if appropriate) and then select the search box and edit the query before dispatching it as a search.

You can rewrite the query text in the following ways:

- a. Add the `android:searchMode` attribute to your searchable configuration with the "queryRewriteFromText" value. In this case, the content from the suggestion's `SUGGEST_COLUMN_TEXT_1` column is used to rewrite the query text.

- b. Add the `android:searchMode` attribute to your searchable configuration with the "queryRewriteFromData" value. In this case, the content from the suggestion's [SUGGEST\\_COLUMN\\_INTENT\\_DATA](#) column is used to rewrite the query text. This should only be used with URI's or other data formats that are intended to be user-visible, such as HTTP URLs. Internal URI schemes should not be used to rewrite the query in this way.
- c. Provide a unique query text string in the [SUGGEST\\_COLUMN\\_QUERY](#) column of your suggestions table. If this column is present and contains a value for the current suggestion, it is used to rewrite the query text (and override either of the previous implementations).

## Exposing search suggestions to Quick Search Box

Once you configure your application to provide custom search suggestions, making them available to the globally accessible Quick Search Box is as easy as modifying your searchable configuration to include an `android:includeInGlobalSearch` as "true".

The only scenario in which additional work is necessary is when your content provider demands a read permission. In which case, you need to add a special `<path-permission>` element for the provider to grant Quick Search Box read access to your content provider. For example:

```
<provider android:name="MySuggestionProvider"
          android:authorities="com.example.MyCustomSuggestionProvider"
          android:readPermission="com.example.provider.READ_MY_DATA"
          android:writePermission="com.example.provider.WRITE_MY_DATA">
    <path-permission android:pathPrefix="/search_suggest_query"
                    android:readPermission="android.permission.GLOBAL_SEARCH" />
</provider>
```

In this example, the provider restricts read and write access to the content. The `<path-permission>` element amends the restriction by granting read access to content inside the "/search\_suggest\_query" path prefix when the "android.permission.GLOBAL\_SEARCH" permission exists. This grants access to Quick Search Box so that it may query your content provider for suggestions.

If your content provider does not enforce read permissions, then Quick Search Box can read it by default.

## Enabling suggestions on a device

When your application is configured to provide suggestions in Quick Search Box, it is not actually enabled to provide suggestions in Quick Search Box, by default. It is the user's choice whether to include suggestions from your application in the Quick Search Box. To enable search suggestions from your application, the user must open "Searchable items" (in Settings > Search) and enable your application as a searchable item.

Each application that is available to Quick Search Box has an entry in the Searchable items settings page. The entry includes the name of the application and a short description of what content can be searched from the application and made available for suggestions in Quick Search Box. To define the description text for your searchable application, add the `android:searchSettingsDescription` attribute to your searchable configuration. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
            android:label="@string/app_label"
            android:hint="@string/search_hint"
            android:searchSuggestAuthority="com.example.MyCustomSuggestionProvider"
            android:searchSuggestIntentAction="android.intent.action.VIEW"
```

```
    android:includeInGlobalSearch="true"
    android:searchSettingsDescription="@string/search_description" >
</searchable>
```

The string for `android:searchSettingsDescription` should be as concise as possible and state the content that is searchable. For example, "Artists, albums, and tracks" for a music application, or "Saved notes" for a notepad application. Providing this description is important so the user knows what kind of suggestions are provided. You should always include this attribute when `android:includeInGlobalSearch` is "true".

Remember that the user must visit the settings menu to enable search suggestions for your application before your search suggestions appear in Quick Search Box. As such, if search is an important aspect of your application, then you might want to consider a way to convey that to your users — you might provide a note the first time they launch the app that instructs them how to enable search suggestions for Quick Search Box.

## Managing Quick Search Box suggestion shortcuts

Suggestions that the user selects from Quick Search Box can be automatically made into shortcuts. These are suggestions that the system has copied from your content provider so it can quickly access the suggestion without the need to re-query your content provider.

By default, this is enabled for all suggestions retrieved by Quick Search Box, but if your suggestion data changes over time, then you can request that the shortcuts be refreshed. For instance, if your suggestions refer to dynamic data, such as a contact's presence status, then you should request that the suggestion shortcuts be refreshed when shown to the user. To do so, include the [SUGGEST\\_COLUMN\\_SHORTCUT\\_ID](#) in your suggestions table. Using this column, you can configure the shortcut behavior for each suggestion in one of the following ways:

- a. Have Quick Search Box re-query your content provider for a fresh version of the suggestion shortcut.

Provide a value in the [SUGGEST\\_COLUMN\\_SHORTCUT\\_ID](#) column and the suggestion is re-queried for a fresh version each time the shortcut is displayed. The shortcut is quickly displayed with whatever data was most recently available until the refresh query returns, at which point the suggestion is refreshed with the new information. The refresh query is sent to your content provider with a URI path of [SUGGEST\\_URI\\_PATH\\_SHORTCUT](#) (instead of [SUGGEST\\_URI\\_PATH\\_QUERY](#)).

The [Cursor](#) you return should contain one suggestion using the same columns as the original suggestion, or be empty, indicating that the shortcut is no longer valid (in which case, the suggestion disappears and the shortcut is removed).

If a suggestion refers to data that could take longer to refresh, such as a network-based refresh, you can also add the [SUGGEST\\_COLUMN\\_SPINNER WHILE\\_REFRESHING](#) column to your suggestions table with a value of "true" in order to show a progress spinner for the right hand icon until the refresh is complete. Any value other than "true" does not show the progress spinner.

- b. Prevent the suggestion from being copied into a shortcut at all.

Provide a value of [SUGGEST\\_NEVER\\_MAKE\\_SHORTCUT](#) in the [SUGGEST\\_COLUMN\\_SHORTCUT\\_ID](#) column. In this case, the suggestion is never copied into a shortcut. This should only be necessary if you absolutely do not want the previously copied suggestion to appear. (Recall that if you provide a normal value for the column, then the suggestion shortcut appears only until the refresh query returns.)

- c. Allow the default shortcut behavior to apply.

Leave the `SUGGEST_COLUMN_SHORTCUT_ID` empty for each suggestion that will not change and can be saved as a shortcut.

If none of your suggestions ever change, then you do not need the `SUGGEST_COLUMN_SHORTCUT_ID` column at all.

**Note:** Quick Search Box ultimately decides whether or not to create a shortcut for a suggestion, considering these values as a strong request from your application—there is no guarantee that the behavior you have requested for your suggestion shortcuts will be honored.

## About Quick Search Box suggestion ranking

Once you make your application's search suggestions available to Quick Search Box, the Quick Search Box ranking determines how the suggestions are surfaced to the user for a particular query. This might depend on how many other apps have results for that query, and how often the user has selected your results compared to those from other apps. There is no guarantee about how your suggestions are ranked, or whether your app's suggestions show at all for a given query. In general, you can expect that providing quality results increases the likelihood that your app's suggestions are provided in a prominent position and apps that provide low quality suggestions are more likely to be ranked lower or not displayed.

See the [Searchable Dictionary sample app](#) for a complete demonstration of custom search suggestions.

# Searchable Configuration

## See also

1. [Creating a Search Interface](#)
2. [Adding Recent Query Suggestions](#)
3. [Adding Custom Suggestions](#)

In order to implement search with assistance from the Android system (to deliver search queries to an activity and provide search suggestions), your application must provide a search configuration in the form of an XML file.

This page describes the search configuration file in terms of its syntax and usage. For more information about how to implement search features for your application, begin with the developer guide about [Creating a Search Interface](#).

### file location:

`res/xml/filename.xml`

Android uses the filename as the resource ID.

### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="string resource"
    android:hint="string resource"
    android:searchMode=[ "queryRewriteFromData" | "queryRewriteFromText" ]
    android:searchButtonText="string resource"
    android:inputType="inputType"
    android:imeOptions="imeOptions"
    android:searchSuggestAuthority="string"
    android:searchSuggestPath="string"
    android:searchSuggestSelection="string"
    android:searchSuggestIntentAction="string"
    android:searchSuggestIntentData="string"
    android:searchSuggestThreshold="int"
    android:includeInGlobalSearch=[ "true" | "false" ]
    android:searchSettingsDescription="string resource"
    android:queryAfterZeroResults=[ "true" | "false" ]
    android:voiceSearchMode=[ "showVoiceSearchButton" | "launchWebSearch" | ...
    android:voiceLanguageModel=[ "free-form" | "web_search" ]
    android:voicePromptText="string resource"
    android:voiceLanguage="string"
    android:voiceMaxResults="int"
    >
    <actionkey
        android: keycode="KEYCODE"
        android: queryActionMsg="string"
        android: suggestActionMsg="string"
        android: suggestActionMsgColumn="string" >
</searchable>
```

**elements:**

## **<searchable>**

Defines all search configurations used by the Android system to provide assisted search.

attributes:

### **android:label**

*String resource.* (Required.) The name of your application. It should be the same as the name applied to the `android:label` attribute of your [activity](#) or [application](#) manifest element. This label is only visible to the user when you set `android:includeInGlobalSearch` to "true", in which case, this label is used to identify your application as a searchable item in the system's search settings.

### **android:hint**

*String resource.* (Recommended.) The text to display in the search text field when no text has been entered. It provides a hint to the user about what content is searchable. For consistency with other Android applications, you should format the string for `android:hint` as "Search <*content-or-product*>". For example, "Search songs and artists" or "Search YouTube".

### **android:searchMode**

*Keyword.* Sets additional modes that control the search presentation. Currently available modes define how the query text should be rewritten when a custom suggestion receives focus. The following mode values are accepted:

<b>Value</b>	<b>Description</b>
"queryRewriteFromText"	Use the value from the <a href="#">SUGGEST_COLUMN_TEXT_1</a> column to rewrite the query text.
"queryRewriteFromData"	Use the value from the <a href="#">SUGGEST_COLUMN_INTENT_DATA</a> column to rewrite the query text. This should only be used when the values in <a href="#">SUGGEST_COLUMN_INTENT_DATA</a> are suitable for user inspection and editing, typically HTTP URI's.

For more information, see the discussion about rewriting the query text in [Adding Custom Suggestions](#).

### **android:searchButtonText**

*String resource.* The text to display in the button that executes search. By default, the button shows a search icon (a magnifying glass), which is ideal for internationalization, so you should not use this attribute to change the button unless the behavior is something other than a search (such as a URL request in a web browser).

### **android:inputType**

*Keyword.* Defines the type of input method (such as the type of soft keyboard) to use. For most searches, in which free-form text is expected, you don't need this attribute. See [inputType](#) for a list of suitable values for this attribute.

### **android:imeOptions**

*Keyword.* Supplies additional options for the input method. For most searches, in which free-form text is expected, you don't need this attribute. The default IME is "actionSearch" (provides the "search" button instead of a carriage return in the soft keyboard). See [imeOptions](#) for a list of suitable values for this attribute.

## Search suggestion attributes

If you have defined a content provider to generate search suggestions, you need to define additional attributes that configure communications with the content provider. When providing search suggestions, you need some of the following <searchable> attributes:

### **android:searchSuggestAuthority**

*String.* (Required to provide search suggestions.) This value must match the authority string provided in the android:authorities attribute of the Android manifest <provider> element.

### **android:searchSuggestPath**

*String.* This path is used as a portion of the suggestions query [Uri](#), after the prefix and authority, but before the standard suggestions path. This is only required if you have a single content provider issuing different types of suggestions (such as for different data types) and you need a way to disambiguate the suggestions queries when you receive them.

### **android:searchSuggestSelection**

*String.* This value is passed into your query function as the selection parameter. Typically this is a WHERE clause for your database, and should contain a single question mark, which is a placeholder for the actual query string that has been typed by the user (for example, "query=?"). However, you can also use any non-null value to trigger the delivery of the query text via the selectionArgs parameter (and then ignore the selection parameter).

### **android:searchSuggestIntentAction**

*String.* The default intent action to be used when a user clicks on a custom search suggestion (such as "android.intent.action.VIEW"). If this is not overridden by the selected suggestion (via the [SUGGEST\\_COLUMN\\_INTENT\\_ACTION](#) column), this value is placed in the action field of the [Intent](#) when the user clicks a suggestion.

### **android:searchSuggestIntentData**

*String.* The default intent data to be used when a user clicks on a custom search suggestion. If not overridden by the selected suggestion (via the [SUGGEST\\_COLUMN\\_INTENT\\_DATA](#) column), this value is placed in the data field of the [Intent](#) when the user clicks a suggestion.

### **android:searchSuggestThreshold**

*Integer.* The minimum number of characters needed to trigger a suggestion look-up. Only guarantees that the system will not query your content provider for anything shorter than the threshold. The default value is 0.

For more information about the above attributes for search suggestions, see the guides for [Adding Recent Query Suggestions](#) and [Adding Custom Suggestions](#).

## Quick Search Box attributes

To make your custom search suggestions available to Quick Search Box, you need some of the following <searchable> attributes:

### **android:includeInGlobalSearch**

*Boolean.* (Required to provide search suggestions in Quick Search Box.) Set to "true" if you want your suggestions to be included in the globally accessible Quick Search Box. The user must still

enable your application as a searchable item in the system search settings before your suggestions will appear in Quick Search Box.

#### **android:searchSettingsDescription**

*String*. Provides a brief description of the search suggestions that you provide to Quick Search Box, which is displayed in the searchable items entry for your application. Your description should concisely describe the content that is searchable. For example, "Artists, albums, and tracks" for a music application, or "Saved notes" for a notepad application.

#### **android:queryAfterZeroResults**

*Boolean*. Set to "true" if you want your content provider to be invoked for supersets of queries that have returned zero results in the past. For example, if your content provider returned zero results for "bo", it should be required for "bob". If set to "false", supersets are ignored for a single session ("bob" does not invoke a requery). This lasts only for the life of the search dialog or the life of the activity when using the search widget (when the search dialog or activity is reopened, "bo" queries your content provider again). The default value is false.

### **Voice search attributes**

To enable voice search, you'll need some of the following <searchable> attributes:

#### **android:voiceSearchMode**

*Keyword*. (Required to provide voice search capabilities.) Enables voice search, with a specific mode for voice search. (Voice search may not be provided by the device, in which case these flags have no effect.) The following mode values are accepted:

<b>Value</b>	<b>Description</b>
"showVoiceSearchButton"	Display a voice search button, if voice search is available on the device. If set, then either "launchWebSearch" or "launchRecognizer" must also be set (separated by the pipe   character).
"launchWebSearch"	The voice search button takes the user directly to a built-in voice web search activity. Most applications don't need this flag, as it takes the user away from the activity in which search was invoked.
"launchRecognizer"	The voice search button takes the user directly to a built-in voice recording activity. This activity prompts the user to speak, transcribes the spoken text, and forwards the resulting query text to the searchable activity, just as if the user typed it into the search UI and clicked the search button.

#### **android:voiceLanguageModel**

*Keyword*. The language model that should be used by the voice recognition system. The following values are accepted:

<b>Value</b>	<b>Description</b>
"free_form"	Use free-form speech recognition for dictating queries. This is primarily optimized for English. This is the default.
"web_search"	Use web-search-term recognition for shorter, search-like phrases. This is available in more languages than "free_form".

Also see [EXTRA\\_LANGUAGE\\_MODEL](#) for more information.

**android:voicePromptText**

*String.* An additional message to display in the voice input dialog.

**android:voiceLanguage**

*String.* The spoken language to be expected, expressed as the string value of a constants in [Lo-cale](#) (such as "de" for German or "fr" for French). This is needed only if it is different from the current value of [Locale.getDefault\(\)](#).

**android:voiceMaxResults**

*Integer.* Forces the maximum number of results to return, including the "best" result which is always provided as the [ACTION\\_SEARCH](#) intent's primary query. Must be 1 or greater. Use [EX-TRA\\_RESULTS](#) to get the results from the intent. If not provided, the recognizer chooses how many results to return.

**<actionkey>**

Defines a device key and behavior for a search action. A search action provides a special behavior at the touch of a button on the device, based on the current query or focused suggestion. For example, the Contacts application provides a search action to initiate a phone call to the currently focused contact suggestion at the press of the CALL button.

Not all action keys are available on every device, and not all keys are allowed to be overridden in this way. For example, the "Home" key cannot be used and must always return to the home screen. Also be sure not to define an action key for a key that's needed for typing a search query. This essentially limits the available and reasonable action keys to the call button and menu button. Also note that action keys are not generally discoverable, so you should not provide them as a core user feature.

You must define the `android: keycode` to define the key and at least one of the other three attributes in order to define the search action.

attributes:

**android: keycode**

*String.* (Required.) A key code from [KeyEvent](#) that represents the action key you wish to respond to (for example "KEYCODE\_CALL"). This is added to the [ACTION\\_SEARCH](#) intent that is passed to your searchable activity. To examine the key code, use [getIntEx- tra\(SearchManager.ACTION\\_KEY\)](#). Not all keys are supported for a search action, as many of them are used for typing, navigation, or system functions.

**android:queryActionMsg**

*String.* An action message to be sent if the action key is pressed while the user is entering query text. This is added to the [ACTION\\_SEARCH](#) intent that the system passes to your searchable activity. To examine the string, use [getStringExtra\(SearchManager.ACTION\\_MSG\)](#).

**android: suggestActionMsg**

*String.* An action message to be sent if the action key is pressed while a suggestion is in focus. This is added to the intent that the system passes to your searchable activity (using the action you've defined for the suggestion). To examine the string, use [getStringEx- tra\(SearchManager.ACTION\\_MSG\)](#). This should only be used if all your suggestions support this action key. If not all suggestions can handle the same action key, then you must instead use the following `android:suggestActionMsgColumn` attribute.

**android: suggestActionMsgColumn**

*String.* The name of the column in your content provider that defines the action message for this action key, which is to be sent if the user presses the action key while a suggestion is in focus.

This attribute lets you control the action key on a suggestion-by-suggestion basis, because, instead of using the `android:suggestActionMsg` attribute to define the action message for all suggestions, each entry in your content provider provides its own action message.

First, you must define a column in your content provider for each suggestion to provide an action message, then provide the name of that column in this attribute. The system looks at your suggestion cursor, using the string provided here to select your action message column, and then select the action message string from the Cursor. That string is added to the intent that the system passes to your searchable activity (using the action you've defined for suggestions). To examine the string, use `getStringExtra(SearchManager.ACTION_MSG)`. If the data does not exist for the selected suggestion, the action key is ignored.

**example:**

XML file saved at `res/xml/searchable.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/search_label"
    android:hint="@string/search_hint"
    android:searchSuggestAuthority="dictionary"
    android:searchSuggestIntentAction="android.intent.action.VIEW"
    android:includeInGlobalSearch="true"
    android:searchSettingsDescription="@string/settings_description" >
</searchable>
```

# Drag and Drop

## Quickview

- Allow users to move data within your Activity layout using graphical gestures.
- Supports operations besides data movement.
- Only works within a single application.
- Requires API 11.

## In this document

1. [Overview](#)
  1. [The drag/drop process](#)
  2. [The drag event listener and callback method](#)
  3. [Drag events](#)
  4. [The drag shadow](#)
2. [Designing a Drag and Drop Operation](#)
  1. [Starting a drag](#)
  2. [Responding to a drag start](#)
  3. [Handling events during the drag](#)
  4. [Responding to a drop](#)
  5. [Responding to a drag end](#)
  6. [Responding to drag events: an example](#)

## Key classes

1. [View](#)
2. [OnLongClickListener](#)
3. [OnDragListener](#)
4. [DragEvent](#)
5. [DragShadowBuilder](#)
6. [ClipData](#)
7. [ClipDescription](#)

## Related Samples

1. [Honeycomb Gallery](#)
2. [DragAndDropDemo.java](#) and [DraggableDot.java](#) in [Api Demos](#).

## See also

1. [Content Providers](#)
2. [Input Events](#)

With the Android drag/drop framework, you can allow your users to move data from one View to another View in the current layout using a graphical drag and drop gesture. The framework includes a drag event class, drag listeners, and helper methods and classes.

Although the framework is primarily designed for data movement, you can use it for other UI actions. For example, you could create an app that mixes colors when the user drags a color icon over another icon. The rest of this topic, however, describes the framework in terms of data movement.

## Overview

A drag and drop operation starts when the user makes some gesture that you recognize as a signal to start dragging data. In response, your application tells the system that the drag is starting. The system calls back to your application to get a representation of the data being dragged. As the user's finger moves this representation ("drag shadow") over the current layout, the system sends drag events to the drag event listener objects and drag event callback methods associated with the [View](#) objects in the layout. Once the user releases the drag shadow, the system ends the drag operation.

You create a drag event listener object ("listeners") from a class that implements [View.OnDragListener](#). You set the drag event listener object for a View with the View object's [setOnDragListener\(\)](#) method. Each View object also has a [onDragEvent\(\)](#) callback method. Both of these are described in more detail in the section [The drag event listener and callback method](#).

**Note:** For the sake of simplicity, the following sections refer to the routine that receives drag events as the "drag event listener", even though it may actually be a callback method.

When you start a drag, you include both the data you are moving and metadata describing this data as part of the call to the system. During the drag, the system sends drag events to the drag event listeners or callback methods of each View in the layout. The listeners or callback methods can use the metadata to decide if they want to accept the data when it is dropped. If the user drops the data over a View object, and that View object's listener or callback method has previously told the system that it wants to accept the drop, then the system sends the data to the listener or callback method in a drag event.

Your application tells the system to start a drag by calling the [startDrag\(\)](#) method. This tells the system to start sending drag events. The method also sends the data that you are dragging.

You can call [startDrag\(\)](#) for any attached View in the current layout. The system only uses the View object to get access to global settings in your layout.

Once your application calls [startDrag\(\)](#), the rest of the process uses events that the system sends to the View objects in your current layout.

## The drag/drop process

There are basically four steps or states in the drag and drop process:

### *Started*

In response to the user's gesture to begin a drag, your application calls [startDrag\(\)](#) to tell the system to start a drag. The arguments [startDrag\(\)](#) provide the data to be dragged, metadata for this data, and a callback for drawing the drag shadow.

The system first responds by calling back to your application to get a drag shadow. It then displays the drag shadow on the device.

Next, the system sends a drag event with action type [ACTION\\_DRAG\\_STARTED](#) to the drag event listeners for all the View objects in the current layout. To continue to receive drag events, including a possible drop event, a drag event listener must return `true`. This registers the listener with the system. Only regis-

tered listeners continue to receive drag events. At this point, listeners can also change the appearance of their View object to show that the listener can accept a drop event.

If the drag event listener returns `false`, then it will not receive drag events for the current operation until the system sends a drag event with action type [ACTION\\_DRAG\\_ENDED](#). By sending `false`, the listener tells the system that it is not interested in the drag operation and does not want to accept the dragged data.

### **Continuing**

The user continues the drag. As the drag shadow intersects the bounding box of a View object, the system sends one or more drag events to the View object's drag event listener (if it is registered to receive events). The listener may choose to alter its View object's appearance in response to the event. For example, if the event indicates that the drag shadow has entered the bounding box of the View (action type [ACTION\\_DRAG\\_ENTERED](#)), the listener can react by highlighting its View.

### **Dropped**

The user releases the drag shadow within the bounding box of a View that can accept the data. The system sends the View object's listener a drag event with action type [ACTION\\_DROP](#). The drag event contains the data that was passed to the system in the call to [startDrag\(\)](#) that started the operation. The listener is expected to return boolean `true` to the system if code for accepting the drop succeeds.

Note that this step only occurs if the user drops the drag shadow within the bounding box of a View whose listener is registered to receive drag events. If the user releases the drag shadow in any other situation, no [ACTION\\_DROP](#) drag event is sent.

### **Ended**

After the user releases the drag shadow, and after the system sends out (if necessary) a drag event with action type [ACTION\\_DROP](#), the system sends out a drag event with action type [ACTION\\_DRAG\\_ENDED](#) to indicate that the drag operation is over. This is done regardless of where the user released the drag shadow. The event is sent to every listener that is registered to receive drag events, even if the listener received the [ACTION\\_DROP](#) event.

Each of these four steps is described in more detail in the section [Designing a Drag and Drop Operation](#).

## **The drag event listener and callback method**

A View receives drag events with either a drag event listener that implements [View.OnDragListener](#) or with its [onDragEvent\(DragEvent\)](#) callback method. When the system calls the method or listener, it passes to them a [DragEvent](#) object.

You will probably want to use the listener in most cases. When you design UIs, you usually don't subclass View classes, but using the callback method forces you to do this in order to override the method. In comparison, you can implement one listener class and then use it with several different View objects. You can also implement it as an anonymous inline class. To set the listener for a View object, call [setOnDragListener\(\)](#).

You can have both a listener and a callback method for View object. If this occurs, the system first calls the listener. The system doesn't call the callback method unless the listener returns `false`.

The combination of the [onDragEvent\(DragEvent\)](#) method and [View.OnDragListener](#) is analogous to the combination of the [onTouchEvent\(\)](#) and [View.OnTouchListener](#) used with touch events.

## Drag events

The system sends out a drag event in the form of a [DragEvent](#) object. The object contains an action type that tells the listener what is happening in the drag/drop process. The object contains other data, depending on the action type.

To get the action type, a listener calls [getAction\(\)](#). There are six possible values, defined by constants in the [DragEvent](#) class. These are listed in [table 1](#).

The [DragEvent](#) object also contains the data that your application provided to the system in the call to [startDrag\(\)](#). Some of the data is valid only for certain action types. The data that is valid for each action type is summarized in [table 2](#). It is also described in detail with the event for which it is valid in the section [Designing a Drag and Drop Operation](#).

**Table 1.** DragEvent action types

getAction() value	Meaning
<a href="#">ACTION_DRAG_STARTED</a>	A View object's drag event listener receives this event action type just after the application calls <a href="#">startDrag()</a> and gets a drag shadow.
<a href="#">ACTION_DRAG_ENTERED</a>	A View object's drag event listener receives this event action type when the drag shadow has just entered the bounding box of the View. This is the first event action type the listener receives when the drag shadow enters the bounding box. If the listener wants to continue receiving drag events for this operation, it must return boolean <code>true</code> to the system.
<a href="#">ACTION_DRAG_LOCATION</a>	A View object's drag event listener receives this event action type after it receives a <a href="#">ACTION_DRAG_ENTERED</a> event while the drag shadow is still within the bounding box of the View.
<a href="#">ACTION_DRAG_EXITED</a>	A View object's drag event listener receives this event action type after it receives a <a href="#">ACTION_DRAG_ENTERED</a> and at least one <a href="#">ACTION_DRAG_LOCATION</a> event, and after the user has moved the drag shadow outside the bounding box of the View.
<a href="#">ACTION_DROP</a>	A View object's drag event listener receives this event action type when the user releases the drag shadow over the View object. This action type is only sent to a View object's listener if the listener returned boolean <code>true</code> in response to the <a href="#">ACTION_DRAG_STARTED</a> drag event. This action type is not sent if the user releases the drag shadow on a View whose listener is not registered, or if the user releases the drag shadow on anything that is not part of the current layout.
	The listener is expected to return boolean <code>true</code> if it successfully processes the drop. Otherwise, it should return <code>false</code> .
<a href="#">ACTION_DRAG_ENDED</a>	A View object's drag event listener receives this event action type when the system is ending the drag operation. This action type is not necessarily preceded by an <a href="#">ACTION_DROP</a> event. If the system sent a <a href="#">ACTION_DROP</a> , receiving the <a href="#">ACTION_DRAG_ENDED</a> action type does not imply that the drop operation succeeded. The listener must call <a href="#">getResult()</a> to get the value that was returned in response to <a href="#">ACTION_DROP</a> . If an <a href="#">ACTION_DROP</a> event was not sent, then <a href="#">getResult()</a> returns <code>false</code> .

**Table 2.** Valid DragEvent data by action type

<a href="#">getAction()</a> value	<a href="#">getClipDescription()</a>	<a href="#">getLocalState()</a>	<a href="#">getX()</a>	<a href="#">getY()</a>	<a href="#">getClipData()</a>	<a href="#">value</a>
<a href="#">ACTION_DRAG_STARTED</a>	X		X		X	
<a href="#">ACTION_DRAG_ENTERED</a>	X		X		X	X
<a href="#">ACTION_DRAG_LOCATION</a>	X		X		X	X
<a href="#">ACTION_DRAG_EXITED</a>	X		X			
<a href="#">ACTION_DROP</a>	X		X		X	X
<a href="#">ACTION_DRAG_ENDED</a>	X		X			X

The [getAction\(\)](#), [describeContents\(\)](#), [writeToParcel\(\)](#), and [toString\(\)](#) methods always return valid data.

If a method does not contain valid data for a particular action type, it returns either `null` or 0, depending on its result type.

## The drag shadow

During a drag and drop operation, the system displays a image that the user drags. For data movement, this image represents the data being dragged. For other operations, the image represents some aspect of the drag operation.

The image is called a drag shadow. You create it with methods you declare for a [View.DragShadowBuilder](#) object, and then pass it to the system when you start a drag using [startDrag\(\)](#). As part of its response to [startDrag\(\)](#), the system invokes the callback methods you've defined in [View.DragShadowBuilder](#) to obtain a drag shadow.

The [View.DragShadowBuilder](#) class has two constructors:

### [View.DragShadowBuilder\(View\)](#)

This constructor accepts any of your application's [View](#) objects. The constructor stores the View object in the [View.DragShadowBuilder](#) object, so during the callback you can access it as you construct your drag shadow. It doesn't have to be associated with the View (if any) that the user selected to start the drag operation.

If you use this constructor, you don't have to extend [View.DragShadowBuilder](#) or override its methods. By default, you will get a drag shadow that has the same appearance as the View you pass as an argument, centered under the location where the user is touching the screen.

### [View.DragShadowBuilder\(\)](#)

If you use this constructor, no View object is available in the [View.DragShadowBuilder](#) object (the field is set to `null`). If you use this constructor, and you don't extend [View.DragShadowBuilder](#) or override its methods, you will get an invisible drag shadow. The system does *not* give an error.

The [View.DragShadowBuilder](#) class has two methods:

### [onProvideShadowMetrics\(\)](#)

The system calls this method immediately after you call [startDrag\(\)](#). Use it to send to the system the dimensions and touch point of the drag shadow. The method has two arguments:

#### [dimensions](#)

A [Point](#) object. The drag shadow width goes in [x](#) and its height goes in [y](#).

### ***touch\_point***

A [Point](#) object. The touch point is the location within the drag shadow that should be under the user's finger during the drag. Its X position goes in [x](#) and its Y position goes in [y](#)

### **onDrawShadow()**

Immediately after the call to [onProvideShadowMetrics\(\)](#) the system calls [onDrawShadow\(\)](#) to get the drag shadow itself. The method has a single argument, a [Canvas](#) object that the system constructs from the parameters you provide in [onProvideShadowMetrics\(\)](#). Use it to draw the drag shadow in the provided [Canvas](#) object.

To improve performance, you should keep the size of the drag shadow small. For a single item, you may want to use a icon. For a multiple selection, you may want to use icons in a stack rather than full images spread out over the screen.

## **Designing a Drag and Drop Operation**

This section shows step-by-step how to start a drag, how to respond to events during the drag, how respond to a drop event, and how to end the drag and drop operation.

### **Starting a drag**

The user starts a drag with a drag gesture, usually a long press, on a View object. In response, you should do the following:

1. As necessary, create a [ClipData](#) and [ClipData.Item](#) for the data being moved. As part of the ClipData object, supply metadata that is stored in a [ClipDescription](#) object within the ClipData. For a drag and drop operation that does not represent data movement, you may want to use null instead of an actual object.

For example, this code snippet shows how to respond to a long press on a ImageView by creating a ClipData object that contains the tag or label of an ImageView. Following this snippet, the next snippet shows how to override the methods in [View.DragShadowBuilder](#):

```
// Create a string for the ImageView label
private static final String IMAGEVIEW_TAG = "icon bitmap"

// Creates a new ImageView
ImageView imageView = new ImageView(this);

// Sets the bitmap for the ImageView from an icon bit map (defined elsewhere)
imageView.setImageBitmap(mIconBitmap);

// Sets the tag
imageView.setTag(IMAGEVIEW_TAG);

...

// Sets a long click listener for the ImageView using an anonymous listener
// implements the OnLongClickListener interface
imageView.setOnLongClickListener(new View.OnLongClickListener() {

    // Defines the one method for the interface, which is called when the
    public boolean onLongClick(View v) {
```

```

// Create a new ClipData.
// This is done in two steps to provide clarity. The convenience method
// ClipData.newPlainText() can create a plain text ClipData in one step.

// Create a new ClipData.Item from the ImageView object's tag
ClipData.Item item = new ClipData.Item(v.getTag());

// Create a new ClipData using the tag as a label, the plain text MIME type
// the already-created item. This will create a new ClipDescription object
// ClipData, and set its MIME type entry to "text/plain"
ClipData dragData = new ClipData(v.getTag(), ClipData.MIMETYPE_TEXT_PLAIN);

// Instantiates the drag shadow builder.
View.DragShadowBuilder myShadow = new MyDragShadowBuilder(imageView);

// Starts the drag

        v.startDrag(dragData,           // the data to be dragged
                    myShadow,          // the drag shadow builder
                    null,              // no need to use local data
                    0                  // flags (not currently used, set to 0)
            );

    }

}

```

2. The following code snippet defines myDragShadowBuilder. It creates a drag shadow for dragging a TextView as a small gray rectangle:

```

private static class MyDragShadowBuilder extends View.DragShadowBuilder {

    // The drag shadow image, defined as a drawable thing
    private static Drawable shadow;

    // Defines the constructor for myDragShadowBuilder
    public MyDragShadowBuilder(View v) {

        // Stores the View parameter passed to myDragShadowBuilder.
        super(v);

        // Creates a draggable image that will fill the Canvas provided
        shadow = new ColorDrawable(Color.LTGRAY);
    }

    // Defines a callback that sends the drag shadow dimensions and touch
    // system.
    @Override
    public void onProvideShadowMetrics (Point size, Point touch)
        // Defines local variables
        private int width, height;

        // Sets the width of the shadow to half the width of the original view
        width = getView().getWidth() / 2;
    }
}

```

```

        // Sets the height of the shadow to half the height of the or
height = getView().getHeight() / 2;

        // The drag shadow is a ColorDrawable. This sets its dimension
        // Canvas that the system will provide. As a result, the drag
        // Canvas.
shadow.setBounds(0, 0, width, height);

        // Sets the size parameter's width and height values. These g
        // through the size parameter.
size.set(width, height);

        // Sets the touch point's position to be in the middle of the
touch.set(width / 2, height / 2);
}

// Defines a callback that draws the drag shadow in a Canvas that
// from the dimensions passed in onProvideShadowMetrics().
@Override
public void onDrawShadow(Canvas canvas) {

    // Draws the ColorDrawable in the Canvas passed in from the s
shadow.draw(canvas);
}
}

```

**Note:** Remember that you don't have to extend [View.DragShadowBuilder](#). The constructor [View.DragShadowBuilder\(View\)](#) creates a default drag shadow that's the same size as the View argument passed to it, with the touch point centered in the drag shadow.

## Responding to a drag start

During the drag operation, the system dispatches drag events to the drag event listeners of the View objects in the current layout. The listeners should react by calling [getAction\(\)](#) to get the action type. At the start of a drag, this methods returns [ACTION\\_DRAG\\_STARTED](#).

In response to an event with the action type [ACTION\\_DRAG\\_STARTED](#), a listener should do the following:

1. Call [getClipDescription\(\)](#) to get the [ClipDescription](#). Use the MIME type methods in [ClipDescription](#) to see if the listener can accept the data being dragged.  
If the drag and drop operation does not represent data movement, this may not be necessary.
2. If the listener can accept a drop, it should return `true`. This tells the system to continue to send drag events to the listener. If it can't accept a drop, it should return `false`, and the system will stop sending drag events until it sends out [ACTION\\_DRAG\\_ENDED](#).

Note that for an [ACTION\\_DRAG\\_STARTED](#) event, these the following [DragEvent](#) methods are not valid: [getClipData\(\)](#), [getX\(\)](#), [getY\(\)](#), and [getResult\(\)](#).

## Handling events during the drag

During the drag, listeners that returned `true` in response to the `ACTION_DRAG_STARTED` drag event continue to receive drag events. The types of drag events a listener receives during the drag depend on the location of the drag shadow and the visibility of the listener's View.

During the drag, listeners primarily use drag events to decide if they should change the appearance of their View.

During the drag, `getAction()` returns one of three values:

- `ACTION_DRAG_ENTERED`: The listener receives this when the touch point (the point on the screen underneath the user's finger) has entered the bounding box of the listener's View.
- `ACTION_DRAG_LOCATION`: Once the listener receives an `ACTION_DRAG_ENTERED` event, and before it receives an `ACTION_DRAG_EXITED` event, it receives a new `ACTION_DRAG_LOCATION` event every time the touch point moves. The `getX()` and `getY()` methods return the X and Y coordinates of the touch point.
- `ACTION_DRAG_EXITED`: This event is sent to a listener that previously received `ACTION_DRAG_ENTERED`, after the drag shadow is no longer within the bounding box of the listener's View.

The listener does not need to react to any of these action types. If the listener returns a value to the system, it is ignored. Here are some guidelines for responding to each of these action types:

- In response to `ACTION_DRAG_ENTERED` or `ACTION_DRAG_LOCATION`, the listener can change the appearance of the View to indicate that it is about to receive a drop.
- An event with the action type `ACTION_DRAG_LOCATION` contains valid data for `getX()` and `getY()`, corresponding to the location of the touch point. The listener may want to use this information to alter the appearance of that part of the View that is at the touch point. The listener can also use this information to determine the exact position where the user is going to drop the drag shadow.
- In response to `ACTION_DRAG_EXITED`, the listener should reset any appearance changes it applied in response to `ACTION_DRAG_ENTERED` or `ACTION_DRAG_LOCATION`. This indicates to the user that the View is no longer an imminent drop target.

## Responding to a drop

When the user releases the drag shadow on a View in the application, and that View previously reported that it could accept the content being dragged, the system dispatches a drag event to that View with the action type `ACTION_DROP`. The listener should do the following:

1. Call `getClipData()` to get the `ClipData` object that was originally supplied in the call to `startDrag()` and store it. If the drag and drop operation does not represent data movement, this may not be necessary.
2. Return boolean `true` to indicate that the drop was processed successfully, or boolean `false` if it was not. The returned value becomes the value returned by `getResult()` for an `ACTION_DRAG_ENDED` event.

Note that if the system does not send out an `ACTION_DROP` event, the value of `getResult()` for an `ACTION_DRAG_ENDED` event is `false`.

For an `ACTION_DROP` event, `getX()` and `getY()` return the X and Y position of the drag point at the moment of the drop, using the coordinate system of the View that received the drop.

The system does allow the user to release the drag shadow on a View whose listener is not receiving drag events. It will also allow the user to release the drag shadow on empty regions of the application's UI, or on areas outside of your application. In all of these cases, the system does not send an event with action type [ACTION\\_DROP](#), although it does send out an [ACTION\\_DRAG\\_ENDED](#) event.

## Responding to a drag end

Immediately after the user releases the drag shadow, the system sends a drag event to all of the drag event listeners in your application, with an action type of [ACTION\\_DRAG\\_ENDED](#). This indicates that the drag operation is over.

Each listener should do the following:

1. If listener changed its View object's appearance during the operation, it should reset the View to its default appearance. This is a visual indication to the user that the operation is over.
2. The listener can optionally call [getRESULT\(\)](#) to find out more about the operation. If a listener returned `true` in response to an event of action type [ACTION\\_DROP](#), then [getRESULT\(\)](#) will return boolean `true`. In all other cases, [getRESULT\(\)](#) returns boolean `false`, including any case in which the system did not send out a [ACTION\\_DROP](#) event.
3. The listener should return boolean `true` to the system.

## Responding to drag events: an example

All drag events are initially received by your drag event method or listener. The following code snippet is a simple example of reacting to drag events in a listener:

```
// Creates a new drag event listener
mDragListen = new myDragEventListener();

View imageView = new ImageView(this);

// Sets the drag event listener for the View
imageView.setOnDragListener(mDragListen);

...

protected class myDragEventListener implements View.OnDragEventListener {

    // This is the method that the system calls when it dispatches a drag event
    // listener.
    public boolean onDrag(View v, DragEvent event) {

        // Defines a variable to store the action type for the incoming event
        final int action = event.getAction();

        // Handles each of the expected events
        switch(action) {

            case DragEvent.ACTION_DRAG_STARTED:

                // Determines if this View can accept the dragged data
                if (event.getClipDescription().hasMimeType(ClipDescription.MIME
```

```
// applies a blue color tint to the View to indicate that i
// data.
v.setColorFilter(Color.BLUE);

// Invalidate the view to force a redraw in the new tint
v.invalidate();

// returns true to indicate that the View can accept the dr
return(true);

} else {

// Returns false. During the current drag and drop operatio
// not receive events again until ACTION_DRAG_ENDED is sent
return(false);

}

break;

case DragEvent.ACTION_DRAG_ENTERED: {

// Applies a green tint to the View. Return true; the return va

v.setColorFilter(Color.GREEN);

// Invalidate the view to force a redraw in the new tint
v.invalidate();

return(true);

break;

case DragEvent.ACTION_DRAG_LOCATION:

// Ignore the event
return(true);

break;

case DragEvent.ACTION_DRAG_EXITED:

// Re-sets the color tint to blue. Returns true; the return va
v.setColorFilter(Color.BLUE);

// Invalidate the view to force a redraw in the new tint
v.invalidate();

return(true);

break;

case DragEvent.ACTION_DROP:

// Gets the item containing the dragged data
```

```
ClipData.Item item = event.getClipData().getItemAt(0);

// Gets the text data from the item.
dragData = item.getText();

// Displays a message containing the dragged data.
Toast.makeText(this, "Dragged data is " + dragData, Toast.LENGTH_SHORT).show();

// Turns off any color tints
v.clearColorFilter();

// Invalidates the view to force a redraw
v.invalidate();

// Returns true. DragEvent.getResult() will return true.
return(true);

break;

case DragEvent.ACTION_DRAG_ENDED:

    // Turns off any color tinting
    v.clearColorFilter();

    // Invalidates the view to force a redraw
    v.invalidate();

    // Does a getResult(), and displays what happened.
    if (event.getResult()) {
        Toast.makeText(this, "The drop was handled.", Toast.LENGTH_SHORT).show();
    } else {
        Toast.makeText(this, "The drop didn't work.", Toast.LENGTH_SHORT).show();
    }

    // returns true; the value is ignored.
    return(true);

break;

// An unknown action type was received.
default:
    Log.e("DragDrop Example", "Unknown action type received by " + v);
    break;
};

};

};
```

# Accessibility

## Topics

1. [Making Applications Accessible](#)
2. [Accessibility Developer Checklist](#)
3. [Building Accessibility Services](#)

## See also

1. [Android Design: Accessibility](#)
2. [Training: Implementing Accessibility](#)
3. [Accessibility Testing Checklist](#)

## Related Videos

- 1.
- 2.

Many Android users have different abilities that require them to interact with their Android devices in different ways. These include users who have visual, physical or age-related limitations that prevent them from fully seeing or using a touchscreen, and users with hearing loss who may not be able to perceive audible information and alerts.

Android provides accessibility features and services for helping these users navigate their devices more easily, including text-to-speech, haptic feedback, gesture navigation, trackball and directional-pad navigation. Android application developers can take advantage of these services to make their applications more accessible.

Android developers can also build their own accessibility services, which can provide enhanced usability features such as audio prompting, physical feedback, and alternative navigation modes. Accessibility services can provide these enhancements for all applications, a set of applications or just a single app.

The following topics show you how to use the Android framework to make applications more accessible.

### [Making Applications Accessible](#)

Development practices and API features to ensure your application is accessible to users with disabilities.

### [Accessibility Developer Checklist](#)

A checklist to help developers ensure that their applications are accessible.

### [Building Accessibility Services](#)

How to use API features to build services that make other applications more accessible for users.

# Making Applications Accessible

## In this document

1. [Labeling User Interface Elements](#)
2. [Enabling Focus Navigation](#)
  1. [Enabling view focus](#)
  2. [Controlling focus order](#)
3. [Building Accessible Custom Views](#)
  1. [Handling directional controller clicks](#)
  2. [Implementing accessibility API methods](#)
  3. [Sending accessibility events](#)
  4. [Populating accessibility events](#)
  5. [Providing a customized accessibility context](#)
  6. [Handling custom touch events](#)
4. [Testing Accessibility](#)

## Key classes

1. [AccessibilityEvent](#)
2. [AccessibilityNodeInfo](#)
3. [AccessibilityNodeInfoCompat](#)
4. [View.AccessibilityDelegate](#)
5. [AccessibilityDelegateCompat](#)

## See also

1. [Accessibility Developer Checklist](#)
- 2.
3. [Accessibility Testing Checklist](#)
- 4.
5. [Android Design: Accessibility](#)
6. [Designing Effective Navigation](#)
7. [Training: Implementing Accessibility](#)

Applications built for Android are more accessible to users with visual, physical or age-related limitations when those users activate accessibility services and features on a device. These services make your application more accessible even if you do not make any accessibility changes to your code. However, there are steps you should take to optimize the accessibility of your application and ensure a pleasant experience for all your users.

Making sure your application is accessible to all users requires only a few steps, particularly when you create your user interface with the components provided by the Android framework. If you use only the standard components for your application, the steps are:

1. Add descriptive text to user interface controls in your application using the [an-droid:contentDescription](#) attribute. Pay particular attention to [ImageButton](#), [Image-View](#) and [CheckBox](#).
2. Make sure that all user interface elements that can accept input (touches or typing) can be reached with a directional controller, such as a trackball, D-pad (physical or virtual) or navigation [gestures](#).
3. Make sure that audio prompts are always accompanied by another visual prompt or notification, to assist users who are deaf or hard of hearing.

4. Test your application using only accessibility navigation services and features. Turn on [TalkBack](#) and [Explore by Touch](#), and then try using your application using only directional controls. For more information on testing for accessibility, see the [Accessibility Testing Checklist](#).

If you build custom controls that extend the [View](#) class, you must complete some additional work to make sure your components are accessible. This document discusses how to make custom view controls compatible with accessibility services.

**Note:** The implementation steps in this document describe the requirements for making your application accessible for users with blindness or low-vision. Be sure to review the requirements for serving users who are deaf and hard of hearing in the [Accessibility Developer Checklist](#)

## Labeling User Interface Elements

Many user interface controls depend on visual cues to indicate their meaning and usage. For example, a note-taking application might use an [ImageButton](#) with a picture of a plus sign to indicate that the user can add a new note. An [EditText](#) component may have a label near it that indicates its purpose. A user with impaired vision can't see these cues well enough to follow them, which makes them useless.

You can make these controls more accessible with the [android:contentDescription](#) XML layout attribute. The text in this attribute does not appear on screen, but if the user enables accessibility services that provide audible prompts, then when the user navigates to that control, the text is spoken.

For this reason, set the [android:contentDescription](#) attribute for every [ImageButton](#), [Imageview](#), [CheckBox](#) in your application's user interface, and add descriptions to any other input controls that might require additional information for users who are not able to see it.

For example, the following [ImageButton](#) sets the content description for the plus button to the `add_note` string resource, which could be defined as "Add note" for an English language interface:

```
<ImageButton  
    android:id="@+id/add_note_button"  
    android:src="@drawable/add_note"  
    android:contentDescription="@string/add_note"/>
```

By including the description, an accessibility service that provides spoken feedback can announce "Add note" when a user moves focus to this button or hovers over it.

**Note:** For [EditText](#) fields, provide an [android:hint](#) attribute *instead* of a content description, to help users understand what content is expected when the text field is empty. When the field is filled, TalkBack reads the entered content to the user, instead of the hint text.

## Enabling Focus Navigation

Focus navigation allows users with disabilities to step through user interface controls using a directional controller. Directional controllers can be physical, such as a trackball, directional pad (D-pad) or arrow keys, or virtual, such as the [Eyes-Free Keyboard](#), or the gestures navigation mode available in Android 4.1 and higher. Directional controllers are a primary means of navigation for many Android users.

To ensure that users can navigate your application using only a directional controller, verify that all user interface (UI) input controls in your application can be reached and activated without using the touchscreen. You

should also verify that clicking with the center button (or OK button) of a directional controller has the same effect as touching a control that already has focus. For information on testing directional controls, see [Testing focus navigation](#).

## Enabling view focus

A user interface element is reachable using directional controls when its `android:focusable` attribute is set to `true`. This setting allows users to focus on the element using the directional controls and then interact with it. The user interface controls provided by the Android framework are focusable by default and visually indicate focus by changing the control's appearance.

Android provides several APIs that let you control whether a user interface control is focusable and even request that a control be given focus:

- [setFocusable\(\)](#)
- [isFocusable\(\)](#)
- [requestFocus\(\)](#)

If a view is not focusable by default, you can make it focusable in your layout file by setting the `an-`  
`droid:focusable` attribute to `true` or by calling the its [setFocusable\(\)](#) method.

## Controlling focus order

When users navigate in any direction using directional controls, focus is passed from one user interface element (view) to another, as determined by the focus order. This order is based on an algorithm that finds the nearest neighbor in a given direction. In rare cases, the algorithm may not match the order that you intended or may not be logical for users. In these situations, you can provide explicit overrides to the ordering using the following XML attributes in your layout file:

### `android:nextFocusDown`

Defines the next view to receive focus when the user navigates down.

### `android:nextFocusLeft`

Defines the next view to receive focus when the user navigates left.

### `android:nextFocusRight`

Defines the next view to receive focus when the user navigates right.

### `android:nextFocusUp`

Defines the next view to receive focus when the user navigates up.

The following example XML layout shows two focusable user interface elements where the `an-`  
`droid:nextFocusDown` and `android:nextFocusUp` attributes have been explicitly set. The  
`TextView` is located to the right of the `EditText`. However, since these properties have been set, the  
`TextView` element can now be reached by pressing the down arrow when focus is on the `EditText` ele-  
ment:

```
<LinearLayout android:orientation="horizontal"
    ...
    <EditText android:id="@+id/edit"
        android:nextFocusDown="@+id/text"
        ...
    >
    <TextView android:id="@+id/text"
        android:focusable="true"
```

```
    android:text="Hello, I am a focusable TextView"
    android:nextFocusUp="@+id/edit"
    ...
</LinearLayout>
```

When modifying focus order, be sure that the navigation works as expected in all directions from each user interface control and when navigating in reverse (to get back to where you came from).

**Note:** You can modify the focus order of user interface components at runtime, using methods such as [setNextFocusDownId\(\)](#) and [setNextFocusRightId\(\)](#).

## Building Accessible Custom Views

If your application requires a [custom view component](#), you must do some additional work to ensure that your custom view is accessible. These are the main tasks for ensuring the accessibility of your view:

- Handle directional controller clicks
- Implement accessibility API methods
- Send [AccessibilityEvent](#) objects specific to your custom view
- Populate [AccessibilityEvent](#) and [AccessibilityNodeInfo](#) for your view

### Handling directional controller clicks

On most devices, clicking a view using a directional controller sends a [KeyEvent](#) with [KEY\\_CODE\\_DPAD\\_CENTER](#) to the view currently in focus. All standard Android views already handle [KEY\\_CODE\\_DPAD\\_CENTER](#) appropriately. When building a custom [View](#) control, make sure this event has the same effect as touching the view on the touchscreen.

Your custom control should also treat the [KEYCODE\\_ENTER](#) event the same as [KEYCODE\\_DPAD\\_CENTER](#). This approach makes interaction from a full keyboard much easier for users.

### Implementing accessibility API methods

Accessibility events are messages about users interaction with visual interface components in your application. These messages are handled by [Accessibility Services](#), which use the information in these events to produce supplemental feedback and prompts. In Android 4.0 (API Level 14) and higher, the methods for generating accessibility events have been expanded to provide more detailed information than the [AccessibilityEventSource](#) interface introduced in Android 1.6 (API Level 4). The expanded accessibility methods are part of the [View](#) class as well as the [View.AccessibilityDelegate](#) class. The methods are as follows:

#### [sendAccessibilityEvent\(\)](#)

(API Level 4) This method is called when a user takes action on a view. The event is classified with a user action type such as [TYPE\\_VIEW\\_CLICKED](#). You typically do not need to implement this method unless you are creating a custom view.

#### [sendAccessibilityEventUnchecked\(\)](#)

(API Level 4) This method is used when the calling code needs to directly control the check for accessibility being enabled on the device ([AccessibilityManager.isEnabled\(\)](#)). If you do implement this method, you must perform the call as if accessibility is enabled, regardless of the actual system setting. You typically do not need to implement this method for a custom view.

## [dispatchPopulateAccessibilityEvent\(\)](#)

(API Level 4) The system calls this method when your custom view generates an accessibility event. As of API Level 14, the default implementation of this method calls [onPopulateAccessibilityEvent\(\)](#) for this view and then the [dispatchPopulateAccessibilityEvent\(\)](#) method for each child of this view. In order to support accessibility services on revisions of Android *prior* to 4.0 (API Level 14) you *must* override this method and populate [getText\(\)](#) with descriptive text for your custom view, which is spoken by accessibility services, such as TalkBack.

## [onPopulateAccessibilityEvent\(\)](#)

(API Level 14) This method sets the spoken text prompt of the [AccessibilityEvent](#) for your view. This method is also called if the view is a child of a view which generates an accessibility event.

**Note:** Modifying additional attributes beyond the text within this method potentially overwrites properties set by other methods. While you can modify attributes of the accessibility event with this method, you should limit these changes to text content, and use the [onInitializeAccessibilityEvent\(\)](#) method to modify other properties of the event.

**Note:** If your implementation of this event completely overrides the output text without allowing other parts of your layout to modify its content, then do not call the super implementation of this method in your code.

## [onInitializeAccessibilityEvent\(\)](#)

(API Level 14) The system calls this method to obtain additional information about the state of the view, beyond text content. If your custom view provides interactive control beyond a simple [TextView](#) or [Button](#), you should override this method and set the additional information about your view into the event using this method, such as password field type, checkbox type or states that provide user interaction or feedback. If you do override this method, you must call its super implementation and then only modify properties that have not been set by the super class.

## [onInitializeAccessibilityNodeInfo\(\)](#)

(API Level 14) This method provides accessibility services with information about the state of the view. The default [View](#) implementation has a standard set of view properties, but if your custom view provides interactive control beyond a simple [TextView](#) or [Button](#), you should override this method and set the additional information about your view into the [AccessibilityNodeInfo](#) object handled by this method.

## [onRequestSendAccessibilityEvent\(\)](#)

(API Level 14) The system calls this method when a child of your view has generated an [AccessibilityEvent](#). This step allows the parent view to amend the accessibility event with additional information. You should implement this method only if your custom view can have child views and if the parent view can provide context information to the accessibility event that would be useful to accessibility services.

In order to support these accessibility methods for a custom view, you should take one of the following approaches:

- If your application targets Android 4.0 (API level 14) and higher, override and implement the accessibility methods listed above directly in your custom view class.
- If your custom view is intended to be compatible with Android 1.6 (API Level 4) and above, add the Android [Support Library](#), revision 5 or higher, to your project. Then, within your custom view class, call the [ViewCompat.setAccessibilityDelegate\(\)](#) method to implement the accessibility methods above. For an example of this approach, see the Android Support Library (revision 5 or higher) sample AccessibilityDelegateSupportActivity in (<sdk>/extras/android/support/v4/samples/Support4Demos/)

In either case, you should implement the following accessibility methods for your custom view class:

- [dispatchPopulateAccessibilityEvent\(\)](#)
- [onPopulateAccessibilityEvent\(\)](#)
- [onInitializeAccessibilityEvent\(\)](#)
- [onInitializeAccessibilityNodeInfo\(\)](#)

For more information about implementing these methods, see [Populating Accessibility Events](#).

## Sending accessibility events

Depending on the specifics of your custom view, it may need to send [AccessibilityEvent](#) objects at a different times or for events not handled by the default implementation. The [View](#) class provides a default implementation for these event types:

- Starting with API Level 4:
  - [TYPE\\_VIEW\\_CLICKED](#)
  - [TYPE\\_VIEW\\_LONG\\_CLICKED](#)
  - [TYPE\\_VIEW\\_FOCUSED](#)
- Starting with API Level 14:
  - [TYPE\\_VIEW\\_SCROLLED](#)
  - [TYPE\\_VIEW\\_HOVER\\_ENTER](#)
  - [TYPE\\_VIEW\\_HOVER\\_EXIT](#)

**Note:** Hover events are associated with the Explore by Touch feature, which uses these events as triggers for providing audible prompts for user interface elements.

In general, you should send an [AccessibilityEvent](#) whenever the content of your custom view changes. For example, if you are implementing a custom slider bar that allows a user to select a numeric value by pressing the left or right arrows, your custom view should emit an event of type [TYPE\\_VIEW\\_TEXT\\_CHANGED](#) whenever the slider value changes. The following sample code demonstrates the use of the [sendAccessibilityEvent\(\)](#) method to report this event.

```
@Override  
public boolean onKeyUp (int keyCode, KeyEvent event) {  
    if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {  
        mCurrentValue--;  
        sendAccessibilityEvent (AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED);  
        return true;  
    }  
    ...  
}
```

## Populating accessibility events

Each [AccessibilityEvent](#) has a set of required properties that describe the current state of the view. These properties include things such as the view's class name, content description and checked state. The specific properties required for each event type are described in the [AccessibilityEvent](#) reference documentation. The [View](#) implementation provides default values for these properties. Many of these values, including the class name and event timestamp, are provided automatically. If you are creating a custom view component, you must provide some information about the content and characteristics of the view. This information may be as simple as a button label, but may also include additional state information that you want to add to the event.

The minimum requirement for providing information to accessibility services with a custom view is to implement [dispatchPopulateAccessibilityEvent\(\)](#). This method is called by the system to request information for an [AccessibilityEvent](#) and makes your custom view compatible with accessibility services on Android 1.6 (API Level 4) and higher. The following example code demonstrates a basic implementation of this method.

```
@Override
public void dispatchPopulateAccessibilityEvent(AccessibilityEvent event) {
    super.dispatchPopulateAccessibilityEvent(event);
    // Call the super implementation to populate its text to the event, which
    // calls onPopulateAccessibilityEvent() on API Level 14 and up.

    // In case this is running on a API revision earlier than 14, check
    // the text content of the event and add an appropriate text
    // description for this custom view:
    CharSequence text = getText();
    if (!TextUtils.isEmpty(text)) {
        event.getText().add(text);
    }
}
```

For Android 4.0 (API Level 14) and higher, use the [onPopulateAccessibilityEvent\(\)](#) and [onInitializeAccessibilityEvent\(\)](#) methods to populate or modify the information in an [AccessibilityEvent](#). Use the [onPopulateAccessibilityEvent\(\)](#) method specifically for adding or modifying the text content of the event, which is turned into audible prompts by accessibility services such as TalkBack. Use the [onInitializeAccessibilityEvent\(\)](#) method for populating additional information about the event, such as the selection state of the view.

In addition, implement the [onInitializeAccessibilityNodeInfo\(\)](#) method. The [AccessibilityNodeInfo](#) objects populated by this method are used by accessibility services to investigate the view hierarchy that generated an accessibility event after receiving that event, to obtain a more detailed context information and provide appropriate feedback to users.

The example code below shows how override these three methods by using [ViewCompat.setAccessibilityDelegate\(\)](#). Note that this sample code requires that the Android [Support Library](#) for API Level 4 (revision 5 or higher) is added to your project.

```
ViewCompat.setAccessibilityDelegate(new AccessibilityDelegateCompat() {
    @Override
    public void onPopulateAccessibilityEvent(View host, AccessibilityEvent event) {
        super.onPopulateAccessibilityEvent(host, event);
        // We call the super implementation to populate its text for the
        // event. Then we add our text not present in a super class.
        // Very often you only need to add the text for the custom view.
        CharSequence text = getText();
        if (!TextUtils.isEmpty(text)) {
            event.getText().add(text);
        }
    }
    @Override
    public void onInitializeAccessibilityEvent(View host, AccessibilityEvent event) {
        super.onInitializeAccessibilityEvent(host, event);
        // We call the super implementation to let super classes
        // set appropriate event properties. Then we add the new property
    }
}
```

```

        // (checked) which is not supported by a super class.
        event.setChecked(isChecked());
    }

    @Override
    public void onInitializeAccessibilityNodeInfo(View host,
        AccessibilityNodeInfoCompat info) {
        super.onInitializeAccessibilityNodeInfo(host, info);
        // We call the super implementation to let super classes set
        // appropriate info properties. Then we add our properties
        // (checkable and checked) which are not supported by a super class.
        info.setCheckable(true);
        info.setChecked(isChecked());
        // Quite often you only need to add the text for the custom view.
        CharSequence text = getText();
        if (!TextUtils.isEmpty(text)) {
            info.setText(text);
        }
    }
}

```

In applications targeting Android 4.0 (API Level 14) and higher, you can implement these methods directly in your custom view class. For another example of this approach, see the Android [Support Library](#) (revision 5 or higher) sample `AccessibilityDelegateSupportActivity` in (`<sdk>/extras/android/support/v4/samples/Support4Demos/`).

**Note:** You may find information on implementing accessibility for custom views written prior to Android 4.0 that describes the use of the [`dispatchPopulateAccessibilityEvent\(\)`](#) method for populating `AccessibilityEvents`. As of the Android 4.0 release, however, the recommended approach is to use the [`onPopulateAccessibilityEvent\(\)`](#) and [`onInitializeAccessibilityEvent\(\)`](#) methods.

## Providing a customized accessibility context

In Android 4.0 (API Level 14), the framework was enhanced to allow accessibility services to inspect the containing view hierarchy of a user interface component that generates an accessibility event. This enhancement allows accessibility services to provide a much richer set of contextual information with which to aid users.

There are some cases where accessibility services cannot get adequate information from the view hierarchy. An example of this is a custom interface control that has two or more separately clickable areas, such as a calendar control. In this case, the services cannot get adequate information because the clickable subsections are not part of the view hierarchy.

custom_calendar_widget						
Month						
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

## Figure 1. A custom calendar view with selectable day elements.

In the example shown in Figure 1, the entire calendar is implemented as a single view, so if you do not do anything else, accessibility services do not receive enough information about the content of the view and the user's selection within the view. For example, if a user clicks on the day containing 17, the accessibility framework only receives the description information for the whole calendar control. In this case, the TalkBack accessibility service would simply announce "Calendar" or, only slightly better, "April Calendar" and the user would be left to wonder what day was selected.

To provide adequate context information for accessibility services in situations like this, the framework provides a way to specify a virtual view hierarchy. A *virtual view hierarchy* is a way for application developers to provide a complementary view hierarchy to accessibility services that more closely matches the actual information on screen. This approach allows accessibility services to provide more useful context information to users.

Another situation where a virtual view hierarchy may be needed is a user interface containing a set of controls (views) that have closely related functions, where an action on one control affects the contents of one or more elements, such as a number picker with separate up and down buttons. In this case, accessibility services cannot get adequate information because action on one control changes content in another and the relationship of those controls may not be apparent to the service. To handle this situation, group the related controls with a containing view and provide a virtual view hierarchy from this container to clearly represent the information and behavior provided by the controls.

In order to provide a virtual view hierarchy for a view, override the [getAccessibilityNodeProvider\(\)](#) method in your custom view or view group and return an implementation of [AccessibilityNodeProvider](#). For an example implementation of this accessibility feature, see [AccessibilityNodeProviderActivity](#) in the ApiDemos sample project. You can implement a virtual view hierarchy that is compatible with Android 1.6 and later by using the [Support Library](#) with the [ViewCompat.getAccessibilityNodeProvider\(\)](#) method and providing an implementation with [AccessibilityNodeProviderCompat](#).

## Handling custom touch events

Custom view controls may require non-standard touch event behavior. For example, a custom control may use the [onTouchEvent\(MotionEvent\)](#) listener method to detect the [ACTION\\_DOWN](#) and [ACTION\\_UP](#) events and trigger a special click event. In order to maintain compatibility with accessibility services, the code that handles this custom click event must do the following:

1. Generate an appropriate [AccessibilityEvent](#) for the interpreted click action.
2. Enable accessibility services to perform the custom click action for users who are not able to use a touch screen.

To handle these requirements in an efficient way, your code should override the [performClick\(\)](#) method, which must call the super implementation of this method and then execute whatever actions are required by the click event. When the custom click action is detected, that code should then call your [performClick\(\)](#) method. The following code example demonstrates this pattern.

```
class CustomTouchView extends View {  
  
    public CustomTouchView(Context context) {  
        super(context);  
    }  
  
    boolean mDownTouch = false;
```

```

@Override
public boolean onTouchEvent(MotionEvent event) {
    super.onTouchEvent(event);

    // Listening for the down and up touch events
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            mDownTouch = true;
            return true;

        case MotionEvent.ACTION_UP:
            if (mDownTouch) {
                mDownTouch = false;
                performClick(); // Call this method to handle the response,
                               // thereby enable accessibility services to
                               // perform this action for a user who cannot
                               // click the touchscreen.
                return true;
            }
    }
    return false; // Return false for other touch events
}

@Override
public boolean performClick() {
    // Calls the super implementation, which generates an AccessibilityEvent
    // and calls the onClick() listener on the view, if any
    super.performClick();

    // Handle the action for the custom click here

    return true;
}
}

```

The pattern shown above makes sure that the custom click event is compatible with accessibility services by using the [performClick\(\)](#) method to both generate an accessibility event and provide an entry point for accessibility services to act on behalf of a user to perform this custom click event.

**Note:** If your custom view has distinct clickable regions, such as a custom calendar view, you must implement a [virtual view hierarchy](#) by overriding [getAccessibilityNodeProvider\(\)](#) in your custom view in order to be compatible with accessibility services.

## Testing Accessibility

Testing the accessibility of your application is an important part of ensuring your users have a great experience. You can test the most important accessibility features by using your application with audible feedback enabled and navigating within your application using only directional controls. For more information on testing accessibility in your application, see the [Accessibility Testing Checklist](#).

# Accessibility Developer Checklist

## In this document

1. [Accessibility Requirements](#)
2. [Accessibility Recommendations](#)
3. [Special Cases and Considerations](#)

## See also

1. [Android Design: Accessibility](#)
2. [Accessibility Testing Checklist](#)
3. [Training: Implementing Accessibility](#)
4. [Designing Effective Navigation](#)

Making an application accessible is about a deep commitment to usability, getting the details right and delighting your users. This document provides a checklist of accessibility requirements, recommendations and considerations to help you make sure your application is accessible. Following this checklist does not guarantee your application is accessible, but it's a good place to start.

Creating an accessible application is not just the responsibility of developers. Involve your design and testing folks as well, and make them aware of the guidelines for these other stages of development:

- [Android Design: Accessibility](#)
- [Accessibility Testing Checklist](#)

In most cases, creating an accessible Android application does not require extensive code restructuring. Rather, it means working through the subtle details of how users interact with your application, so you can provide them with feedback they can sense and understand. This checklist helps you focus on the key development issues to get the details of accessibility right.

## Accessibility Requirements

The following steps must be completed in order to ensure a minimum level of application accessibility.

1. **Describe user interface controls:** Provide content [descriptions](#) for user interface components that do not have visible text, particularly [ImageButton](#), [ImageView](#) and [CheckBox](#) components. Use the [android:contentDescription](#) XML layout attribute or the [setContentDescription\(CharSequence\)](#) method to provide this information for accessibility services. (Exception: [decorative graphics](#))
2. **Enable focus-based navigation:** Make sure [users can navigate](#) your screen layouts using hardware-based or software directional controls (D-pads, trackballs, keyboards and navigation gestures). In a few cases, you may need to make user interface components [focusable](#) or change the [focus order](#) to be more logical for user actions.
3. **Custom view controls:** If you build [custom interface controls](#) for your application, [implement accessibility interfaces](#) for your custom views and provide content descriptions. For custom controls that are intended to be compatible with versions of Android back to 1.6, use the [Support Library](#) to implement the latest accessibility features.
4. **No audio-only feedback:** Audio feedback must always have a secondary feedback mechanism to support users who are deaf or hard of hearing. For example, a sound alert for the arrival of a message must be accompanied by a system [Notification](#), haptic feedback (if available) or other visual alert.

5. **Test:** Test accessibility by navigating your application using directional controls, and using eyes-free navigation with TalkBack enabled. For more accessibility testing information, see the [Accessibility Testing Checklist](#).

## Accessibility Recommendations

The following steps are recommended for ensuring the accessibility of your application. If you do not take these actions, it may impact the overall accessibility and quality of your application.

1. **Android Design Accessibility Guidelines:** Before building your layouts, review and follow the accessibility guidelines provided in the [Design guidelines](#).
2. **Framework-provided controls:** Use Android's built-in user interface controls whenever possible, as these components provide accessibility support by default.
3. **Temporary or self-hiding controls and notifications:** Avoid having user interface controls that fade out or disappear after a certain amount of time. If this behavior is important to your application, provide an alternative interface for these functions.

## Special Cases and Considerations

The following list describes specific situations where action should be taken to ensure an accessible app. Review this list to see if any of these special cases and considerations apply to your application, and take the appropriate action.

1. **Text field hints:** For [EditText](#) fields, provide an [android:hint](#) attribute *instead* of a content description, to help users understand what content is expected when the text field is empty and allow the contents of the field to be spoken when it is filled.
2. **Custom controls with high visual context:** If your application contains a [custom control](#) with a high degree of visual context (such as a calendar control), default accessibility services processing may not provide adequate descriptions for users, and you should consider providing a [virtual view hierarchy](#) for your control using [AccessibilityNodeProvider](#).
3. **Custom controls and click handling:** If a custom control in your application performs specific handling of user touch interaction, such as listening with [onTouchEvent \(MotionEvent\)](#) for [MotionEvent.ACTION\\_DOWN](#) and [MotionEvent.ACTION\\_UP](#) and treating it as a click event, you must trigger an [AccessibilityEvent](#) equivalent to a click and provide a way for accessibility services to perform this action for users. For more information, see [Handling custom touch events](#).
4. **Controls that change function:** If you have buttons or other controls that change function during the normal activity of a user in your application (for example, a button that changes from **Play** to **Pause**), make sure you also change the [android:contentDescription](#) of the button appropriately.
5. **Prompts for related controls:** Make sure sets of controls which provide a single function, such as the [DatePicker](#), provide useful audio feedback when an user interacts with the individual controls.
6. **Video playback and captioning:** If your application provides video playback, it must support captioning and subtitles to assist users who are deaf or hard of hearing. Your video playback controls must also clearly indicate if captioning is available for a video and provide a clear way of enabling captions.
7. **Supplemental accessibility audio feedback:** Use only the Android accessibility framework to provide accessibility audio feedback for your app. Accessibility services such as [TalkBack](#) should be the only way your application provides accessibility audio prompts to users. Provide the prompting information with a [android:contentDescription](#) XML layout attribute or dynamically add it using accessibility framework APIs. For example, if your application takes action that you want to announce to a user, such as automatically turning the page of a book, use the [announceForAccessibility \(CharSequence\)](#) method to have accessibility services speak this information to the user.
8. **Custom controls with complex visual interactions:** For custom controls that provide complex or non-standard visual interactions, provide a [virtual view hierarchy](#) for your control using [Accessibili-](#)

[tyNodeProvider](#) that allows accessibility services to provide a simplified interaction model for the user. If this approach is not feasible, consider providing an alternate view that is accessible.

9. **Sets of small controls:** If you have controls that are smaller than the minimum recommended touch size in your application screens, consider grouping these controls together using a [ViewGroup](#) and providing a [android:contentDescription](#) for the group.
10. **Decorative images and graphics:** Elements in application screens that are purely decorative and do not provide any content or enable a user action should not have accessibility content descriptions.

# Building Accessibility Services

## Topics

1. [Manifest Declarations and Permissions](#)
  1. [Accessibility service declaration](#)
  2. [Accessibility service configuration](#)
2. [Registering for Accessibility Events](#)
3. [AccessibilityService Methods](#)
4. [Getting Event Details](#)
5. [Taking Action for Users](#)
  1. [Listening for gestures](#)
  2. [Using accessibility actions](#)
  3. [Using focus types](#)
6. [Example Code](#)

## Key classes

1. [AccessibilityService](#)
2. [AccessibilityServiceInfo](#)
3. [AccessibilityEvent](#)
4. [AccessibilityRecord](#)
5. [AccessibilityNodeInfo](#)

## See also

1. [Training: Implementing Accessibility](#)

An accessibility service is an application that provides user interface enhancements to assist users with disabilities, or who may temporarily be unable to fully interact with a device. For example, users who are driving, taking care of a young child or attending a very loud party might need additional or alternative interface feedback.

Android provides standard accessibility services, including TalkBack, and developers can create and distribute their own services. This document explains the basics of building an accessibility service.

The ability for you to build and deploy accessibility services was introduced with Android 1.6 (API Level 4) and received significant improvements with Android 4.0 (API Level 14). The Android [Support Library](#) was also updated with the release of Android 4.0 to provide support for these enhanced accessibility features back to Android 1.6. Developers aiming for widely compatible accessibility services are encouraged to use the Support Library and develop for the more advanced accessibility features introduced in Android 4.0.

## Manifest Declarations and Permissions

Applications that provide accessibility services must include specific declarations in their application manifests to be treated as an accessibility service by the Android system. This section explains the required and optional settings for accessibility services.

## Accessibility service declaration

In order to be treated as an accessibility service, your application must include the `service` element (rather than the `activity` element) within the `application` element in its manifest. In addition, within the `service` element, you must also include an accessibility service intent filter. For compatibility with Android 4.1 and higher, the manifest must also request the [BIND\\_ACCESSIBILITY\\_SERVICE](#) permission as shown in the following sample:

```
<application>
    <service android:name=".MyAccessibilityService"
        android:label="@string/accessibility_service_label">
        <intent-filter>
            <action android:name="android.accessibilityservice.AccessibilityService" />
        </intent-filter>
    </service>
    <uses-permission android:name="android.permission.BIND_ACCESSIBILITY_SERVICE" />
</application>
```

These declarations are required for all accessibility services deployed on Android 1.6 (API Level 4) or higher.

## Accessibility service configuration

Accessibility services must also provide a configuration which specifies the types of accessibility events that the service handles and additional information about the service. The configuration of an accessibility service is contained in the [AccessibilityServiceInfo](#) class. Your service can build and set a configuration using an instance of this class and [setServiceInfo\(\)](#) at runtime. However, not all configuration options are available using this method.

Beginning with Android 4.0, you can include a `<meta-data>` element in your manifest with a reference to a configuration file, which allows you to set the full range of options for your accessibility service, as shown in the following example:

```
<service android:name=".MyAccessibilityService">
    ...
    <meta-data
        android:name="android.accessibilityservice"
        android:resource="@xml/accessibility_service_config" />
</service>
```

This meta-data element refers to an XML file that you create in your application's resource directory (`<project_dir>/res/xml/accessibility_service_config.xml`). The following code shows example contents for the service configuration file:

```
<accessibility-service xmlns:android="http://schemas.android.com/apk/res/android"
    android:description="@string/accessibility_service_description"
    android:packageName="com.example.android.apis"
    android:accessibilityEventTypes="typeAllMask"
    android:accessibilityFlags="flagDefault"
    android:accessibilityFeedbackType="feedbackSpoken"
    android:notificationTimeout="100"
    android:canRetrieveWindowContent="true"
    android:settingsActivity="com.example.android.accessibility.ServiceSettings"
/>
```

For more information about the XML attributes which can be used in the accessibility service configuration file, follow these links to the reference documentation:

- [android:description](#)
- [android:packageName](#)s
- [android:accessibilityEventTypes](#)
- [android:accessibilityFlags](#)
- [android:accessibilityFeedbackType](#)
- [android:notificationTimeout](#)
- [android:canRetrieveWindowContent](#)
- [android:settingsActivity](#)

For more information about which configuration settings can be dynamically set at runtime, see the [AccessibilityServiceInfo](#) reference documentation.

## Registering for Accessibility Events

One of the most important functions of the accessibility service configuration parameters is to allow you to specify what types of accessibility events your service can handle. Being able to specify this information enables accessibility services to cooperate with each other, and allows you as a developer the flexibility to handle only specific events types from specific applications. The event filtering can include the following criteria:

- **Package Names** - Specify the package names of applications whose accessibility events you want your service to handle. If this parameter is omitted, your accessibility service is considered available to service accessibility events for any application. This parameter can be set in the accessibility service configuration files with the `android:packageName` attribute as a comma-separated list, or set using the [AccessibilityServiceInfo.packageNames](#) member.
- **Event Types** - Specify the types of accessibility events you want your service to handle. This parameter can be set in the accessibility service configuration files with the `android:accessibilityEventTypes` attribute as a list separated by the `|` character (for example `accessibilityEventTypes="typeViewClicked|typeViewFocused"`), or set using the [AccessibilityServiceInfo.eventTypes](#) member.

When setting up your accessibility service, carefully consider what events your service is able to handle and only register for those events. Since users can activate more than one accessibility services at a time, your service must not consume events that it is not able to handle. Remember that other services may handle those events in order to improve a user's experience.

**Note:** The Android framework dispatches accessibility events to more than one accessibility service if the services provide different [feedback types](#). However, if two or more services provide the same feedback type, then only the first registered service receives the event.

## AccessibilityService Methods

An accessibility service must extend the [AccessibilityService](#) class and override the following methods from that class. These methods are presented in the order in which they are called by the Android system, from when the service is started ([onServiceConnected\(\)](#)), while it is running ([onAccessibilityEvent\(\)](#), [onInterrupt\(\)](#)) to when it is shut down ([onUnbind\(\)](#)).

- [onServiceConnected\(\)](#) - (optional) This system calls this method when it successfully connects to your accessibility service. Use this method to do any one-time setup steps for your service, including connecting to user feedback system services, such as the audio manager or device vibrator. If you want

to set the configuration of your service at runtime or make one-time adjustments, this is a convenient location from which to call [setServiceInfo\(\)](#).

- [onAccessibilityEvent\(\)](#) - (required) This method is called back by the system when it detects an [AccessibilityEvent](#) that matches the event filtering parameters specified by your accessibility service. For example, when the user clicks a button or focuses on a user interface control in an application for which your accessibility service is providing feedback. When this happens, the system calls this method, passing the associated [AccessibilityEvent](#), which the service can then interpret and use to provide feedback to the user. This method may be called many times over the lifecycle of your service.
- [onInterrupt\(\)](#) - (required) This method is called when the system wants to interrupt the feedback your service is providing, usually in response to a user action such as moving focus to a different control. This method may be called many times over the lifecycle of your service.
- [onUnbind\(\)](#) - (optional) This method is called when the system is about to shutdown the accessibility service. Use this method to do any one-time shutdown procedures, including de-allocating user feedback system services, such as the audio manager or device vibrator.

These callback methods provide the basic structure for your accessibility service. It is up to you to decide on how to process data provided by the Android system in the form of [AccessibilityEvent](#) objects and provide feedback to the user. For more information about getting information from an accessibility event, see the [Implementing Accessibility](#) training.

## Getting Event Details

The Android system provides information to accessibility services about the user interface interaction through [AccessibilityEvent](#) objects. Prior to Android 4.0, the information available in an accessibility event, while providing a significant amount of detail about a user interface control selected by the user, offered limited contextual information. In many cases, this missing context information might be critical to understanding the meaning of the selected control.

An example of an interface where context is critical is a calendar or day planner. If the user selects a 4:00 PM time slot in a Monday to Friday day list and the accessibility service announces “4 PM”, but does not announce the weekday name, the day of the month, or the month name, the resulting feedback is confusing. In this case, the context of a user interface control is critical to a user who wants to schedule a meeting.

Android 4.0 significantly extends the amount of information that an accessibility service can obtain about an user interface interaction by composing accessibility events based on the view hierarchy. A view hierarchy is the set of user interface components that contain the component (its parents) and the user interface elements that may be contained by that component (its children). In this way, the Android system can provide much richer detail about accessibility events, allowing accessibility services to provide more useful feedback to users.

An accessibility service gets information about an user interface event through an [AccessibilityEvent](#) passed by the system to the service’s [onAccessibilityEvent\(\)](#) callback method. This object provides details about the event, including the type of object being acted upon, its descriptive text and other details. Starting in Android 4.0 (and supported in previous releases through the [AccessibilityEventCompat](#) object in the Support Library), you can obtain additional information about the event using these calls:

- [AccessibilityEvent.getRecordCount\(\)](#) and [getRecord\(int\)](#) - These methods allow you to retrieve the set of [AccessibilityRecord](#) objects which contributed to the [AccessibilityEvent](#) passed to you by the system. This level of detail provides more context for the event that triggered your accessibility service.
- [AccessibilityEvent.getSource\(\)](#) - This method returns an [AccessibilityNodeInfo](#) object. This object allows you to request view layout hierarchy (parents and children) of the component

that originated the accessibility event. This feature allows an accessibility service to investigate the full context of an event, including the content and state of any enclosing views or child views.

**Important:** The ability to investigate the view hierarchy from an [AccessibilityEvent](#) potentially exposes private user information to your accessibility service. For this reason, your service must request this level of access through the accessibility [service configuration XML](#) file, by including the `canRetrieveWindowContent` attribute and setting it to `true`. If you do not include this setting in your service configuration xml file, calls to [getSource\(\)](#) fail.

**Note:** In Android 4.1 (API Level 16) and higher, the [getSource\(\)](#) method, as well as [AccessibilityNodeInfo.getChild\(\)](#) and [getParent\(\)](#), return only view objects that are considered important for accessibility (views that draw content or respond to user actions). If your service requires all views, it can request them by setting the `flags` member of the service's [AccessibilityServiceInfo](#) instance to [FLAG\\_INCLUDE\\_NOT\\_IMPORTANT\\_VIEWS](#).

## Taking Action for Users

Starting with Android 4.0 (API Level 14), accessibility services can act on behalf of users, including changing the input focus and selecting (activating) user interface elements. In Android 4.1 (API Level 16) the range of actions has been expanded to include scrolling lists and interacting with text fields. Accessibility services can also take global actions, such as navigating to the Home screen, pressing the Back button, opening the notifications screen and recent applications list. Android 4.1 also includes a new type of focus, *Accessibility Focus*, which makes all visible elements selectable by an accessibility service.

These new capabilities make it possible for developers of accessibility services to create alternative navigation modes such as [gesture navigation](#), and give users with disabilities improved control of their Android devices.

## Listening for gestures

Accessibility services can listen for specific gestures and respond by taking action on behalf of a user. This feature, added in Android 4.1 (API Level 16), and requires that your accessibility service request activation of the Explore by Touch feature. Your service can request this activation by setting the `flags` member of the service's [AccessibilityServiceInfo](#) instance to [FLAG\\_REQUEST\\_TOUCH\\_EXPLORATION\\_MODE](#), as shown in the following example.

```
public class MyAccessibilityService extends AccessibilityService {  
    @Override  
    public void onCreate() {  
        getServiceInfo().flags = AccessibilityServiceInfo.FLAG_REQUEST_TOUCH_EXPLORATION_MODE;  
    }  
    ...  
}
```

Once your service has requested activation of Explore by Touch, the user must allow the feature to be turned on, if it is not already active. When this feature is active, your service receives notification of accessibility gestures through your service's [onGesture\(\)](#) callback method and can respond by taking actions for the user.

## Using accessibility actions

Accessibility services can take action on behalf of users to make interacting with applications simpler and more productive. The ability of accessibility services to perform actions was added in Android 4.0 (API Level 14) and significantly expanded with Android 4.1 (API Level 16).

In order to take actions on behalf of users, your accessibility service must [register](#) to receive events from a few or many applications and request permission to view the content of applications by setting the [an-droid:canRetrieveWindowContent](#) to true in the [service configuration file](#). When events are received by your service, it can then retrieve the [AccessibilityNodeInfo](#) object from the event using [getSource\(\)](#). With the [AccessibilityNodeInfo](#) object, your service can then explore the view hierarchy to determine what action to take and then act for the user using [performAction\(\)](#).

```
public class MyAccessibilityService extends AccessibilityService {  
  
    @Override  
    public void onAccessibilityEvent(AccessibilityEvent event) {  
        // get the source node of the event  
        AccessibilityNodeInfo nodeInfo = event.getSource();  
  
        // Use the event and node information to determine  
        // what action to take  
  
        // take action on behalf of the user  
        nodeInfo.performAction(AccessibilityNodeInfo.ACTION_SCROLL_FORWARD);  
  
        // recycle the nodeInfo object  
        nodeInfo.recycle();  
    }  
    ...  
}
```

The [performAction\(\)](#) method allows your service to take action within an application. If your service needs to perform a global action such as navigating to the Home screen, pressing the Back button, opening the notifications screen or recent applications list, then use the [performGlobalAction\(\)](#) method.

## Using focus types

Android 4.1 (API Level 16) introduces a new type of user interface focus called *Accessibility Focus*. This type of focus can be used by accessibility services to select any visible user interface element and act on it. This focus type is different from the more well known *Input Focus*, which determines what on-screen user interface element receives input when a user types characters, presses **Enter** on a keyboard or pushes the center button of a D-pad control.

Accessibility Focus is completely separate and independent from Input Focus. In fact, it is possible for one element in a user interface to have Input Focus while another element has Accessibility Focus. The purpose of Accessibility Focus is to provide accessibility services with a method of interacting with any visible element on a screen, regardless of whether or not the element is input-focusable from a system perspective. You can see accessibility focus in action by testing accessibility gestures. For more information about testing this feature, see [Testing gesture navigation](#).

**Note:** Accessibility services that use Accessibility Focus are responsible for synchronizing the current Input Focus when an element is capable of this type of focus. Services that do not synchronize Input Focus with Accessibility Focus run the risk of causing problems in applications that expect input focus to be in a specific location when certain actions are taken.

An accessibility service can determine what user interface element has Input Focus or Accessibility Focus using the [AccessibilityNodeInfo.findFocus\(\)](#) method. You can also search for elements that can be selected with Input Focus using the [focusSearch\(\)](#) method. Finally, your accessibility service can set Acces-

sibility Focus using the [`performAction\(AccessibilityNodeInfo.ACTION\_SET\_ACCESSIBILITY\_FOCUS\)`](#) method.

## Example Code

The API Demo project contains two samples which can be used as a starting point for generating accessibility services (<sdk>/samples/<platform>/ApiDemos/src/com/example/android/apis/accessibility):

- [\*\*ClockBackService\*\*](#) - This service is based on the original implementation of [\*\*AccessibilityService\*\*](#) and can be used as a base for developing basic accessibility services that are compatible with Android 1.6 (API Level 4) and higher.
- [\*\*TaskBackService\*\*](#) - This service is based on the enhanced accessibility APIs introduced in Android 4.0 (API Level 14). However, you can use the Android [\*\*Support Library\*\*](#) to substitute classes introduced in later API levels (e.g., [\*\*AccessibilityRecord\*\*](#), [\*\*AccessibilityNodeInfo\*\*](#)) with equivalent support package classes (e.g., [\*\*AccessibilityRecordCompat\*\*](#), [\*\*AccessibilityNodeInfoCompat\*\*](#)) to make this example work with API versions back to Android 1.6 (API Level 4).

# Styles and Themes

## In this document

1. [Defining Styles](#)
  1. [Inheritance](#)
  2. [Style Properties](#)
2. [Applying Styles and Themes to the UI](#)
  1. [Apply a style to a View](#)
  2. [Apply a theme to an Activity or application](#)
  3. [Select a theme based on platform version](#)
3. [Using Platform Styles and Themes](#)

## See also

1. [Style and Theme Resources](#)
2. [R.style](#) for Android styles and themes
3. [R.attr](#) for all style attributes

A **style** is a collection of properties that specify the look and format for a [View](#) or window. A style can specify properties such as height, padding, font color, font size, background color, and much more. A style is defined in an XML resource that is separate from the XML that specifies the layout.

Styles in Android share a similar philosophy to cascading stylesheets in web design—they allow you to separate the design from the content.

For example, by using a style, you can take this layout XML:

```
<TextView  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:textColor="#00FF00"  
    android:typeface="monospace"  
    android:text="@string/hello" />
```

And turn it into this:

```
<TextView  
    style="@style/CodeFont"  
    android:text="@string/hello" />
```

All of the attributes related to style have been removed from the layout XML and put into a style definition called `CodeFont`, which is then applied with the `style` attribute. You'll see the definition for this style in the following section.

A **theme** is a style applied to an entire [Activity](#) or application, rather than an individual [View](#) (as in the example above). When a style is applied as a theme, every View in the Activity or application will apply each style property that it supports. For example, you can apply the same `CodeFont` style as a theme for an Activity and then all text inside that Activity will have green monospace font.

# Defining Styles

To create a set of styles, save an XML file in the `res/values/` directory of your project. The name of the XML file is arbitrary, but it must use the `.xml` extension and be saved in the `res/values/` folder.

The root node of the XML file must be `<resources>`.

For each style you want to create, add a `<style>` element to the file with a `name` that uniquely identifies the style (this attribute is required). Then add an `<item>` element for each property of that style, with a `name` that declares the style property and a value to go with it (this attribute is required). The value for the `<item>` can be a keyword string, a hex color, a reference to another resource type, or other value depending on the style property. Here's an example file with a single style:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CodeFont" parent="@android:style/TextAppearance.Medium">
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

Each child of the `<resources>` element is converted into an application resource object at compile-time, which can be referenced by the value in the `<style>` element's `name` attribute. This example style can be referenced from an XML layout as `@style/CodeFont` (as demonstrated in the introduction above).

The `parent` attribute in the `<style>` element is optional and specifies the resource ID of another style from which this style should inherit properties. You can then override the inherited style properties if you want to.

Remember, a style that you want to use as an Activity or application theme is defined in XML exactly the same as a style for a View. A style such as the one defined above can be applied as a style for a single View or as a theme for an entire Activity or application. How to apply a style for a single View or as an application theme is discussed later.

## Inheritance

The `parent` attribute in the `<style>` element lets you specify a style from which your style should inherit properties. You can use this to inherit properties from an existing style and then define only the properties that you want to change or add. You can inherit from styles that you've created yourself or from styles that are built into the platform. (See [Using Platform Styles and Themes](#), below, for information about inheriting from styles defined by the Android platform.) For example, you can inherit the Android platform's default text appearance and then modify it:

```
<style name="GreenText" parent="@android:style/TextAppearance">
    <item name="android:textColor">#00FF00</item>
</style>
```

If you want to inherit from styles that you've defined yourself, you *do not* have to use the `parent` attribute. Instead, just prefix the name of the style you want to inherit to the name of your new style, separated by a period. For example, to create a new style that inherits the `CodeFont` style defined above, but make the color red, you can author the new style like this:

```
<style name="CodeFont.Red">
    <item name="android:textColor">#FF0000</item>
</style>
```

Notice that there is no `parent` attribute in the `<style>` tag, but because the `name` attribute begins with the `CodeFont` style name (which is a style that you have created), this style inherits all style properties from that style. This style then overrides the `android:textColor` property to make the text red. You can reference this new style as `@style/CodeFont.Red`.

You can continue inheriting like this as many times as you'd like, by chaining names with periods. For example, you can extend `CodeFont.Red` to be bigger, with:

```
<style name="CodeFont.Red.Big">
    <item name="android:textSize">30sp</item>
</style>
```

This inherits from both `CodeFont` and `CodeFont.Red` styles, then adds the `android:textSize` property.

**Note:** This technique for inheritance by chaining together names only works for styles defined by your own resources. You can't inherit Android built-in styles this way. To reference a built-in style, such as [TextAppearance](#), you must use the `parent` attribute.

## Style Properties

Now that you understand how a style is defined, you need to learn what kind of style properties—defined by the `<item>` element—are available. You're probably familiar with some already, such as [layout\\_width](#) and [textColor](#). Of course, there are many more style properties you can use.

The best place to find properties that apply to a specific [View](#) is the corresponding class reference, which lists all of the supported XML attributes. For example, all of the attributes listed in the table of [TextView XML attributes](#) can be used in a style definition for a [TextView](#) element (or one of its subclasses). One of the attributes listed in the reference is [android:inputType](#), so where you might normally place the [android:inputType](#) attribute in an `<EditText>` element, like this:

```
<EditText
    android:inputType="number"
    ... />
```

You can instead create a style for the [EditText](#) element that includes this property:

```
<style name="Numbers">
    <item name="android:inputType">number</item>
    ...
</style>
```

So your XML for the layout can now implement this style:

```
<EditText
    style="@style/Numbers"
    ... />
```

This simple example may look like more work, but when you add more style properties and factor-in the ability to re-use the style in various places, the pay-off can be huge.

For a reference of all available style properties, see the [R.attr](#) reference. Keep in mind that all View objects don't accept all the same style attributes, so you should normally refer to the specific [View](#) class for supported style properties. However, if you apply a style to a View that does not support all of the style properties, the View will apply only those properties that are supported and simply ignore the others.

Some style properties, however, are not supported by any View element and can only be applied as a theme. These style properties apply to the entire window and not to any type of View. For example, style properties for a theme can hide the application title, hide the status bar, or change the window's background. These kind of style properties do not belong to any View object. To discover these theme-only style properties, look at the [R.attr](#) reference for attributes that begin with `window`. For instance, `windowNoTitle` and `windowBackground` are style properties that are effective only when the style is applied as a theme to an Activity or application. See the next section for information about applying a style as a theme.

**Note:** Don't forget to prefix the property names in each `<item>` element with the `android:` namespace. For example: `<item name="android:inputType">`.

## Applying Styles and Themes to the UI

There are two ways to set a style:

- To an individual View, by adding the `style` attribute to a View element in the XML for your layout.
- Or, to an entire Activity or application, by adding the `android:theme` attribute to the `<activity>` or `<application>` element in the Android manifest.

When you apply a style to a single [View](#) in the layout, the properties defined by the style are applied only to that [View](#). If a style is applied to a [viewGroup](#), the child [View](#) elements will **not** inherit the style properties—only the element to which you directly apply the style will apply its properties. However, you *can* apply a style so that it applies to all [View](#) elements—by applying the style as a theme.

To apply a style definition as a theme, you must apply the style to an [Activity](#) or application in the Android manifest. When you do so, every [View](#) within the Activity or application will apply each property that it supports. For example, if you apply the `CodeFont` style from the previous examples to an Activity, then all View elements that support the text style properties will apply them. Any View that does not support the properties will ignore them. If a View supports only some of the properties, then it will apply only those properties.

### Apply a style to a View

Here's how to set a style for a View in the XML layout:

```
<TextView  
    style="@style/CodeFont"  
    android:text="@string/hello" />
```

Now this `TextView` will be styled as defined by the style named `CodeFont`. (See the sample above, in [Defining Styles](#).)

**Note:** The `style` attribute does *not* use the `android:` namespace prefix.

### Apply a theme to an Activity or application

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

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

If you want a theme applied to just one Activity in your application, then add the `android:theme` attribute to the `<activity>` tag instead.

Just as Android provides other built-in resources, there are many pre-defined themes that you can use, to avoid writing them yourself. For example, you can use the `Dialog` theme and make your Activity appear like a dialog box:

```
<activity android:theme="@android:style/Theme.Dialog">
```

Or if you want the background to be transparent, use the `Translucent` theme:

```
<activity android:theme="@android:style/Theme.Translucent">
```

If you like a theme, but want to tweak it, just add the theme as the parent of your custom theme. For example, you can modify the traditional light theme to use your own color like this:

```
<color name="custom_theme_color">#b0b0ff</color>
<style name="CustomTheme" parent="android:Theme.Light">
    <item name="android:windowBackground">@color/custom_theme_color</item>
    <item name="android:colorBackground">@color/custom_theme_color</item>
</style>
```

(Note that the color needs to supplied as a separate resource here because the `android:windowBackground` attribute only supports a reference to another resource; unlike `android:colorBackground`, it can not be given a color literal.)

Now use `CustomTheme` instead of `Theme.Light` inside the Android Manifest:

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

## Select a theme based on platform version

Newer versions of Android have additional themes available to applications, and you might want to use these while running on those platforms while still being compatible with older versions. You can accomplish this through a custom theme that uses resource selection to switch between different parent themes, based on the platform version.

For example, here is the declaration for a custom theme which is simply the standard platforms default light theme. It would go in an XML file under `res/values` (typically `res/values/styles.xml`):

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

To have this theme use the newer holographic theme when the application is running on Android 3.0 (API Level 11) or higher, you can place an alternative declaration for the theme in an XML file in `res/values-v11`, but make the parent theme the holographic theme:

```
<style name="LightThemeSelector" parent="android:Theme.Holo.Light">
    ...
</style>
```

Now use this theme like you would any other, and your application will automatically switch to the holographic theme if running on Android 3.0 or higher.

A list of the standard attributes that you can use in themes can be found at [R.styleable.Theme](#).

For more information about providing alternative resources, such as themes and layouts, based on the platform version or other device configurations, see the [Providing Resources](#) document.

## Using Platform Styles and Themes

The Android platform provides a large collection of styles and themes that you can use in your applications. You can find a reference of all available styles in the [R.style](#) class. To use the styles listed here, replace all underscores in the style name with a period. For example, you can apply the [Theme\\_NoTitleBar](#) theme with "@android:style/Theme.NoTitleBar".

The [R.style](#) reference, however, is not well documented and does not thoroughly describe the styles, so viewing the actual source code for these styles and themes will give you a better understanding of what style properties each one provides. For a better reference to the Android styles and themes, see the following source code:

- [Android Styles \(styles.xml\)](#)
- [Android Themes \(themes.xml\)](#)

These files will help you learn through example. For instance, in the Android themes source code, you'll find a declaration for `<style name="Theme.Dialog">`. In this definition, you'll see all of the properties that are used to style dialogs that are used by the Android framework.

For more information about the syntax for styles and themes in XML, see the [Style Resource](#) document.

For a reference of available style attributes that you can use to define a style or theme (e.g., "windowBackground" or "textAppearance"), see [R.attr](#) or the respective View class for which you are creating a style.

# Custom Components

## In this document

1. [The Basic Approach](#)
2. [Fully Customized Components](#)
3. [Compound Controls](#)
4. [Modifying an Existing View Type](#)

Android offers a sophisticated and powerful componentized model for building your UI, based on the fundamental layout classes: [View](#) and [ViewGroup](#). To start with, the platform includes a variety of prebuilt View and ViewGroup subclasses — called widgets and layouts, respectively — that you can use to construct your UI.

A partial list of available widgets includes [Button](#), [TextView](#), [EditText](#), [ListView](#), [CheckBox](#), [RadioButton](#), [Gallery](#), [Spinner](#), and the more special-purpose [AutoCompleteTextView](#), [ImageSwitcher](#), and [TextSwitcher](#).

Among the layouts available are [LinearLayout](#), [FrameLayout](#), [RelativeLayout](#), and others. For more examples, see [Common Layout Objects](#).

If none of the prebuilt widgets or layouts meets your needs, you can create your own View subclass. If you only need to make small adjustments to an existing widget or layout, you can simply subclass the widget or layout and override its methods.

Creating your own View subclasses gives you precise control over the appearance and function of a screen element. To give an idea of the control you get with custom views, here are some examples of what you could do with them:

- You could create a completely custom-rendered View type, for example a "volume control" knob rendered using 2D graphics, and which resembles an analog electronic control.
- You could combine a group of View components into a new single component, perhaps to make something like a ComboBox (a combination of popup list and free entry text field), a dual-pane selector control (a left and right pane with a list in each where you can re-assign which item is in which list), and so on.
- You could override the way that an EditText component is rendered on the screen (the [Notepad Tutorial](#) uses this to good effect, to create a lined-notepad page).
- You could capture other events like key presses and handle them in some custom way (such as for a game).

The sections below explain how to create custom Views and use them in your application. For detailed reference information, see the [View](#) class.

## The Basic Approach

Here is a high level overview of what you need to know to get started in creating your own View components:

1. Extend an existing [View](#) class or subclass with your own class.
2. Override some of the methods from the superclass. The superclass methods to override start with 'on', for example, [onDraw\(\)](#), [onMeasure\(\)](#), and [onKeyDown\(\)](#). This is similar to the `on...` events in [Activity](#) or [ListActivity](#) that you override for lifecycle and other functionality hooks.
3. Use your new extension class. Once completed, your new extension class can be used in place of the view upon which it was based.

**Tip:** Extension classes can be defined as inner classes inside the activities that use them. This is useful because it controls access to them but isn't necessary (perhaps you want to create a new public View for wider use in your application).

## Fully Customized Components

Fully customized components can be used to create graphical components that appear however you wish. Perhaps a graphical VU meter that looks like an old analog gauge, or a sing-a-long text view where a bouncing ball moves along the words so you can sing along with a karaoke machine. Either way, you want something that the built-in components just won't do, no matter how you combine them.

Fortunately, you can easily create components that look and behave in any way you like, limited perhaps only by your imagination, the size of the screen, and the available processing power (remember that ultimately your application might have to run on something with significantly less power than your desktop workstation).

To create a fully customized component:

1. The most generic view you can extend is, unsurprisingly, [View](#), so you will usually start by extending this to create your new super component.
2. You can supply a constructor which can take attributes and parameters from the XML, and you can also consume your own such attributes and parameters (perhaps the color and range of the VU meter, or the width and damping of the needle, etc.)
3. You will probably want to create your own event listeners, property accessors and modifiers, and possibly more sophisticated behavior in your component class as well.
4. You will almost certainly want to override `onMeasure()` and are also likely to need to override `onDraw()` if you want the component to show something. While both have default behavior, the default `onDraw()` will do nothing, and the default `onMeasure()` will always set a size of 100x100 — which is probably not what you want.
5. Other `on...` methods may also be overridden as required.

### Extend `onDraw()` and `onMeasure()`

The `onDraw()` method delivers you a [Canvas](#) upon which you can implement anything you want: 2D graphics, other standard or custom components, styled text, or anything else you can think of.

**Note:** This does not apply to 3D graphics. If you want to use 3D graphics, you must extend [SurfaceView](#) instead of `View`, and draw from a separate thread. See the `GLSurfaceViewActivity` sample for details.

`onMeasure()` is a little more involved. `onMeasure()` is a critical piece of the rendering contract between your component and its container. `onMeasure()` should be overridden to efficiently and accurately report the measurements of its contained parts. This is made slightly more complex by the requirements of limits from the parent (which are passed in to the `onMeasure()` method) and by the requirement to call the `setMeasuredDimension()` method with the measured width and height once they have been calculated. If you fail to call this method from an overridden `onMeasure()` method, the result will be an exception at measurement time.

At a high level, implementing `onMeasure()` looks something like this:

1. The overridden `onMeasure()` method is called with width and height measure specifications (`widthMeasureSpec` and `heightMeasureSpec` parameters, both are integer codes representing dimensions) which should be treated as requirements for the restrictions on the width and height measurements you should produce. A full reference to the kind of restrictions these specifications can re-

quire can be found in the reference documentation under [View.onMeasure\(int, int\)](#) (this reference documentation does a pretty good job of explaining the whole measurement operation as well).

2. Your component's `onMeasure()` method should calculate a measurement width and height which will be required to render the component. It should try to stay within the specifications passed in, although it can choose to exceed them (in this case, the parent can choose what to do, including clipping, scrolling, throwing an exception, or asking the `onMeasure()` to try again, perhaps with different measurement specifications).
3. Once the width and height are calculated, the `setMeasuredDimension(int width, int height)` method must be called with the calculated measurements. Failure to do this will result in an exception being thrown.

Here's a summary of some of the other standard methods that the framework calls on views:

Category	Methods	Description
Creation	Constructors	There is a form of the constructor that are called when the view is created from code and a form that is called when the view is inflated from a layout file. The second form should parse and apply any attributes defined in the layout file.
	<a href="#">onFinishInflate()</a>	Called after a view and all of its children has been inflated from XML.
Layout	<a href="#">onMeasure(int, int)</a>	Called to determine the size requirements for this view and all of its children.
	<a href="#">onLayout(boolean, int, int, int, int)</a>	Called when this view should assign a size and position to all of its children.
	<a href="#">onSizeChanged(int, int, int, int)</a>	Called when the size of this view has changed.
Drawing	<a href="#">onDraw(Canvas)</a>	Called when the view should render its content.
Event processing	<a href="#">onKeyDown(int, KeyEvent)</a>	Called when a new key event occurs.
	<a href="#">onKeyUp(int, KeyEvent)</a>	Called when a key up event occurs.
	<a href="#">onTrackballEvent(MotionEvent)</a>	Called when a trackball motion event occurs.
	<a href="#">onTouchEvent(MotionEvent)</a>	Called when a touch screen motion event occurs.
Focus	<a href="#">onFocusChanged(boolean, int, Rect)</a>	Called when the view gains or loses focus.

Category	Methods	Description
Attaching	<a href="#">onWindowFocusChanged (boolean)</a>	Called when the window containing the view gains or loses focus.
	<a href="#">onAttachedToWindow ()</a>	Called when the view is attached to a window.
	<a href="#">onDetachedFromWindow ()</a>	Called when the view is detached from its window.
	<a href="#">onWindowVisibilityChanged (int)</a>	Called when the visibility of the window containing the view has changed.

## A Custom View Example

The CustomView sample in the [API Demos](#) provides an example of a customized View. The custom View is defined in the [LabelView](#) class.

The LabelView sample demonstrates a number of different aspects of custom components:

- Extending the View class for a completely custom component.
- Parameterized constructor that takes the view inflation parameters (parameters defined in the XML). Some of these are passed through to the View superclass, but more importantly, there are some custom attributes defined and used for LabelView.
- Standard public methods of the type you would expect to see for a label component, for example `setText()`, `setTextSize()`, `setTextColor()` and so on.
- An overridden `onMeasure` method to determine and set the rendering size of the component. (Note that in LabelView, the real work is done by a private `measureWidth()` method.)
- An overridden `onDraw()` method to draw the label onto the provided canvas.

You can see some sample usages of the LabelView custom View in [custom\\_view\\_1.xml](#) from the samples. In particular, you can see a mix of both `android:` namespace parameters and custom `app:` namespace parameters. These `app:` parameters are the custom ones that the LabelView recognizes and works with, and are defined in a styleable inner class inside of the samples R resources definition class.

## Compound Controls

If you don't want to create a completely customized component, but instead are looking to put together a reusable component that consists of a group of existing controls, then creating a Compound Component (or Compound Control) might fit the bill. In a nutshell, this brings together a number of more atomic controls (or views) into a logical group of items that can be treated as a single thing. For example, a Combo Box can be thought of as a combination of a single line EditText field and an adjacent button with an attached PopupList. If you press the button and select something from the list, it populates the EditText field, but the user can also type something directly into the EditText if they prefer.

In Android, there are actually two other Views readily available to do this: [Spinner](#) and [AutoCompleteTextView](#), but regardless, the concept of a Combo Box makes an easy-to-understand example.

To create a compound component:

1. The usual starting point is a Layout of some kind, so create a class that extends a Layout. Perhaps in the case of a Combo box we might use a LinearLayout with horizontal orientation. Remember that other layouts can be nested inside, so the compound component can be arbitrarily complex and structured. Note that just like with an Activity, you can use either the declarative (XML-based) approach to creating the contained components, or you can nest them programmatically from your code.
2. In the constructor for the new class, take whatever parameters the superclass expects, and pass them through to the superclass constructor first. Then you can set up the other views to use within your new component; this is where you would create the EditText field and the PopupList. Note that you also might introduce your own attributes and parameters into the XML that can be pulled out and used by your constructor.
3. You can also create listeners for events that your contained views might generate, for example, a listener method for the List Item Click Listener to update the contents of the EditText if a list selection is made.
4. You might also create your own properties with accessors and modifiers, for example, allow the EditText value to be set initially in the component and query for its contents when needed.
5. In the case of extending a Layout, you don't need to override the `onDraw()` and `onMeasure()` methods since the layout will have default behavior that will likely work just fine. However, you can still override them if you need to.
6. You might override other `on...` methods, like `onKeyDown()`, to perhaps choose certain default values from the popup list of a combo box when a certain key is pressed.

To summarize, the use of a Layout as the basis for a Custom Control has a number of advantages, including:

- You can specify the layout using the declarative XML files just like with an activity screen, or you can create views programmatically and nest them into the layout from your code.
- The `onDraw()` and `onMeasure()` methods (plus most of the other `on...` methods) will likely have suitable behavior so you don't have to override them.
- In the end, you can very quickly construct arbitrarily complex compound views and re-use them as if they were a single component.

## Examples of Compound Controls

In the API Demos project that comes with the SDK, there are two List examples — Example 4 and Example 6 under Views/Lists demonstrate a SpeechView which extends LinearLayout to make a component for displaying Speech quotes. The corresponding classes in the sample code are `List4.java` and `List6.java`.

## Modifying an Existing View Type

There is an even easier option for creating a custom View which is useful in certain circumstances. If there is a component that is already very similar to what you want, you can simply extend that component and just override the behavior that you want to change. You can do all of the things you would do with a fully customized component, but by starting with a more specialized class in the View hierarchy, you can also get a lot of behavior for free that probably does exactly what you want.

For example, the SDK includes a [NotePad application](#) in the samples. This demonstrates many aspects of using the Android platform, among them is extending an EditText View to make a lined notepad. This is not a perfect example, and the APIs for doing this might change from this early preview, but it does demonstrate the principles.

If you haven't done so already, import the NotePad sample into Eclipse (or just look at the source using the link provided). In particular look at the definition of `MyEditText` in the [NoteEditor.java](#) file.

Some points to note here

## 1. The Definition

The class is defined with the following line:

```
public static class MyEditText extends EditText
```

- It is defined as an inner class within the NoteEditor activity, but it is public so that it could be accessed as NoteEditor.MyEditText from outside of the NoteEditor class if desired.
- It is static, meaning it does not generate the so-called "synthetic methods" that allow it to access data from the parent class, which in turn means that it really behaves as a separate class rather than something strongly related to NoteEditor. This is a cleaner way to create inner classes if they do not need access to state from the outer class, keeps the generated class small, and allows it to be used easily from other classes.
- It extends EditText, which is the View we have chosen to customize in this case. When we are finished, the new class will be able to substitute for a normal EditText view.

## 2. Class Initialization

As always, the super is called first. Furthermore, this is not a default constructor, but a parameterized one. The EditText is created with these parameters when it is inflated from an XML layout file, thus, our constructor needs to both take them and pass them to the superclass constructor as well.

## 3. Overridden Methods

In this example, there is only one method to be overridden: `onDraw()` — but there could easily be others needed when you create your own custom components.

For the NotePad sample, overriding the `onDraw()` method allows us to paint the blue lines on the EditText view canvas (the canvas is passed into the overridden `onDraw()` method). The `super.onDraw()` method is called before the method ends. The superclass method should be invoked, but in this case, we do it at the end after we have painted the lines we want to include.

## 4. Use the Custom Component

We now have our custom component, but how can we use it? In the NotePad example, the custom component is used directly from the declarative layout, so take a look at `note_editor.xml` in the `res/layout` folder.

```
<view  
    class="com.android.notepad.NoteEditor$MyEditText"  
    id="@+id/note"  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent"  
    android:background="@android:drawable/empty"  
    android:padding="10dip"  
    android:scrollbars="vertical"  
    android:fadingEdge="vertical" />
```

- The custom component is created as a generic view in the XML, and the class is specified using the full package. Note also that the inner class we defined is referenced using the NoteEditor\$MyEditText notation which is a standard way to refer to inner classes in the Java programming language.

If your custom View component is not defined as an inner class, then you can, alternatively, declare the View component with the XML element name, and exclude the `class` attribute. For example:

```
<com.android.notepad.MyEditText  
    id="@+id/note"  
    ... />
```

Notice that the `MyEditText` class is now a separate class file. When the class is nested in the `NoteEditor` class, this technique will not work.

- The other attributes and parameters in the definition are the ones passed into the custom component constructor, and then passed through to the `EditText` constructor, so they are the same parameters that you would use for an `EditText` view. Note that it is possible to add your own parameters as well, and we will touch on this again below.

And that's all there is to it. Admittedly this is a simple case, but that's the point — creating custom components is only as complicated as you need it to be.

A more sophisticated component may override even more `on...` methods and introduce some of its own helper methods, substantially customizing its properties and behavior. The only limit is your imagination and what you need the component to do.



# App Resources

It takes more than just code to build a great app. Resources are the additional files and static content that your code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and more.

## Blog Articles

### New Tools For Managing Screen Sizes

Android 3.2 includes new tools for supporting devices with a wide range of screen sizes. One important result is better support for a new size of screen; what is typically called a ?7-inch? tablet. This release also offers several new APIs to simplify developers? work in adjusting to different screen sizes.

### Holo Everywhere

Before Android 4.0 the variance in system themes from device to device could make it difficult to design an app with a single predictable look and feel. We set out to improve this situation for the developer community in Ice Cream Sandwich and beyond.

### New Mode for Apps on Large Screens

Android tablets are becoming more popular, and we're pleased to note that the vast majority of apps resize to the larger screens just fine. To keep the few apps that don't resize well from frustrating users with awkward-looking apps on their tablets, Android 3.2 introduces a screen compatibility mode that makes these apps more usable on tablets.

## Training

### Supporting Different Devices

This class teaches you how to use basic platform features that leverage alternative resources and other features so your app can provide an optimized user experience on a variety of Android-compatible devices, using a single application package (APK).

## **Designing for Multiple Screens**

This class shows you how to implement a user interface that's optimized for several screen configurations.

# Resources Overview

## Topics

1. [Providing Resources](#)
2. [Accessing Resources](#)
3. [Handling Runtime Changes](#)
4. [Localization](#)

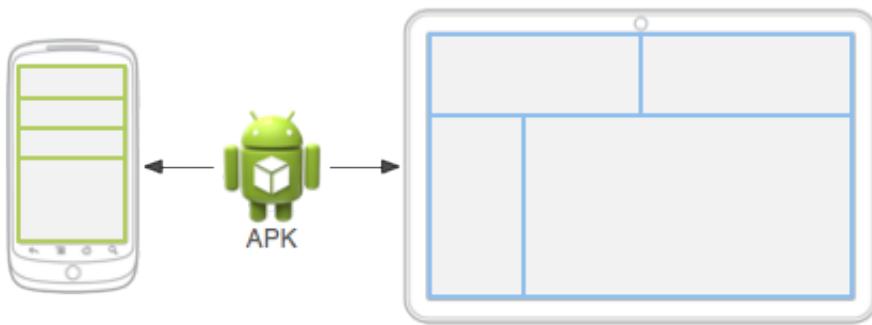
## Reference

1. [Resource Types](#)

You should always externalize resources such as images and strings from your application code, so that you can maintain them independently. Externalizing your resources also allows you to provide alternative resources that support specific device configurations such as different languages or screen sizes, which becomes increasingly important as more Android-powered devices become available with different configurations. In order to provide compatibility with different configurations, you must organize resources in your project's `res/` directory, using various sub-directories that group resources by type and configuration.



**Figure 1.** Two different devices, each using the default layout (the app provides no alternative layouts).



**Figure 2.** Two different devices, each using a different layout provided for different screen sizes.

For any type of resource, you can specify *default* and multiple *alternative* resources for your application:

- Default resources are those that should be used regardless of the device configuration or when there are no alternative resources that match the current configuration.
- Alternative resources are those that you've designed for use with a specific configuration. To specify that a group of resources are for a specific configuration, append an appropriate configuration qualifier to the directory name.

For example, while your default UI layout is saved in the `res/layout/` directory, you might specify a different layout to be used when the screen is in landscape orientation, by saving it in the `res/layout-land/` directory. Android automatically applies the appropriate resources by matching the device's current configuration to your resource directory names.

Figure 1 illustrates how the system applies the same layout for two different devices when there are no alternative resources available. Figure 2 shows the same application when it adds an alternative layout resource for larger screens.

The following documents provide a complete guide to how you can organize your application resources, specify alternative resources, access them in your application, and more:

### [Providing Resources](#)

What kinds of resources you can provide in your app, where to save them, and how to create alternative resources for specific device configurations.

### [Accessing Resources](#)

How to use the resources you've provided, either by referencing them from your application code or from other XML resources.

### [Handling Runtime Changes](#)

How to manage configuration changes that occur while your Activity is running.

### [Localization](#)

A bottom-up guide to localizing your application using alternative resources. While this is just one specific use of alternative resources, it is very important in order to reach more users.

### [Resource Types](#)

A reference of various resource types you can provide, describing their XML elements, attributes, and syntax. For example, this reference shows you how to create a resource for application menus, drawables, animations, and more.

# Providing Resources

## Quickview

- Different types of resources belong in different subdirectories of `res/`
- Alternative resources provide configuration-specific resource files
- Always include default resources so your app does not depend on specific device configurations

## In this document

1. [Grouping Resource Types](#)
2. [Providing Alternative Resources](#)
  1. [Qualifier name rules](#)
  2. [Creating alias resources](#)
3. [Providing the Best Device Compatibility with Resources](#)
4. [How Android Finds the Best-matching Resource](#)

## See also

1. [Accessing Resources](#)
2. [Resource Types](#)
3. [Supporting Multiple Screens](#)

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

Once you externalize your application resources, you can access them using resource IDs that are generated in your project's `R` class. How to use resources in your application is discussed in [Accessing Resources](#). This document shows you how to group your resources in your Android project and provide alternative resources for specific device configurations.

## Grouping Resource Types

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

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

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

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

Directory	Resource Type
<code>animator/</code>	XML files that define <a href="#">property animations</a> .
<code>anim/</code>	XML files that define <a href="#">tween animations</a> . (Property animations can also be saved in this directory, but the <code>animator/</code> directory is preferred for property animations to distinguish between the two types.)
<code>color/</code>	XML files that define a state list of colors. See <a href="#">Color State List Resource</a>
<code>drawable/</code>	Bitmap files (.png, .9.png, .jpg, .gif) or XML files that are compiled into the following drawable resource subtypes: <ul style="list-style-type: none"><li>• Bitmap files</li><li>• Nine-Patches (re-sizable bitmaps)</li><li>• State lists</li><li>• Shapes</li><li>• Animation drawables</li><li>• Other drawables</li></ul> See <a href="#">Drawable Resources</a> .
<code>layout/</code>	XML files that define a user interface layout. See <a href="#">Layout Resource</a> .
<code>menu/</code>	XML files that define application menus, such as an Options Menu, Context Menu, or Sub Menu. See <a href="#">Menu Resource</a> .
<code>raw/</code>	Arbitrary files to save in their raw form. To open these resources with a raw <a href="#">InputStream</a> , call <a href="#">Resources.openRawResource()</a> with the resource ID, which is <code>R.raw.filename</code> . However, if you need access to original file names and file hierarchy, you might consider saving some resources in the <code>assets/</code> directory (instead of <code>res/raw/</code> ). Files in <code>assets/</code> are not given a resource ID, so you can read them only using <a href="#">AssetManager</a> .
<code>values/</code>	XML files that contain simple values, such as strings, integers, and colors. <p>Whereas XML resource files in other <code>res/</code> subdirectories define a single resource based on the XML filename, files in the <code>values/</code> directory describe multiple resources. For a file in this directory, each child of the <code>&lt;resources&gt;</code> element defines a single resource. For example, a <code>&lt;string&gt;</code> element creates an <code>R.string</code> resource and a <code>&lt;color&gt;</code> element creates an <code>R.color</code> resource.</p> Because each resource is defined with its own XML element, you can name the file whatever you want and place different resource types in one file. However, for clarity, you might want to place unique resource types in different files. For example, here are some filename conventions for resources you can create in this directory:
	<ul style="list-style-type: none"><li>• arrays.xml for resource arrays (<a href="#">typed arrays</a>).</li><li>• colors.xml for <a href="#">color values</a></li><li>• dimens.xml for <a href="#">dimension values</a>.</li></ul>

- strings.xml for [string values](#).
- styles.xml for [styles](#).

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

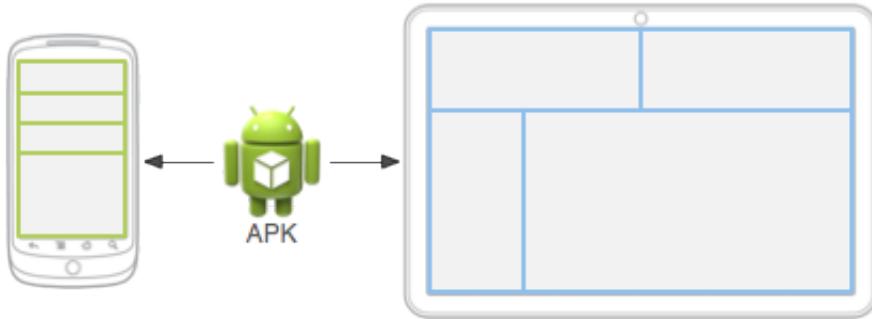
xml/ Arbitrary XML files that can be read at runtime by calling [Resources.getXML\(\)](#). Various XML configuration files must be saved here, such as a [searchable configuration](#).

**Caution:** Never save resource files directly inside the res/ directory—it will cause a compiler error.

For more information about certain types of resources, see the [Resource Types](#) documentation.

The resources that you save in the subdirectories defined in table 1 are your "default" resources. That is, these resources define the default design and content for your application. However, different types of Android-powered devices might call for different types of resources. For example, if a device has a larger than normal screen, then you should provide different layout resources that take advantage of the extra screen space. Or, if a device has a different language setting, then you should provide different string resources that translate the text in your user interface. To provide these different resources for different device configurations, you need to provide alternative resources, in addition to your default resources.

## Providing Alternative Resources



**Figure 1.** Two different devices, each using different layout resources.

Almost every application should provide alternative resources to support specific device configurations. For instance, you should include alternative drawable resources for different screen densities and alternative string resources for different languages. At runtime, Android detects the current device configuration and loads the appropriate resources for your application.

To specify configuration-specific alternatives for a set of resources:

1. Create a new directory in res/ named in the form <resources\_name>-<config\_qualifier>.
  - <resources\_name> is the directory name of the corresponding default resources (defined in table 1).
  - <qualifier> is a name that specifies an individual configuration for which these resources are to be used (defined in table 2).

You can append more than one <qualifier>. Separate each one with a dash.

**Caution:** When appending multiple qualifiers, you must place them in the same order in which they are listed in table 2. If the qualifiers are ordered wrong, the resources are ignored.

- Save the respective alternative resources in this new directory. The resource files must be named exactly the same as the default resource files.

For example, here are some default and alternative resources:

```
res/
    drawable/
        icon.png
        background.png
    drawable-hdpi/
        icon.png
        background.png
```

The `hdpi` qualifier indicates that the resources in that directory are for devices with a high-density screen. The images in each of these drawable directories are sized for a specific screen density, but the filenames are exactly the same. This way, the resource ID that you use to reference the `icon.png` or `background.png` image is always the same, but Android selects the version of each resource that best matches the current device, by comparing the device configuration information with the qualifiers in the resource directory name.

Android supports several configuration qualifiers and you can add multiple qualifiers to one directory name, by separating each qualifier with a dash. Table 2 lists the valid configuration qualifiers, in order of precedence—if you use multiple qualifiers for a resource directory, you must add them to the directory name in the order they are listed in the table.

**Table 2.** Configuration qualifier names.

	<b>Configuration Qualifier Values</b>	<b>Description</b>
MCC and MNC	Examples: <code>mcc310</code> <code>mcc310-mnc004</code> <code>mcc208-mnc00</code> etc.	The mobile country code (MCC), optionally followed by mobile network code (MNC) of the SIM card in the device. For example, <code>mcc310</code> is U.S. on any carrier, <code>mcc310-mnc004</code> is U.S. on Verizon, and <code>mcc208-mnc00</code> is France on Orange.  If the device uses a radio connection (GSM phone), the MCC and MNC values come from the SIM card.  You can also use the MCC alone (for example, to include country-specific legal restrictions in your application). If you need to specify based on the language only, then use the <code>language</code> qualifier instead (discussed next). If you decide to use the MCC and MNC qualifiers, do so with care and test that it works as expected.  Also see the configuration fields <a href="#">mcc</a> , and <a href="#">mnc</a> , which indicate the current mobile country and mobile network code, respectively.
Language and region	Examples: <code>en</code> <code>fr</code> <code>en-rUS</code> <code>fr-rFR</code> <code>fr-rCA</code> etc.	The language is defined by a two-letter <a href="#">ISO 639-1</a> language code, optionally followed by a letter <a href="#">ISO 3166-1-alpha-2</a> region code (preceded by lowercase "r").  The codes are <i>not</i> case-sensitive; the <code>r</code> prefix is used to distinguish the region port from the language code. You cannot specify a language and a region alone.  This can change during the life of your application if the user changes his or her language or region system settings. See <a href="#">Handling Runtime Changes</a> for information about how this can affect your application during runtime.

See [Localization](#) for a complete guide to localizing your application for other languages.

Also see the [locale](#) configuration field, which indicates the current locale.

The layout direction of your application. `ldrtl` means "layout-direction-right-to-left" and `ldltr` means "layout-direction-left-to-right" and is the default implicit value.

This can apply to any resource such as layouts, drawables, or values.

For example, if you want to provide some specific layout for the Arabic language and want to use the generic layout for any other "right-to-left" language (like Persian or Hebrew) then you would have:

```
res/
    layout/
        main.xml      (Default layout)
    layout-ar/
        main.xml      (Specific layout for Arabic)
    layout-ldrtl/
        main.xml      (Any "right-to-left" language, except
                      for Arabic, because the "ar" language
                      has a higher precedence.)
```

**Note:** To enable right-to-left layout features for your app, you must set [supportsRtl](#) to "true" and set [targetSdkVersion](#) to 17 or higher.

*Added in API level 17.*

The fundamental size of a screen, as indicated by the shortest dimension of the available area. Specifically, the device's `smallestWidth` is the shortest of the screen's available widths (you may also think of it as the "smallest possible width" for the screen). You can use this qualifier to ensure that, regardless of the screen's current orientation, your application always has at least `<N> dps` of width available for its UI.

For example, if your layout requires that its smallest dimension of screen area be at least 600dp at all times, then you can use this qualifier to create the layout resources, `res/layout-sw600dp/`. The system will use these resources only when the smallest dimension of the screen is at least 600dp, regardless of whether the 600dp side is the user-perceived width. The `smallestWidth` is a fixed screen size characteristic of the device; **the `smallestWidth` does not change when the screen's orientation changes**.

The `smallestWidth` of a device takes into account screen decorations and system UI elements. For example, if the device has some persistent UI elements on the screen that account for space along the horizontal axis of the `smallestWidth`, the system declares the `smallestWidth` to be smaller than the actual screen size, because those are screen pixels not available for your UI. Thus, the value you specify should be the actual smallest dimension *required by your layout* (usually, this value is the "smallest width" that your layout supports, regardless of the screen's current orientation).

Some values you might use here for common screen sizes:

- 320, for devices with screen configurations such as:

`sw<N>dp`

Examples:

`smallestWidth`    `sw320dp`  
                      `sw600dp`  
                      `sw720dp`  
                      etc.

- 240x320 ldpi (QVGA handset)
- 320x480 mdpi (handset)
- 480x800 hdpi (high density handset)
- 480, for screens such as 480x800 mdpi (tablet/handset).
- 600, for screens such as 600x1024 mdpi (7" tablet).
- 720, for screens such as 720x1280 mdpi (10" tablet).

When your application provides multiple resource directories with different values for the `smallestWidth` qualifier, the system uses the one closest to (without exceeding) the device's current width.

*Added in API level 13.*

Also see the [android:requiresSmallestWidthDp](#) attribute, which declares the minimum smallestWidth with which your application is compatible, and the [smallestWidthDp](#) configuration field, which holds the device's smallestWidth value.

For more information about designing for different screens and using this qualifier, see the [Porting Multiple Screens](#) developer guide.

Specifies a minimum available screen width, in dp units at which the resource should be used—defined by the <N> value. This configuration value will change when the orientation changes between landscape and portrait to match the current actual width.

When your application provides multiple resource directories with different values for the `screenWidthDp` configuration, the system uses the one closest to (without exceeding) the device's current available width. The value here takes into account screen decorations, so if the device has some UI elements on the left or right edge of the display, it uses a value for the width that is less than the real screen size, accounting for these UI elements and reducing the applicable space.

*Added in API level 13.*

Also see the [screenWidthDp](#) configuration field, which holds the current screen width.

For more information about designing for different screens and using this qualifier, see the [Porting Multiple Screens](#) developer guide.

Specifies a minimum available screen height, in "dp" units at which the resource should be used—defined by the <N> value. This configuration value will change when the orientation changes between landscape and portrait to match the current actual height.

When your application provides multiple resource directories with different values for the `screenHeightDp` configuration, the system uses the one closest to (without exceeding) the device's current available height. The value here takes into account screen decorations, so if the device has some UI elements on the top or bottom edge of the display, it uses a value for the height that is less than the real screen size, accounting for these UI elements and reducing the applicable space. Screen decorations that are not fixed (such as a phone status bar that can be hidden when full screen) are *not* accounted for here, nor are window decorations like the title bar, so applications must be prepared to deal with a somewhat smaller space than they may expect.

Available width	w<N>dp Examples: w720dp w1024dp etc.
-----------------	--

Available height	h<N>dp Examples: h720dp h1024dp etc.
------------------	--

*Added in API level 13.*

Also see the [screenHeightDp](#) configuration field, which holds the current screen height in dp units.

For more information about designing for different screens and using this qualifier, see the [Supporting Multiple Screens](#) developer guide.

- **small**: Screens that are of similar size to a low-density QVGA screen. The minimum layout size for a small screen is approximately 320x426 dp units. Examples include WVGA low density and VGA high density.
- **normal**: Screens that are of similar size to a medium-density HVGA screen. The minimum layout size for a normal screen is approximately 320x470 dp units. Examples include such screens as a WQVGA low density, HVGA medium density, WVGA high density.
- **large**: Screens that are of similar size to a medium-density VGA screen. The minimum layout size for a large screen is approximately 480x640 dp units. Examples include WVGA medium density screens.
- **xlarge**: Screens that are considerably larger than the traditional medium-density screen. The minimum layout size for an xlarge screen is approximately 640x960 dp units. In most cases, devices with extra large screens would be too large to fit in a pocket and would most likely be tablet-style devices. *Added in API level 9.*

Screen size	small normal large xlarge
-------------	------------------------------------

**Note:** Using a size qualifier does not imply that the resources are *only* for screens of that size. If you do not provide alternative resources with qualifiers that better match the current screen configuration, the system may use whichever resources are the [best match](#).

**Caution:** If all your resources use a size qualifier that is *larger* than the current screen size, the system will **not** use them and your application will crash at runtime (for example, if all your resources are tagged with the `xlarge` qualifier, but the device is a normal-size screen).

*Added in API level 4.*

See [Supporting Multiple Screens](#) for more information.

Also see the [screenLayout](#) configuration field, which indicates whether the screen is long, not long, normal, or large.

- **long**: Long screens, such as WQVGA, WVGA, FWVGA
- **notlong**: Not long screens, such as QVGA, HVGA, and VGA

*Added in API level 4.*

This is based purely on the aspect ratio of the screen (a "long" screen is wider). This is not related to the screen orientation.

Also see the [screenLayout](#) configuration field, which indicates whether the screen is portrait or landscape.

- **port**: Device is in portrait orientation (vertical)
- **land**: Device is in landscape orientation (horizontal)

Screen aspect	long notlong
Screen orientation	port land

This can change during the life of your application if the user rotates the screen. See the [Runtime Changes](#) for information about how this affects your application during runtime.

Also see the [orientation](#) configuration field, which indicates the current device orientation.

- car: Device is displaying in a car dock
- desk: Device is displaying in a desk dock
- television: Device is displaying on a television, providing a "ten foot UI" where its UI is on a large screen that the user is far away from, primarily operating via a remote control or DPAD or other non-pointer interaction
- appliance: Device is serving as an appliance, with no display

UI mode	car desk television appliance
---------	--

*Added in API level 8, television added in API 13.*

For information about how your app can respond when the device is inserted into or removed from a dock, read [Determining and Monitoring the Docking State and Type](#).

This can change during the life of your application if the user places the device in a different mode. You can enable or disable some of these modes using [UiModeManager](#). See [Handling Runtime Changes](#) for information about how this affects your application during runtime.

- night: Night time
- notnight: Day time

*Added in API level 8.*

Night mode	night notnight
------------	-------------------

This can change during the life of your application if night mode is left in auto mode, in which case the mode changes based on the time of day. You can enable or disable the night mode using [UiModeManager](#). See [Handling Runtime Changes](#) for information about how this affects your application during runtime.

- ldpi: Low-density screens; approximately 120dpi.
- mdpi: Medium-density (on traditional HVGA) screens; approximately 160dpi.
- hdpi: High-density screens; approximately 240dpi.
- xhdpi: Extra high-density screens; approximately 320dpi. *Added in API level 17.*
- nodpi: This can be used for bitmap resources that you do not want to be forced to match the device density.
- tvdpi: Screens somewhere between mdpi and hdpi; approximately 213dpi. It is considered a "primary" density group. It is mostly intended for televisions, but you shouldn't need it—providing mdpi and hdpi resources is sufficient for most devices. The system will scale them as appropriate. This qualifier was introduced with API level 13.

Screen pixel density (dpi)	ldpi mdpi hdpi xhdpi nodpi tvdpi
----------------------------	---

There is a 3:4:6:8 scaling ratio between the four primary densities (ignoring the tvdpi value). So, a 9x9 bitmap in ldpi is 12x12 in mdpi, 18x18 in hdpi and 24x24 in xhdpi.

If you decide that your image resources don't look good enough on a television or other high-density devices and want to try tvdpi resources, the scaling factor is 1.33\*mdpi. For example, a 100px image for mdpi screens should be 133px x 133px for tvdpi.

**Note:** Using a density qualifier does not imply that the resources are *only* for screens of that density. If you do not provide alternative resources with qualifiers that better match the device configuration, the system may use whichever resources are the [best match](#).

See [Supporting Multiple Screens](#) for more information about how to handle different screen densities and how Android might scale your bitmaps to fit the current density.

		<ul style="list-style-type: none"> <li><code>notouch</code>: Device does not have a touchscreen.</li> <li><code>finger</code>: Device has a touchscreen that is intended to be used through direction of the user's finger.</li> </ul>
Touchscreen type	<code>notouch</code> <code>finger</code>	Also see the <a href="#">touchscreen</a> configuration field, which indicates the type of touch device.
Keyboard availability	<code>keysexposed</code> <code>keyshidden</code> <code>keyssoft</code>	<ul style="list-style-type: none"> <li><code>keysexposed</code>: Device has a keyboard available. If the device has a software keyboard enabled (which is likely), this may be used even when the hardware keyboard is exposed to the user, even if the device has no hardware keyboard. If no software keyboard is provided or it's disabled, then this is only used when a hardware keyboard is visible.</li> <li><code>keyshidden</code>: Device has a hardware keyboard available but it is hidden. The device does <i>not</i> have a software keyboard enabled.</li> <li><code>keyssoft</code>: Device has a software keyboard enabled, whether it's visible or not.</li> </ul> <p>If you provide <code>keysexposed</code> resources, but not <code>keyssoft</code> resources, the system provides <code>keysexposed</code> resources regardless of whether a keyboard is visible, as long as there is at least one software keyboard enabled.</p> <p>This can change during the life of your application if the user opens a hardware keyboard. See the <a href="#">Handling Runtime Changes</a> for information about how this affects your application at runtime.</p>
Primary text input method	<code>nokeys</code> <code>qwerty</code> <code>12key</code>	<ul style="list-style-type: none"> <li><code>nokeys</code>: Device has no hardware keys for text input.</li> <li><code>qwerty</code>: Device has a hardware qwerty keyboard, whether it's visible to the user or not.</li> <li><code>12key</code>: Device has a hardware 12-key keyboard, whether it's visible to the user or not.</li> </ul> <p>Also see the <a href="#">keyboard</a> configuration field, which indicates the primary text input methods available.</p> <ul style="list-style-type: none"> <li><code>navexposed</code>: Navigation keys are available to the user.</li> <li><code>navhidden</code>: Navigation keys are not available (such as behind a closed lid).</li> </ul> <p>This can change during the life of your application if the user reveals the navigation keys. See the <a href="#">Handling Runtime Changes</a> for information about how this affects your application at runtime.</p> <p>Also see the <a href="#">navigationHidden</a> configuration field, which indicates whether the navigation keys are hidden.</p> <ul style="list-style-type: none"> <li><code>nonav</code>: Device has no navigation facility other than using the touchscreen.</li> <li><code>dpad</code>: Device has a directional-pad (d-pad) for navigation.</li> <li><code>trackball</code>: Device has a trackball for navigation.</li> <li><code>wheel</code>: Device has a directional wheel(s) for navigation (uncommon).</li> </ul> <p>Also see the <a href="#">navigation</a> configuration field, which indicates the type of navigation methods available.</p>
Primary non-touch navigation method	<code>nonav</code> <code>dpad</code> <code>trackball</code> <code>wheel</code>	

## Examples:

Platform Version (API level)	v3	The API level supported by the device. For example, v1 for API level 1 (devices with 1.0 or higher) and v4 for API level 4 (devices with Android 1.6 or higher). See the <a href="#">levels</a> document for more information about these values.
	v4	
	v7	

etc.

**Note:** Some configuration qualifiers have been added since Android 1.0, so not all versions of Android support all the qualifiers. Using a new qualifier implicitly adds the platform version qualifier so that older devices are sure to ignore it. For example, using a `w600dp` qualifier will automatically include the `v13` qualifier, because the available-width qualifier was new in API level 13. To avoid any issues, always include a set of default resources (a set of resources with *no qualifiers*). For more information, see the section about [Providing the Best Device Compatibility with Resources](#).

## Qualifier name rules

Here are some rules about using configuration qualifier names:

- You can specify multiple qualifiers for a single set of resources, separated by dashes. For example, `drawable-en-rUS-land` applies to US-English devices in landscape orientation.
- The qualifiers must be in the order listed in [table 2](#). For example:
  - Wrong: `drawable-hdpi-port/`
  - Correct: `drawable-port-hdpi/`
- Alternative resource directories cannot be nested. For example, you cannot have `res/drawable/drawable-en/`.
- Values are case-insensitive. The resource compiler converts directory names to lower case before processing to avoid problems on case-insensitive file systems. Any capitalization in the names is only to benefit readability.
- Only one value for each qualifier type is supported. For example, if you want to use the same drawable files for Spain and France, you *cannot* have a directory named `drawable-rES-rFR/`. Instead you need two resource directories, such as `drawable-rES/` and `drawable-rFR/`, which contain the appropriate files. However, you are not required to actually duplicate the same files in both locations. Instead, you can create an alias to a resource. See [Creating alias resources](#) below.

After you save alternative resources into directories named with these qualifiers, Android automatically applies the resources in your application based on the current device configuration. Each time a resource is requested, Android checks for alternative resource directories that contain the requested resource file, then [finds the best-matching resource](#) (discussed below). If there are no alternative resources that match a particular device configuration, then Android uses the corresponding default resources (the set of resources for a particular resource type that does not include a configuration qualifier).

## Creating alias resources

When you have a resource that you'd like to use for more than one device configuration (but do not want to provide as a default resource), you do not need to put the same resource in more than one alternative resource directory. Instead, you can (in some cases) create an alternative resource that acts as an alias for a resource saved in your default resource directory.

**Note:** Not all resources offer a mechanism by which you can create an alias to another resource. In particular, animation, menu, raw, and other unspecified resources in the `xml/` directory do not offer this feature.

For example, imagine you have an application icon, `icon.png`, and need unique version of it for different locales. However, two locales, English-Canadian and French-Canadian, need to use the same version. You might assume that you need to copy the same image into the resource directory for both English-Canadian and

French-Canadian, but it's not true. Instead, you can save the image that's used for both as `icon_ca.png` (any name other than `icon.png`) and put it in the default `res/drawable/` directory. Then create an `icon.xml` file in `res/drawable-en-rCA/` and `res/drawable-fr-rCA/` that refers to the `icon_ca.png` resource using the `<bitmap>` element. This allows you to store just one version of the PNG file and two small XML files that point to it. (An example XML file is shown below.)

## Drawable

To create an alias to an existing drawable, use the `<bitmap>` element. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
    android:src="@drawable/icon_ca" />
```

If you save this file as `icon.xml` (in an alternative resource directory, such as `res/drawable-en-rCA/`), it is compiled into a resource that you can reference as `R.drawable.icon`, but is actually an alias for the `R.drawable.icon_ca` resource (which is saved in `res/drawable/`).

## Layout

To create an alias to an existing layout, use the `<include>` element, wrapped in a `<merge>`. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<merge>
    <include layout="@layout/main_ltr"/>
</merge>
```

If you save this file as `main.xml`, it is compiled into a resource you can reference as `R.layout.main`, but is actually an alias for the `R.layout.main_ltr` resource.

## Strings and other simple values

To create an alias to an existing string, simply use the resource ID of the desired string as the value for the new string. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello</string>
    <string name="hi">@string/hello</string>
</resources>
```

The `R.string.hi` resource is now an alias for the `R.string.hello`.

[Other simple values](#) work the same way. For example, a color:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="yellow">#f00</color>
    <color name="highlight">@color/red</color>
</resources>
```

# Providing the Best Device Compatibility with Resources

In order for your application to support multiple device configurations, it's very important that you always provide default resources for each type of resource that your application uses.

For example, if your application supports several languages, always include a `values/` directory (in which your strings are saved) *without* a [language and region qualifier](#). If you instead put all your string files in directories that have a language and region qualifier, then your application will crash when run on a device set to a language that your strings do not support. But, as long as you provide default `values/` resources, then your application will run properly (even if the user doesn't understand that language—it's better than crashing).

Likewise, if you provide different layout resources based on the screen orientation, you should pick one orientation as your default. For example, instead of providing layout resources in `layout-land/` for landscape and `layout-port/` for portrait, leave one as the default, such as `layout/` for landscape and `layout-port/` for portrait.

Providing default resources is important not only because your application might run on a configuration you had not anticipated, but also because new versions of Android sometimes add configuration qualifiers that older versions do not support. If you use a new resource qualifier, but maintain code compatibility with older versions of Android, then when an older version of Android runs your application, it will crash if you do not provide default resources, because it cannot use the resources named with the new qualifier. For example, if your [`minSdkVersion`](#) is set to 4, and you qualify all of your drawable resources using [`night mode`](#) (night or not-night, which were added in API Level 8), then an API level 4 device cannot access your drawable resources and will crash. In this case, you probably want `notnight` to be your default resources, so you should exclude that qualifier so your drawable resources are in either `drawable/` or `drawable-night/`.

So, in order to provide the best device compatibility, always provide default resources for the resources your application needs to perform properly. Then create alternative resources for specific device configurations using the configuration qualifiers.

There is one exception to this rule: If your application's [`minSdkVersion`](#) is 4 or greater, you *do not* need default drawable resources when you provide alternative drawable resources with the [`screen density`](#) qualifier. Even without default drawable resources, Android can find the best match among the alternative screen densities and scale the bitmaps as necessary. However, for the best experience on all types of devices, you should provide alternative drawables for all three types of density.

## How Android Finds the Best-matching Resource

When you request a resource for which you provide alternatives, Android selects which alternative resource to use at runtime, depending on the current device configuration. To demonstrate how Android selects an alternative resource, assume the following drawable directories each contain different versions of the same images:

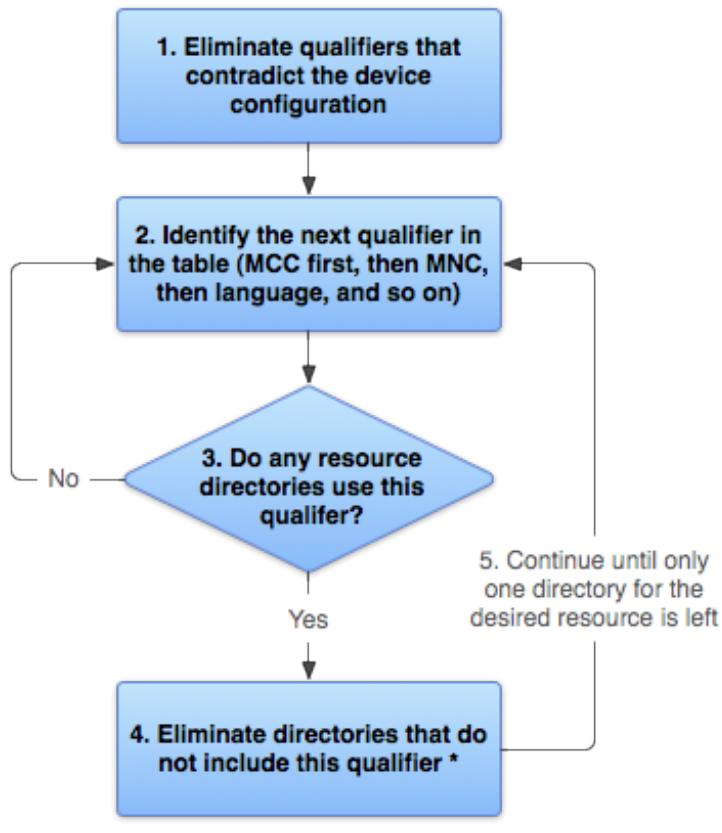
```
drawable/  
drawable-en/  
drawable-fr-rCA/  
drawable-en-port/  
drawable-en-notouch-12key/  
drawable-port-ldpi/  
drawable-port-notouch-12key/
```

And assume the following is the device configuration:

Locale = en-GB  
Screen orientation = port  
Screen pixel density = hdpi  
Touchscreen type = notouch  
Primary text input method = 12key

By comparing the device configuration to the available alternative resources, Android selects drawables from drawable-en-port.

The system arrives at its decision for which resources to use with the following logic:



\* If the qualifier is screen density, the system selects the "best match" and the process is done

**Figure 2.** Flowchart of how Android finds the best-matching resource.

1. Eliminate resource files that contradict the device configuration.

The drawable-fr-rCA/ directory is eliminated, because it contradicts the en-GB locale.

```
drawable/
drawable-en/
drawable-fr-rCA/
drawable-en-port/
drawable-en-notouch-12key/
drawable-port-ldpi/
drawable-port-notouch-12key/
```

**Exception:** Screen pixel density is the one qualifier that is not eliminated due to a contradiction. Even though the screen density of the device is hdpi, drawable-port-ldpi/ is not eliminated because

every screen density is considered to be a match at this point. More information is available in the [Supporting Multiple Screens](#) document.

2. Pick the (next) highest-precedence qualifier in the list ([table 2](#)). (Start with MCC, then move down.)
3. Do any of the resource directories include this qualifier?
  - If No, return to step 2 and look at the next qualifier. (In the example, the answer is "no" until the language qualifier is reached.)
  - If Yes, continue to step 4.
3. Eliminate resource directories that do not include this qualifier. In the example, the system eliminates all the directories that do not include a language qualifier:

```
drawable/  
drawable-en/  
drawable-en-port/  
drawable-en-notouch-12key/  
drawable-port-ldpi/  
drawable-port-notouch-12key/
```

**Exception:** If the qualifier in question is screen pixel density, Android selects the option that most closely matches the device screen density. In general, Android prefers scaling down a larger original image to scaling up a smaller original image. See [Supporting Multiple Screens](#).

4. Go back and repeat steps 2, 3, and 4 until only one directory remains. In the example, screen orientation is the next qualifier for which there are any matches. So, resources that do not specify a screen orientation are eliminated:

```
drawable-en/  
drawable-en-port/  
drawable-en-notouch-12key/
```

The remaining directory is `drawable-en-port`.

Though this procedure is executed for each resource requested, the system further optimizes some aspects. One such optimization is that once the device configuration is known, it might eliminate alternative resources that can never match. For example, if the configuration language is English ("en"), then any resource directory that has a language qualifier set to something other than English is never included in the pool of resources checked (though a resource directory *without* the language qualifier is still included).

When selecting resources based on the screen size qualifiers, the system will use resources designed for a screen smaller than the current screen if there are no resources that better match (for example, a large-size screen will use normal-size screen resources if necessary). However, if the only available resources are *larger* than the current screen, the system will **not** use them and your application will crash if no other resources match the device configuration (for example, if all layout resources are tagged with the `xlarge` qualifier, but the device is a normal-size screen).

**Note:** The *precedence* of the qualifier (in [table 2](#)) is more important than the number of qualifiers that exactly match the device. For example, in step 4 above, the last choice on the list includes three qualifiers that exactly match the device (orientation, touchscreen type, and input method), while `drawable-en` has only one parameter that matches (language). However, language has a higher precedence than these other qualifiers, so `drawable-port-notouch-12key` is out.

To learn more about how to use resources in your application, continue to [Accessing Resources](#).

# Accessing Resources

## Quickview

- Resources can be referenced from code using integers from `R.java`, such as `R.drawable.myimage`
- Resources can be referenced from resources using a special XML syntax, such as `@drawable/myimage`
- You can also access your app resources with methods in [Resources](#)

## Key classes

1. [Resources](#)

## In this document

1. [Accessing Resources from Code](#)
2. [Accessing Resources from XML](#)
  1. [Referencing style attributes](#)
3. [Accessing Platform Resources](#)

## See also

1. [Providing Resources](#)
2. [Resource Types](#)

Once you provide a resource in your application (discussed in [Providing Resources](#)), you can apply it by referencing its resource ID. All resource IDs are defined in your project's `R` class, which the `aapt` tool automatically generates.

When your application is compiled, `aapt` generates the `R` class, which contains resource IDs for all the resources in your `res/` directory. For each type of resource, there is an `R` subclass (for example, `R.drawable` for all drawable resources), and for each resource of that type, there is a static integer (for example, `R.drawable.icon`). This integer is the resource ID that you can use to retrieve your resource.

Although the `R` class is where resource IDs are specified, you should never need to look there to discover a resource ID. A resource ID is always composed of:

- The *resource type*: Each resource is grouped into a "type," such as `string`, `drawable`, and `layout`. For more about the different types, see [Resource Types](#).
- The *resource name*, which is either: the filename, excluding the extension; or the value in the XML `android:name` attribute, if the resource is a simple value (such as a string).

There are two ways you can access a resource:

- **In code:** Using a static integer from a sub-class of your `R` class, such as:

```
R.string.hello
```

`string` is the resource type and `hello` is the resource name. There are many Android APIs that can access your resources when you provide a resource ID in this format. See [Accessing Resources in Code](#).

- **In XML:** Using a special XML syntax that also corresponds to the resource ID defined in your `R` class, such as:

```
@string/hello
```

`string` is the resource type and `hello` is the resource name. You can use this syntax in an XML resource any place where a value is expected that you provide in a resource. See [Accessing Resources from XML](#).

## Accessing Resources in Code

You can use a resource in code by passing the resource ID as a method parameter. For example, you can set an `ImageView` to use the `res/drawable/myimage.png` resource using [`setImageResource\(\)`](#):

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

You can also retrieve individual resources using methods in [Resources](#), which you can get an instance of with [`getResources\(\)`](#).

## Access to Original Files

While uncommon, you might need access your original files and directories. If you do, then saving your files in `res/` won't work for you, because the only way to read a resource from `res/` is with the resource ID. Instead, you can save your resources in the `assets/` directory.

Files saved in the `assets/` directory are *not* given a resource ID, so you can't reference them through the `R` class or from XML resources. Instead, you can query files in the `assets/` directory like a normal file system and read raw data using [AssetManager](#).

However, if all you require is the ability to read raw data (such as a video or audio file), then save the file in the `res/raw/` directory and read a stream of bytes using [`openRawResource\(\)`](#).

## Syntax

Here's the syntax to reference a resource in code:

```
[<package_name>.]R.<resource_type>.<resource_name>
```

- `<package_name>` is the name of the package in which the resource is located (not required when referencing resources from your own package).
- `<resource_type>` is the `R` subclass for the resource type.
- `<resource_name>` is either the resource filename without the extension or the `android:name` attribute value in the XML element (for simple values).

See [Resource Types](#) for more information about each resource type and how to reference them.

## Use cases

There are many methods that accept a resource ID parameter and you can retrieve resources using methods in [Resources](#). You can get an instance of [Resources](#) with [Context.getResources\(\)](#).

Here are some examples of accessing resources in code:

```
// Load a background for the current screen from a drawable resource  
getWindow\(\).setBackgroundDrawableResource\(R.drawable.my\_background\_image\) ;  
  
// Set the Activity title by getting a string from the Resources object, because  
// this method requires a CharSequence rather than a resource ID  
getWindow\(\).setTitle\(getResources\(\).getText\(R.string.main\_title\)\);  
  
// Load a custom layout for the current screen  
setContentview\(R.layout.main\_screen\);  
  
// Set a slide in animation by getting an Animation from the Resources object  
mFlipper.setInAnimation(AnimationUtils.loadAnimation(this,  
    R.anim.hyperspace_in));  
  
// Set the text on a TextView object using a resource ID  
TextView msgTextView = (TextView) findViewById(R.id.msg) ;  
msgTextView.setText\(R.string.hello\_message\);
```

**Caution:** You should never modify the `R.java` file by hand—it is generated by the `aapt` tool when your project is compiled. Any changes are overridden next time you compile.

## Accessing Resources from XML

You can define values for some XML attributes and elements using a reference to an existing resource. You will often do this when creating layout files, to supply strings and images for your widgets.

For example, if you add a [Button](#) to your layout, you should use a [string resource](#) for the button text:

```
<Button  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:text="@string/submit" />
```

## Syntax

Here is the syntax to reference a resource in an XML resource:

```
@[<package_name>:]<resource_type>/<resource_name>
```

- `<package_name>` is the name of the package in which the resource is located (not required when referencing resources from the same package)
- `<resource_type>` is the `R` subclass for the resource type
- `<resource_name>` is either the resource filename without the extension or the `android:name` attribute value in the XML element (for simple values).

See [Resource Types](#) for more information about each resource type and how to reference them.

## Use cases

In some cases you must use a resource for a value in XML (for example, to apply a drawable image to a widget), but you can also use a resource in XML any place that accepts a simple value. For example, if you have the following resource file that includes a [color resource](#) and a [string resource](#):

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

You can use these resources in the following layout file to set the text color and text string:

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

In this case you don't need to specify the package name in the resource reference because the resources are from your own package. To reference a system resource, you would need to include the package name. For example:

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

**Note:** You should use string resources at all times, so that your application can be localized for other languages. For information about creating alternative resources (such as localized strings), see [Providing Alternative Resources](#). For a complete guide to localizing your application for other languages, see [Localization](#).

You can even use resources in XML to create aliases. For example, you can create a drawable resource that is an alias for another drawable resource:

```
<?xml version="1.0" encoding="utf-8"?>
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
    android:src="@drawable/other_drawable" />
```

This sounds redundant, but can be very useful when using alternative resource. Read more about [Creating alias resources](#).

## Referencing style attributes

A style attribute resource allows you to reference the value of an attribute in the currently-applied theme. Referencing a style attribute allows you to customize the look of UI elements by styling them to match standard variations supplied by the current theme, instead of supplying a hard-coded value. Referencing a style attribute essentially says, "use the style that is defined by this attribute, in the current theme."

To reference a style attribute, the name syntax is almost identical to the normal resource format, but instead of the at-symbol (@), use a question-mark (?), and the resource type portion is optional. For instance:

```
? [<package_name>:] [<resource_type>/] <resource_name>
```

For example, here's how you can reference an attribute to set the text color to match the "primary" text color of the system theme:

```
<EditText id="text"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:textColor="?android:textColorSecondary"  
    android:text="@string/hello_world" />
```

Here, the `android:textColor` attribute specifies the name of a style attribute in the current theme. Android now uses the value applied to the `android:textColorSecondary` style attribute as the value for `android:textColor` in this widget. Because the system resource tool knows that an attribute resource is expected in this context, you do not need to explicitly state the type (which would be `?android:attr/textColorSecondary`)—you can exclude the `attr` type.

## Accessing Platform Resources

Android contains a number of standard resources, such as styles, themes, and layouts. To access these resource, qualify your resource reference with the `android` package name. For example, Android provides a layout resource you can use for list items in a [ListAdapter](#):

```
setListAdapter(new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, strings));
```

In this example, `simple_list_item_1` is a layout resource defined by the platform for items in a [ListView](#). You can use this instead of creating your own layout for list items. For more information, see the [List View](#) developer guide.

# Handling Runtime Changes

## In this document

1. [Retaining an Object During a Configuration Change](#)
2. [Handling the Configuration Change Yourself](#)

## See also

1. [Providing Resources](#)
2. [Accessing Resources](#)
3. [Faster Screen Orientation Change](#)

Some device configurations can change during runtime (such as screen orientation, keyboard availability, and language). When such a change occurs, Android restarts the running [Activity](#) (`onDestroy()` is called, followed by `onCreate()`). The restart behavior is designed to help your application adapt to new configurations by automatically reloading your application with alternative resources that match the new device configuration.

To properly handle a restart, it is important that your activity restores its previous state through the normal [Activity lifecycle](#), in which Android calls `onSaveInstanceState()` before it destroys your activity so that you can save data about the application state. You can then restore the state during `onCreate()` or `onRestoreInstanceState()`.

To test that your application restarts itself with the application state intact, you should invoke configuration changes (such as changing the screen orientation) while performing various tasks in your application. Your application should be able to restart at any time without loss of user data or state in order to handle events such as configuration changes or when the user receives an incoming phone call and then returns to your application much later after your application process may have been destroyed. To learn how you can restore your activity state, read about the [Activity lifecycle](#).

However, you might encounter a situation in which restarting your application and restoring significant amounts of data can be costly and create a poor user experience. In such a situation, you have two other options:

- a. [Retain an object during a configuration change](#)

Allow your activity to restart when a configuration changes, but carry a stateful [Object](#) to the new instance of your activity.

- b. [Handle the configuration change yourself](#)

Prevent the system from restarting your activity during certain configuration changes, but receive a callback when the configurations do change, so that you can manually update your activity as necessary.

## Retaining an Object During a Configuration Change

If restarting your activity requires that you recover large sets of data, re-establish a network connection, or perform other intensive operations, then a full restart due to a configuration change might be a slow user experience. Also, it might not be possible for you to completely restore your activity state with the [Bundle](#) that the

system saves for you with the `onSaveInstanceState()` callback—it is not designed to carry large objects (such as bitmaps) and the data within it must be serialized then deserialized, which can consume a lot of memory and make the configuration change slow. In such a situation, you can alleviate the burden of reinitializing your activity by retaining a stateful `Object` when your activity is restarted due to a configuration change.

To retain an object during a runtime configuration change:

1. Override the `onRetainNonConfigurationInstance()` method to return the object you would like to retain.
2. When your activity is created again, call `getLastNonConfigurationInstance()` to recover your object.

When the Android system shuts down your activity due to a configuration change, it calls `onRetainNonConfigurationInstance()` between the `onStop()` and `onDestroy()` callbacks. In your implementation of `onRetainNonConfigurationInstance()`, you can return any `Object` that you need in order to efficiently restore your state after the configuration change.

A scenario in which this can be valuable is if your application loads a lot of data from the web. If the user changes the orientation of the device and the activity restarts, your application must re-fetch the data, which could be slow. What you can do instead is implement `onRetainNonConfigurationInstance()` to return an object carrying your data and then retrieve the data when your activity starts again with `getLastNonConfigurationInstance()`. For example:

```
@Override  
public Object onRetainNonConfigurationInstance() {  
    final MyDataObject data = collectMyLoadedData();  
    return data;  
}
```

**Caution:** While you can return any object, you should never pass an object that is tied to the `Activity`, such as a `Drawable`, an `Adapter`, a `View` or any other object that's associated with a `Context`. If you do, it will leak all the views and resources of the original activity instance. (Leaking resources means that your application maintains a hold on them and they cannot be garbage-collected, so lots of memory can be lost.)

Then retrieve the data when your activity starts again:

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
  
    final MyDataObject data = (MyDataObject) getLastNonConfigurationInstance();  
    if (data == null) {  
        data = loadMyData();  
    }  
    ...  
}
```

In this case, `getLastNonConfigurationInstance()` returns the data saved by `onRetainNonConfigurationInstance()`. If data is null (which happens when the activity starts due to any reason other than a configuration change) then this code loads the data object from the original source.

# Handling the Configuration Change Yourself

If your application doesn't need to update resources during a specific configuration change *and* you have a performance limitation that requires you to avoid the activity restart, then you can declare that your activity handles the configuration change itself, which prevents the system from restarting your activity.

**Note:** Handling the configuration change yourself can make it much more difficult to use alternative resources, because the system does not automatically apply them for you. This technique should be considered a last resort when you must avoid restarts due to a configuration change and is not recommended for most applications.

To declare that your activity handles a configuration change, edit the appropriate [`<activity>`](#) element in your manifest file to include the [`android:configChanges`](#) attribute with a value that represents the configuration you want to handle. Possible values are listed in the documentation for the [`an-droid:configChanges`](#) attribute (the most commonly used values are "orientation" to prevent restarts when the screen orientation changes and "keyboardHidden" to prevent restarts when the keyboard availability changes). You can declare multiple configuration values in the attribute by separating them with a pipe | character.

For example, the following manifest code declares an activity that handles both the screen orientation change and keyboard availability change:

```
<activity android:name=".MyActivity"
          android:configChanges="orientation|keyboardHidden"
          android:label="@string/app_name">
```

Now, when one of these configurations change, MyActivity does not restart. Instead, the MyActivity receives a call to [`onConfigurationChanged\(\)`](#). This method is passed a [`Configuration`](#) object that specifies the new device configuration. By reading fields in the [`Configuration`](#), you can determine the new configuration and make appropriate changes by updating the resources used in your interface. At the time this method is called, your activity's [`Resources`](#) object is updated to return resources based on the new configuration, so you can easily reset elements of your UI without the system restarting your activity.

**Caution:** Beginning with Android 3.2 (API level 13), **the "screen size" also changes** when the device switches between portrait and landscape orientation. Thus, if you want to prevent runtime restarts due to orientation change when developing for API level 13 or higher (as declared by the [`minSdkVersion`](#) and [`targetSdkVersion`](#) attributes), you must include the "screenSize" value in addition to the "orientation" value. That is, you must declare `android:configChanges="orientation|screenSize"`. However, if your application targets API level 12 or lower, then your activity always handles this configuration change itself (this configuration change does not restart your activity, even when running on an Android 3.2 or higher device).

For example, the following [`onConfigurationChanged\(\)`](#) implementation checks the current device orientation:

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);

    // Checks the orientation of the screen
    if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
        Toast.makeText(this, "landscape", Toast.LENGTH_SHORT).show();
    } else if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT) {
        Toast.makeText(this, "portrait", Toast.LENGTH_SHORT).show();
    }
}
```

```
    }  
}
```

The [Configuration](#) object represents all of the current configurations, not just the ones that have changed. Most of the time, you won't care exactly how the configuration has changed and can simply re-assign all your resources that provide alternatives to the configuration that you're handling. For example, because the [Re-resources](#) object is now updated, you can reset any [ImageViews](#) with [setImageResource\(\)](#) and the appropriate resource for the new configuration is used (as described in [Providing Resources](#)).

Notice that the values from the [Configuration](#) fields are integers that are matched to specific constants from the [Configuration](#) class. For documentation about which constants to use with each field, refer to the appropriate field in the [Configuration](#) reference.

**Remember:** When you declare your activity to handle a configuration change, you are responsible for resetting any elements for which you provide alternatives. If you declare your activity to handle the orientation change and have images that should change between landscape and portrait, you must re-assign each resource to each element during [onConfigurationChanged\(\)](#).

If you don't need to update your application based on these configuration changes, you can instead *not* implement [onConfigurationChanged\(\)](#). In which case, all of the resources used before the configuration change are still used and you've only avoided the restart of your activity. However, your application should always be able to shutdown and restart with its previous state intact, so you should not consider this technique an escape from retaining your state during normal activity lifecycle. Not only because there are other configuration changes that you cannot prevent from restarting your application, but also because you should handle events such as when the user leaves your application and it gets destroyed before the user returns to it.

For more about which configuration changes you can handle in your activity, see the [an-droid:configChanges](#) documentation and the [Configuration](#) class.

# Localizing with Resources

## Quickview

- Use resource sets to create a localized app.
- Android loads the correct resource set for the user's language and locale.
- If localized resources are not available, Android loads your default resources.

## In this document

1. [Overview: Resource-Switching in Android](#)
2. [Using Resources for Localization](#)
3. [Localization Strategies](#)
4. [Testing Localized Applications](#)

## See also

1. [Localization Checklist](#)
2. [Providing Resources](#)
3. [Layouts](#)
4. [Activity Lifecycle](#)

Android will run on many devices in many regions. To reach the most users, your application should handle text, audio files, numbers, currency, and graphics in ways appropriate to the locales where your application will be used.

This document describes best practices for localizing Android applications. The principles apply whether you are developing your application using ADT with Eclipse, Ant-based tools, or any other IDE.

You should already have a working knowledge of Java and be familiar with Android resource loading, the declaration of user interface elements in XML, development considerations such as Activity lifecycle, and general principles of internationalization and localization.

It is good practice to use the Android resource framework to separate the localized aspects of your application as much as possible from the core Java functionality:

- You can put most or all of the *contents* of your application's user interface into resource files, as described in this document and in [Providing Resources](#).
- The *behavior* of the user interface, on the other hand, is driven by your Java code. For example, if users input data that needs to be formatted or sorted differently depending on locale, then you would use Java to handle the data programmatically. This document does not cover how to localize your Java code.

For a short guide to localizing strings in your app, see the training lesson, [Supporting Different Languages](#).

## Overview: Resource-Switching in Android

Resources are text strings, layouts, sounds, graphics, and any other static data that your Android application needs. An application can include multiple sets of resources, each customized for a different device configuration. When a user runs the application, Android automatically selects and loads the resources that best match the device.

(This document focuses on localization and locale. For a complete description of resource-switching and all the types of configurations that you can specify — screen orientation, touchscreen type, and so on — see [Providing Alternative Resources](#).)

## When you write your application:

You create a set of default resources, plus alternatives to be used in different locales.



## When a user runs your application:

The Android system selects which resources to load, based on the device's locale.

When you write your application, you create default and alternative resources for your application to use. To create resources, you place files within specially named subdirectories of the project's `res/` directory.

## Why Default Resources Are Important

Whenever the application runs in a locale for which you have not provided locale-specific text, Android will load the default strings from `res/values/strings.xml`. If this default file is absent, or if it is missing a string that your application needs, then your application will not run and will show an error. The example below illustrates what can happen when the default text file is incomplete.

*Example:*

An application's Java code refers to just two strings, `text_a` and `text_b`. This application includes a localized resource file (`res/values-en/strings.xml`) that defines `text_a` and `text_b` in English. This application also includes a default resource file (`res/values/strings.xml`) that includes a definition for `text_a`, but not for `text_b`:

- This application might compile without a problem. An IDE such as Eclipse will not highlight any errors if a resource is missing.
- When this application is launched on a device with locale set to English, the application might run without a problem, because `res/values-en/strings.xml` contains both of the needed text strings.
- However, **the user will see an error message and a Force Close button** when this application is launched on a device set to a language other than English. The application will not load.

To prevent this situation, make sure that a `res/values/strings.xml` file exists and that it defines every needed string. The situation applies to all types of resources, not just strings: You need to create a set of default resource files containing all the resources that your application calls upon — layouts, drawables, animations, etc. For information about testing, see [Testing for Default Resources](#).

## Using Resources for Localization

### How to Create Default Resources

Put the application's default text in a file with the following location and name:

`res/values/strings.xml` (required directory)

The text strings in `res/values/strings.xml` should use the default language, which is the language that you expect most of your application's users to speak.

The default resource set must also include any default drawables and layouts, and can include other types of resources such as animations.

`res/drawable/`(required directory holding at least one graphic file, for the application's icon on Google Play)

`res/layout/` (required directory holding an XML file that defines the default layout)

`res/anim/` (required if you have any `res/anim-<qualifiers>` folders)

`res/xml/` (required if you have any `res/xml-<qualifiers>` folders)

`res/raw/` (required if you have any `res/raw-<qualifiers>` folders)

**Tip:** In your code, examine each reference to an Android resource. Make sure that a default resource is defined for each one. Also make sure that the default string file is complete: A *localized* string file can contain a subset of the strings, but the *default* string file must contain them all.

## How to Create Alternative Resources

A large part of localizing an application is providing alternative text for different languages. In some cases you will also provide alternative graphics, sounds, layouts, and other locale-specific resources.

An application can specify many `res/<qualifiers>/` directories, each with different qualifiers. To create an alternative resource for a different locale, you use a qualifier that specifies a language or a language-region combination. (The name of a resource directory must conform to the naming scheme described in [Providing Alternative Resources](#), or else it will not compile.)

*Example:*

Suppose that your application's default language is English. Suppose also that you want to localize all the text in your application to French, and most of the text in your application (everything except the application's title) to Japanese. In this case, you could create three alternative `strings.xml` files, each stored in a locale-specific resource directory:

1. `res/values/strings.xml`  
Contains English text for all the strings that the application uses, including text for a string named `title`.
2. `res/values-fr/strings.xml`  
Contain French text for all the strings, including `title`.
3. `res/values-ja/strings.xml`  
Contain Japanese text for all the strings *except* `title`.

If your Java code refers to `R.string.title`, here is what will happen at runtime:

- If the device is set to any language other than French, Android will load `title` from the `res/values/strings.xml` file.
- If the device is set to French, Android will load `title` from the `res/values-fr/strings.xml` file.

Notice that if the device is set to Japanese, Android will look for `title` in the `res/values-ja/strings.xml` file. But because no such string is included in that file, Android will fall back to the default, and will load `title` in English from the `res/values/strings.xml` file.

## Which Resources Take Precedence?

If multiple resource files match a device's configuration, Android follows a set of rules in deciding which file to use. Among the qualifiers that can be specified in a resource directory name, **locale almost always takes precedence**.

*Example:*

Assume that an application includes a default set of graphics and two other sets of graphics, each optimized for a different device setup:

- `res/drawable/`  
Contains default graphics.
- `res/drawable-small-land-stylus/`  
Contains graphics optimized for use with a device that expects input from a stylus and has a QVGA low-density screen in landscape orientation.
- `res/drawable-ja/`  
Contains graphics optimized for use with Japanese.

If the application runs on a device that is configured to use Japanese, Android will load graphics from `res/drawable-ja/`, even if the device happens to be one that expects input from a stylus and has a QVGA low-density screen in landscape orientation.

**Exception:** The only qualifiers that take precedence over locale in the selection process are MCC and MNC (mobile country code and mobile network code).

*Example:*

Assume that you have the following situation:

- The application code calls for `R.string.text_a`
- Two relevant resource files are available:
  - `res/values-mcc404/strings.xml`, which includes `text_a` in the application's default language, in this case English.
  - `res/values-hi/strings.xml`, which includes `text_a` in Hindi.
- The application is running on a device that has the following configuration:
  - The SIM card is connected to a mobile network in India (MCC 404).
  - The language is set to Hindi (`hi`).

Android will load `text_a` from `res/values-mcc404/strings.xml` (in English), even if the device is configured for Hindi. That is because in the resource-selection process, Android will prefer an MCC match over a language match.

The selection process is not always as straightforward as these examples suggest. Please read [How Android Finds the Best-matching Resource](#) for a more nuanced description of the process. All the qualifiers are described and listed in order of precedence in [Table 2 of Providing Alternative Resources](#).

## Referring to Resources in Java

In your application's Java code, you refer to resources using the syntax `R.resource_type.re-source_name` or `android.R.resource_type.resource_name`. For more about this, see [Accessing Resources](#).

# Localization Strategies

## Design your application to work in any locale

You cannot assume anything about the device on which a user will run your application. The device might have hardware that you were not anticipating, or it might be set to a locale that you did not plan for or that you cannot test. Design your application so that it will function normally or fail gracefully no matter what device it runs on.

**Important:** Make sure that your application includes a full set of default resources.

Make sure to include `res/drawable/` and a `res/values/` folders (without any additional modifiers in the folder names) that contain all the images and text that your application will need.

If an application is missing even one default resource, it will not run on a device that is set to an unsupported locale. For example, the `res/values/strings.xml` default file might lack one string that the application needs: When the application runs in an unsupported locale and attempts to load `res/values/strings.xml`, the user will see an error message and a Force Close button. An IDE such as Eclipse will not highlight this kind of error, and you will not see the problem when you test the application on a device or emulator that is set to a supported locale.

For more information, see [Testing for Default Resources](#).

## Design a flexible layout

If you need to rearrange your layout to fit a certain language (for example German with its long words), you can create an alternative layout for that language (for example `res/layout-de/main.xml`). However, doing this can make your application harder to maintain. It is better to create a single layout that is more flexible.

Another typical situation is a language that requires something different in its layout. For example, you might have a contact form that should include two name fields when the application runs in Japanese, but three name fields when the application runs in some other language. You could handle this in either of two ways:

- Create one layout with a field that you can programmatically enable or disable, based on the language, or
- Have the main layout include another layout that includes the changeable field. The second layout can have different configurations for different languages.

## Avoid creating more resource files and text strings than you need

You probably do not need to create a locale-specific alternative for every resource in your application. For example, the layout defined in the `res/layout/main.xml` file might work in any locale, in which case there would be no need to create any alternative layout files.

Also, you might not need to create alternative text for every string. For example, assume the following:

- Your application's default language is American English. Every string that the application uses is defined, using American English spellings, in `res/values/strings.xml`.
- For a few important phrases, you want to provide British English spelling. You want these alternative strings to be used when your application runs on a device in the United Kingdom.

To do this, you could create a small file called `res/values-en-rGB/strings.xml` that includes only the strings that should be different when the application runs in the U.K. For all the rest of the strings, the application will fall back to the defaults and use what is defined in `res/values/strings.xml`.

## Use the Android Context object for manual locale lookup

You can look up the locale using the [Context](#) object that Android makes available:

```
String locale = context.getResources().getConfiguration().locale.getDisplayName();
```

# Testing Localized Applications

## Testing on a Device

Keep in mind that the device you are testing may be significantly different from the devices available to consumers in other geographies. The locales available on your device may differ from those available on other devices. Also, the resolution and density of the device screen may differ, which could affect the display of strings and drawables in your UI.

To change the locale on a device, use the Settings application (Home > Menu > Settings > Locale & text > Select locale).

## Testing on an Emulator

For details about using the emulator, see See [Android Emulator](#).

## Creating and using a custom locale

A "custom" locale is a language/region combination that the Android system image does not explicitly support. (For a list of supported locales in Android platforms see the Version Notes in the [SDK](#) tab). You can test how your application will run in a custom locale by creating a custom locale in the emulator. There are two ways to do this:

- Use the Custom Locale application, which is accessible from the Application tab. (After you create a custom locale, switch to it by pressing and holding the locale name.)
- Change to a custom locale from the adb shell, as described below.

When you set the emulator to a locale that is not available in the Android system image, the system itself will display in its default language. Your application, however, should localize properly.

## Changing the emulator locale from the adb shell

To change the locale in the emulator by using the adb shell.

1. Pick the locale you want to test and determine its language and region codes, for example `f_r` for French and `CA` for Canada.
2. Launch an emulator.
3. From a command-line shell on the host computer, run the following command:  
`adb shell`  
or if you have a device attached, specify that you want the emulator by adding the `-e` option:  
`adb -e shell`

4. At the adb shell prompt (#), run this command:

```
setprop persist.sys.language [language code];setprop persist.sys.country [country code];stop;sleep 5;start  
Replace bracketed sections with the appropriate codes from Step 1.
```

For instance, to test in Canadian French:

```
setprop persist.sys.language fr;setprop persist.sys.country CA;stop;sleep 5;start
```

This will cause the emulator to restart. (It will look like a full reboot, but it is not.) Once the Home screen appears again, re-launch your application (for example, click the Run icon in Eclipse), and the application will launch with the new locale.

## Testing for Default Resources

Here's how to test whether an application includes every string resource that it needs:

1. Set the emulator or device to a language that your application does not support. For example, if the application has French strings in `res/values-fr/` but does not have any Spanish strings in `res/values-es/`, then set the emulator's locale to Spanish. (You can use the Custom Locale application to set the emulator to an unsupported locale.)
2. Run the application.
3. If the application shows an error message and a Force Close button, it might be looking for a string that is not available. Make sure that your `res/values/strings.xml` file includes a definition for every string that the application uses.

If the test is successful, repeat it for other types of configurations. For example, if the application has a layout file called `res/layout-land/main.xml` but does not contain a file called `res/layout-port/main.xml`, then set the emulator or device to portrait orientation and see if the application will run.

## Localization Checklist

For an overview of the process of localizing an Android application, see the [Localization Checklist](#).

# Resource Types

## See also

1. [Providing Resources](#)
2. [Accessing Resources](#)

Each of the documents in this section describe the usage, format and syntax for a certain type of application resource that you can provide in your resources directory (`res/`).

Here's a brief summary of each resource type:

### [Animation Resources](#)

Define pre-determined animations.

Tween animations are saved in `res/anim/` and accessed from the `R.anim` class.

Frame animations are saved in `res/drawable/` and accessed from the `R.drawable` class.

### [Color State List Resource](#)

Define a color resources that changes based on the View state.

Saved in `res/color/` and accessed from the `R.color` class.

### [Drawable Resources](#)

Define various graphics with bitmaps or XML.

Saved in `res/drawable/` and accessed from the `R.drawable` class.

### [Layout Resource](#)

Define the layout for your application UI.

Saved in `res/layout/` and accessed from the `R.layout` class.

### [Menu Resource](#)

Define the contents of your application menus.

Saved in `res/menu/` and accessed from the `R.menu` class.

### [String Resources](#)

Define strings, string arrays, and plurals (and include string formatting and styling).

Saved in `res/values/` and accessed from the `R.string`, `R.array`, and `R.plurals` classes.

### [Style Resource](#)

Define the look and format for UI elements.

Saved in `res/values/` and accessed from the `R.style` class.

### [More Resource Types](#)

Define values such as booleans, integers, dimensions, colors, and other arrays.

Saved in `res/values/` but each accessed from unique `R` sub-classes (such as `R.bool`, `R.integer`, `R.dimen`, etc.).

# Animation Resources

## In this document

1. [Property Animation](#)
2. [View Animation](#)
  1. [Tween animation](#)
  2. [Frame animation](#)

## See also

1. [View Animation](#)
2. [Property Animation](#)

An animation resource can define one of two types of animations:

### [Property Animation](#)

Creates an animation by modifying an object's property values over a set period of time with an [Animator](#).

### [View Animation](#)

There are two types of animations that you can do with the view animation framework:

- [Tween animation](#): Creates an animation by performing a series of transformations on a single image with an [Animation](#)
- [Frame animation](#): or creates an animation by showing a sequence of images in order with an [AnimationDrawable](#).

## Property Animation

An animation defined in XML that modifies properties of the target object, such as background color or alpha value, over a set amount of time.

### file location:

`res/animator/filename.xml`

The filename will be used as the resource ID.

### compiled resource datatype:

Resource pointer to a [ValueAnimator](#), [ObjectAnimator](#), or [AnimatorSet](#).

### resource reference:

In Java: `R.animator.filename`

In XML: `@[package:]animator/filename`

### syntax:

```
<set  
    android:ordering=["together" | "sequentially"]>  
  
    <objectAnimator  
        android:propertyName="string"
```

```

        android:duration="int"
        android:valueFrom="float | int | color"
        android:valueTo="float | int | color"
        android:startOffset="int"
        android:repeatCount="int"
        android:repeatMode=["repeat" | "reverse"]
        android:valueType=["intType" | "floatType"]/>

<animator>
    android:duration="int"
    android:valueFrom="float | int | color"
    android:valueTo="float | int | color"
    android:startOffset="int"
    android:repeatCount="int"
    android:repeatMode=["repeat" | "reverse"]
    android:valueType=["intType" | "floatType"]/>

<set>
    ...
</set>
</set>

```

The file must have a single root element: either `<set>`, `<objectAnimator>`, or `<valueAnimator>`. You can group animation elements together inside the `<set>` element, including other `<set>` elements.

## elements:

### `<set>`

A container that holds other animation elements (`<objectAnimator>`, `<valueAnimator>`, or other `<set>` elements). Represents an [AnimatorSet](#).

You can specify nested `<set>` tags to further group animations together. Each `<set>` can define its own `ordering` attribute.

attributes:

#### `android:ordering`

*Keyword.* Specifies the play ordering of animations in this set.

Value	Description
sequentially	Play animations in this set sequentially
together (default)	Play animations in this set at the same time.

### `<objectAnimator>`

Animates a specific property of an object over a specific amount of time. Represents an [ObjectAnimator](#).

attributes:

#### `android:propertyName`

*String. Required.* The object's property to animate, referenced by its name. For example you can specify "alpha" or "backgroundColor" for a View object. The `objectAnimator` element does not expose a `target` attribute, however, so you cannot set the object to animate in the

XML declaration. You have to inflate your animation XML resource by calling [loadAnimator\(\)](#) and call [setTarget\(\)](#) to set the target object that contains this property.

#### **android:valueTo**

*float, int, or color.* **Required.** The value where the animated property ends. Colors are represented as six digit hexadecimal numbers (for example, #333333).

#### **android:valueFrom**

*float, int, or color.* The value where the animated property starts. If not specified, the animation starts at the value obtained by the property's get method. Colors are represented as six digit hexadecimal numbers (for example, #333333).

#### **android:duration**

*int.* The time in milliseconds of the animation. 300 milliseconds is the default.

#### **android:startOffset**

*int.* The amount of milliseconds the animation delays after [start\(\)](#) is called.

#### **android:repeatCount**

*int.* How many times to repeat an animation. Set to "-1" to infinitely repeat or to a positive integer. For example, a value of "1" means that the animation is repeated once after the initial run of the animation, so the animation plays a total of two times. The default value is "0", which means no repetition.

#### **android:repeatMode**

*int.* How an animation behaves when it reaches the end of the animation. `android:repeatCount` must be set to a positive integer or "-1" for this attribute to have an effect. Set to "reverse" to have the animation reverse direction with each iteration or "repeat" to have the animation loop from the beginning each time.

#### **android:valueType**

*Keyword.* Do not specify this attribute if the value is a color. The animation framework automatically handles color values

##### **Value**

##### **Description**

`intType` Specifies that the animated values are integers

`floatType` (default) Specifies that the animated values are floats

## **<animator>**

Performs an animation over a specified amount of time. Represents a [ValueAnimator](#).

attributes:

#### **android:valueTo**

*float, int, or color.* **Required.** The value where the animation ends. Colors are represented as six digit hexadecimal numbers (for example, #333333).

#### **android:valueFrom**

*float, int, or color.* **Required.** The value where the animation starts. Colors are represented as six digit hexadecimal numbers (for example, #333333).

#### **android:duration**

*int.* The time in milliseconds of the animation. 300ms is the default.

**android:startOffset**

*int.* The amount of milliseconds the animation delays after [start\(\)](#) is called.

**android:repeatCount**

*int.* How many times to repeat an animation. Set to "-1" to infinitely repeat or to a positive integer. For example, a value of "1" means that the animation is repeated once after the initial run of the animation, so the animation plays a total of two times. The default value is "0", which means no repetition.

**android:repeatMode**

*int.* How an animation behaves when it reaches the end of the animation. android:repeatCount must be set to a positive integer or "-1" for this attribute to have an effect. Set to "reverse" to have the animation reverse direction with each iteration or "repeat" to have the animation loop from the beginning each time.

**android:valueType**

*Keyword.* Do not specify this attribute if the value is a color. The animation framework automatically handles color values.

**Value****Description**

intType Specifies that the animated values are integers

floatType (default) Specifies that the animated values are floats

**example:**

XML file saved at res/animator/property\_animator.xml:

```
<set android:ordering="sequentially">
    <set>
        <objectAnimator
            android:propertyName="x"
            android:duration="500"
            android:valueTo="400"
            android:valueType="intType"/>
        <objectAnimator
            android:propertyName="y"
            android:duration="500"
            android:valueTo="300"
            android:valueType="intType"/>
    </set>
    <objectAnimator
        android:propertyName="alpha"
        android:duration="500"
        android:valueTo="1f"/>
</set>
```

In order to run this animation, you must inflate the XML resources in your code to an [AnimatorSet](#) object, and then set the target objects for all of the animations before starting the animation set. Calling [setTarget\(\)](#) sets a single target object for all children of the [AnimatorSet](#) as a convenience. The following code shows how to do this:

```
AnimatorSet set = (AnimatorSet) AnimatorInflater.loadAnimator(myContext,
    R.anim.property_animator);
set.setTarget(myObject);
set.start();
```

**see also:**

- [Property Animation](#)
- [API Demos](#) for examples on how to use the property animation system.

## View Animation

The view animation framework supports both tween and frame by frame animations, which can both be declared in XML. The following sections describe how to use both methods.

### Tween animation

An animation defined in XML that performs transitions such as rotating, fading, moving, and stretching on a graphic.

**file location:**

`res/anim/filename.xml`

The filename will be used as the resource ID.

**compiled resource datatype:**

Resource pointer to an [Animation](#).

**resource reference:**

In Java: `R.anim.filename`

In XML: `@[package:]anim/filename`

**syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@[package:]anim/interpolator_resource"
    android:shareInterpolator=["true" | "false"] >
    <alpha
        android:fromAlpha="float"
        android:toAlpha="float" />
    <scale
        android:fromXScale="float"
        android:toXScale="float"
        android:fromYScale="float"
        android:toYScale="float"
        android:pivotX="float"
        android:pivotY="float" />
    <translate
        android:fromXDelta="float"
        android:toXDelta="float"
        android:fromYDelta="float"
        android:toYDelta="float" />
    <rotate
        android:fromDegrees="float"
        android:toDegrees="float"
        android:pivotX="float"
        android:pivotY="float" />
<set
```

```
</set>  
</set>
```

The file must have a single root element: either an `<alpha>`, `<scale>`, `<translate>`, `<rotate>`, or `<set>` element that holds a group (or groups) of other animation elements (even nested `<set>` elements).

## elements:

### `<set>`

A container that holds other animation elements (`<alpha>`, `<scale>`, `<translate>`, `<rotate>`) or other `<set>` elements. Represents an [AnimationSet](#).

attributes:

#### `android:interpolator`

*Interpolator resource.* An [Interpolator](#) to apply on the animation. The value must be a reference to a resource that specifies an interpolator (not an interpolator class name). There are default interpolator resources available from the platform or you can create your own interpolator resource. See the discussion below for more about [Interpolators](#).

#### `android:shareInterpolator`

*Boolean.* "true" if you want to share the same interpolator among all child elements.

### `<alpha>`

A fade-in or fade-out animation. Represents an [AlphaAnimation](#).

attributes:

#### `android:fromAlpha`

*Float.* Starting opacity offset, where 0.0 is transparent and 1.0 is opaque.

#### `android:toAlpha`

*Float.* Ending opacity offset, where 0.0 is transparent and 1.0 is opaque.

For more attributes supported by `<alpha>`, see the [Animation](#) class reference (of which, all XML attributes are inherited by this element).

### `<scale>`

A resizing animation. You can specify the center point of the image from which it grows outward (or inward) by specifying `pivotX` and `pivotY`. For example, if these values are 0, 0 (top-left corner), all growth will be down and to the right. Represents a [ScaleAnimation](#).

attributes:

#### `android:fromXScale`

*Float.* Starting X size offset, where 1.0 is no change.

#### `android:toXScale`

*Float.* Ending X size offset, where 1.0 is no change.

#### `android:fromYScale`

*Float.* Starting Y size offset, where 1.0 is no change.

#### `android:toYScale`

*Float.* Ending Y size offset, where 1.0 is no change.

**android:pivotX**

*Float.* The X coordinate to remain fixed when the object is scaled.

**android:pivotY**

*Float.* The Y coordinate to remain fixed when the object is scaled.

For more attributes supported by <scale>, see the [Animation](#) class reference (of which, all XML attributes are inherited by this element).

**<translate>**

A vertical and/or horizontal motion. Supports the following attributes in any of the following three formats: values from -100 to 100 ending with "%", indicating a percentage relative to itself; values from -100 to 100 ending in "%p", indicating a percentage relative to its parent; a float value with no suffix, indicating an absolute value. Represents a [TranslateAnimation](#).

attributes:

**android:fromXDelta**

*Float or percentage.* Starting X offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element width (such as "5%"), or in percentage relative to the parent width (such as "5%p").

**android:toXDelta**

*Float or percentage.* Ending X offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element width (such as "5%"), or in percentage relative to the parent width (such as "5%p").

**android:fromYDelta**

*Float or percentage.* Starting Y offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element height (such as "5%"), or in percentage relative to the parent height (such as "5%p").

**android:toYDelta**

*Float or percentage.* Ending Y offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element height (such as "5%"), or in percentage relative to the parent height (such as "5%p").

For more attributes supported by <translate>, see the [Animation](#) class reference (of which, all XML attributes are inherited by this element).

**<rotate>**

A rotation animation. Represents a [RotateAnimation](#).

attributes:

**android:fromDegrees**

*Float.* Starting angular position, in degrees.

**android:toDegrees**

*Float.* Ending angular position, in degrees.

### **android:pivotX**

*Float or percentage.* The X coordinate of the center of rotation. Expressed either: in pixels relative to the object's left edge (such as "5"), in percentage relative to the object's left edge (such as "5%"), or in percentage relative to the parent container's left edge (such as "5%p").

### **android:pivotY**

*Float or percentage.* The Y coordinate of the center of rotation. Expressed either: in pixels relative to the object's top edge (such as "5"), in percentage relative to the object's top edge (such as "5%"), or in percentage relative to the parent container's top edge (such as "5%p").

For more attributes supported by <rotate>, see the [Animation](#) class reference (of which, all XML attributes are inherited by this element).

### **example:**

XML file saved at res/anim/hyperspace\_jump.xml:

```
<set xmlns:android="http://schemas.android.com/apk/res/android"  
      android:shareInterpolator="false">  
    <scale  
        android:interpolator="@android:anim/accelerate_decelerate_interpolator"  
        android:fromXScale="1.0"  
        android:toXScale="1.4"  
        android:fromYScale="1.0"  
        android:toYScale="0.6"  
        android:pivotX="50%"  
        android:pivotY="50%"  
        android:fillAfter="false"  
        android:duration="700" />  
    <set  
        android:interpolator="@android:anim/accelerate_interpolator"  
        android:startOffset="700">  
      <scale  
          android:fromXScale="1.4"  
          android:toXScale="0.0"  
          android:fromYScale="0.6"  
          android:toYScale="0.0"  
          android:pivotX="50%"  
          android:pivotY="50%"  
          android:duration="400" />  
      <rotate  
          android:fromDegrees="0"  
          android:toDegrees="-45"  
          android:toYScale="0.0"  
          android:pivotX="50%"  
          android:pivotY="50%"  
          android:duration="400" />  
    </set>  
</set>
```

This application code will apply the animation to an [ImageView](#) and start the animation:

```
ImageView image = (ImageView) findViewById(R.id.image);  
Animation hyperspaceJump = AnimationUtils.loadAnimation(this, R.anim.hyperspaceJump);  
image.startAnimation(hyperspaceJump);
```

**see also:**

- [2D Graphics: Tween Animation](#)

## Interpolators

An interpolator is an animation modifier defined in XML that affects the rate of change in an animation. This allows your existing animation effects to be accelerated, decelerated, repeated, bounced, etc.

An interpolator is applied to an animation element with the `android:interpolator` attribute, the value of which is a reference to an interpolator resource.

All interpolators available in Android are subclasses of the [Interpolator](#) class. For each interpolator class, Android includes a public resource you can reference in order to apply the interpolator to an animation using the `android:interpolator` attribute. The following table specifies the resource to use for each interpolator:

Interpolator class	Resource ID
<a href="#">AccelerateDecelerateInterpolator</a>	<code>@android:anim/accelerate_decelerate_interpolator</code>
<a href="#">AccelerateInterpolator</a>	<code>@android:anim/accelerate_interpolator</code>
<a href="#">AnticipateInterpolator</a>	<code>@android:anim/anticipate_interpolator</code>
<a href="#">AnticipateOvershootInterpolator</a>	<code>@android:anim/anticipate_overshoot_interpolator</code>
<a href="#">BounceInterpolator</a>	<code>@android:anim/bounce_interpolator</code>
<a href="#">CycleInterpolator</a>	<code>@android:anim/cycle_interpolator</code>
<a href="#">DecelerateInterpolator</a>	<code>@android:anim/decelerate_interpolator</code>
<a href="#">LinearInterpolator</a>	<code>@android:anim/linear_interpolator</code>
<a href="#">OvershootInterpolator</a>	<code>@android:anim/overshoot_interpolator</code>

Here's how you can apply one of these with the `android:interpolator` attribute:

```
<set android:interpolator="@android:anim/accelerate_interpolator">
    ...
</set>
```

## Custom interpolators

If you're not satisfied with the interpolators provided by the platform (listed in the table above), you can create a custom interpolator resource with modified attributes. For example, you can adjust the rate of acceleration for the [AnticipateInterpolator](#), or adjust the number of cycles for the [CycleInterpolator](#). In order to do so, you need to create your own interpolator resource in an XML file.

**file location:**

`res/anim/filename.xml`

The filename will be used as the resource ID.

**compiled resource datatype:**

Resource pointer to the corresponding interpolator object.

**resource reference:**

In XML: `@[package:]anim/filename`

## syntax:

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

If you don't apply any attributes, then your interpolator will function exactly the same as those provided by the platform (listed in the table above).

## elements:

Notice that each [Interpolator](#) implementation, when defined in XML, begins its name in lowercase.

### **<accelerateDecelerateInterpolator>**

The rate of change starts and ends slowly but accelerates through the middle.

No attributes.

### **<accelerateInterpolator>**

The rate of change starts out slowly, then accelerates.

attributes:

#### **android:factor**

*Float*. The acceleration rate (default is 1).

### **<anticipateInterpolator>**

The change starts backward then flings forward.

attributes:

#### **android:tension**

*Float*. The amount of tension to apply (default is 2).

### **<anticipateOvershootInterpolator>**

The change starts backward, flings forward and overshoots the target value, then settles at the final value.

attributes:

#### **android:tension**

*Float*. The amount of tension to apply (default is 2).

#### **android:extraTension**

*Float*. The amount by which to multiply the tension (default is 1.5).

### **<bounceInterpolator>**

The change bounces at the end.

No attributes

### **<cycleInterpolator>**

Repeats the animation for a specified number of cycles. The rate of change follows a sinusoidal pattern.

attributes:

**android:cycles**

*Integer.* The number of cycles (default is 1).

**<decelerateInterpolator>**

The rate of change starts out quickly, then decelerates.

attributes:

**android:factor**

*Float.* The deceleration rate (default is 1).

**<linearInterpolator>**

The rate of change is constant.

No attributes.

**<overshootInterpolator>**

The change flings forward and overshoots the last value, then comes back.

attributes:

**android:tension**

*Float.* The amount of tension to apply (default is 2).

**example:**

XML file saved at `res/anim/my_overshoot_interpolator.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<overshootInterpolator xmlns:android="http://schemas.android.com/apk/res/android"
    android:tension="7.0"
/>
```

This animation XML will apply the interpolator:

```
<scale xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@anim/my_overshoot_interpolator"
    android:fromXScale="1.0"
    android:toXScale="3.0"
    android:fromYScale="1.0"
    android:toYScale="3.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:duration="700" />
```

## Frame animation

An animation defined in XML that shows a sequence of images in order (like a film).

**file location:**

`res/drawable/filename.xml`

The filename will be used as the resource ID.

## **compiled resource datatype:**

Resource pointer to an [AnimationDrawable](#).

## **resource reference:**

In Java: R.drawable.filename

In XML: @ [package:]drawable.filename

## **syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot=["true" | "false"] >
    <item
        android:drawable="@ [package:]drawable/drawable_resource_name"
        android:duration="integer" />
</animation-list>
```

## **elements:**

### **<animation-list>**

**Required.** This must be the root element. Contains one or more <item> elements.

attributes:

#### **android:oneshot**

*Boolean.* "true" if you want to perform the animation once; "false" to loop the animation.

### **<item>**

A single frame of animation. Must be a child of a <animation-list> element.

attributes:

#### **android:drawable**

*Drawable resource.* The drawable to use for this frame.

#### **android:duration**

*Integer.* The duration to show this frame, in milliseconds.

## **example:**

### **XML file saved at res/anim/rocket.xml:**

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
    <item android:drawable="@drawable/rocket_thrust1" android:duration="2000" />
    <item android:drawable="@drawable/rocket_thrust2" android:duration="2000" />
    <item android:drawable="@drawable/rocket_thrust3" android:duration="2000" />
</animation-list>
```

## **This application code will set the animation as the background for a View, then play the animation:**

```
ImageView rocketImage = (ImageView) findViewById(R.id.rocket_image);
rocketImage.setBackgroundResource(R.drawable.rocket_thrust);

rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
rocketAnimation.start();
```

**see also:**

- [2D Graphics: Frame Animation](#)

# Color State List Resource

## See also

1. [Color \(simple value\)](#)

A [ColorStateList](#) is an object you can define in XML that you can apply as a color, but will actually change colors, depending on the state of the [View](#) object to which it is applied. For example, a [Button](#) widget can exist in one of several different states (pressed, focused, or neither) and, using a color state list, you can provide a different color during each state.

You can describe the state list in an XML file. Each color is defined in an `<item>` element inside a single `<selector>` element. Each `<item>` uses various attributes to describe the state in which it should be used.

During each state change, the state list is traversed top to bottom and the first item that matches the current state will be used—the selection is *not* based on the "best match," but simply the first item that meets the minimum criteria of the state.

**Note:** If you want to provide a static color resource, use a simple [Color](#) value.

### file location:

`res/color/filename.xml`

The filename will be used as the resource ID.

### compiled resource datatype:

Resource pointer to a [ColorStateList](#).

### resource reference:

In Java: `R.color.filename`

In XML: `@[package:]color/filename`

### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:color="hex_color"
        android:state_pressed=["true" | "false"]
        android:state_focused=["true" | "false"]
        android:state_selected=["true" | "false"]
        android:state_checkable=["true" | "false"]
        android:state_checked=["true" | "false"]
        android:state_enabled=["true" | "false"]
        android:state_window_focused=["true" | "false"] />
</selector>
```

### elements:

`<selector>`

**Required.** This must be the root element. Contains one or more `<item>` elements.

attributes:

```
xmlns:android
```

*String. Required.* Defines the XML namespace, which must be "http://schemas.android.com/apk/res/android".

## <item>

Defines a color to use during certain states, as described by its attributes. Must be a child of a <selector> element.

attributes:

**android:color**

*Hexadeximal color. Required.* The color is specified with an RGB value and optional alpha channel.

The value always begins with a pound (#) character and then followed by the Alpha-Red-Green-Blue information in one of the following formats:

- #RGB
- #ARGB
- #RRGGBB
- #AARRGGBB

**android:state\_pressed**

*Boolean.* "true" if this item should be used when the object is pressed (such as when a button is touched(clicked)); "false" if this item should be used in the default, non-pressed state.

**android:state\_focused**

*Boolean.* "true" if this item should be used when the object is focused (such as when a button is highlighted using the trackball/d-pad); "false" if this item should be used in the default, non-focused state.

**android:state\_selected**

*Boolean.* "true" if this item should be used when the object is selected (such as when a tab is opened); "false" if this item should be used when the object is not selected.

**android:state\_checkable**

*Boolean.* "true" if this item should be used when the object is checkable; "false" if this item should be used when the object is not checkable. (Only useful if the object can transition between a checkable and non-checkable widget.)

**android:state\_checked**

*Boolean.* "true" if this item should be used when the object is checked; "false" if it should be used when the object is un-checked.

**android:state\_enabled**

*Boolean.* "true" if this item should be used when the object is enabled (capable of receiving touch/click events); "false" if it should be used when the object is disabled.

**android:state\_window\_focused**

*Boolean.* "true" if this item should be used when the application window has focus (the application is in the foreground), "false" if this item should be used when the application window does not have focus (for example, if the notification shade is pulled down or a dialog appears).

**Note:** Remember that the first item in the state list that matches the current state of the object will be applied. So if the first item in the list contains none of the state attributes above, then it will be applied

every time, which is why your default value should always be last (as demonstrated in the following example).

**example:**

XML file saved at `res/color/button_text.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
          android:color="#ffff0000"/> <!-- pressed -->
    <item android:state_focused="true"
          android:color="#ff0000ff"/> <!-- focused -->
    <item android:color="#ff000000"/> <!-- default -->
</selector>
```

This layout XML will apply the color list to a View:

```
<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    android:textColor="@color/button_text" />
```

**see also:**

- [Color \(simple value\)](#)
- [ColorStateList](#)
- [State List Drawable](#)

# Drawable Resources

## See also

1. [2D Graphics](#)

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

### [Bitmap File](#)

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

### [Nine-Patch File](#)

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

### [Layer List](#)

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

### [State List](#)

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

### [Level List](#)

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

### [Transition Drawable](#)

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

### [Inset Drawable](#)

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

### [Clip Drawable](#)

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

### [Scale Drawable](#)

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

### [Shape Drawable](#)

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

Also see the [Animation Resource](#) document for how to create an [AnimationDrawable](#).

**Note:** A [color resource](#) can also be used as a drawable in XML. For example, when creating a [state list drawable](#), you can reference a color resource for the `android:drawable` attribute (`android:drawable="@color/green"`).

# Bitmap

A bitmap image. Android supports bitmap files in three formats: .png (preferred), .jpg (acceptable), .gif (discouraged).

You can reference a bitmap file directly, using the filename as the resource ID, or create an alias resource ID in XML.

**Note:** Bitmap files may be automatically optimized with lossless image compression by the aapt tool during the build process. For example, a true-color PNG that does not require more than 256 colors may be converted to an 8-bit PNG with a color palette. This will result in an image of equal quality but which requires less memory. So be aware that the image binaries placed in this directory can change during the build. If you plan on reading an image as a bit stream in order to convert it to a bitmap, put your images in the `res/raw/` folder instead, where they will not be optimized.

## Bitmap File

A bitmap file is a .png, .jpg, or .gif file. Android creates a [Drawable](#) resource for any of these files when you save them in the `res/drawable/` directory.

### file location:

`res/drawable/filename.png` (.png, .jpg, or .gif)

The filename is used as the resource ID.

### compiled resource datatype:

Resource pointer to a [BitmapDrawable](#).

### resource reference:

In Java: `R.drawable.filename`

In XML: `@[package:]drawable/filename`

### example:

With an image saved at `res/drawable/myimage.png`, this layout XML applies the image to a View:

```
<ImageView  
    android:layout_height="wrap_content"  
    android:layout_width="wrap_content"  
    android:src="@drawable/myimage" />
```

The following application code retrieves the image as a [Drawable](#):

```
Resources res = getResources\(\);  
Drawable drawable = res.getDrawable(R.drawable.myimage);
```

### see also:

- [2D Graphics](#)
- [BitmapDrawable](#)

## XML Bitmap

An XML bitmap is a resource defined in XML that points to a bitmap file. The effect is an alias for a raw bitmap file. The XML can specify additional properties for the bitmap such as dithering and tiling.

**Note:** You can use a `<bitmap>` element as a child of an `<item>` element. For example, when creating a [state list](#) or [layer list](#), you can exclude the `android:drawable` attribute from an `<item>` element and nest a `<bitmap>` inside it that defines the drawable item.

#### file location:

`res/drawable/filename.xml`

The filename is used as the resource ID.

#### compiled resource datatype:

Resource pointer to a [BitmapDrawable](#).

#### resource reference:

In Java: `R.drawable.filename`

In XML: `@[package:]drawable/filename`

#### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<bitmap
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:src="@[package:]drawable/drawable_resource"
    android:antialias=["true" | "false"]
    android:dither=["true" | "false"]
    android:filter=["true" | "false"]
    android:gravity=["top" | "bottom" | "left" | "right" | "center_vertical"
                    | "fill_vertical" | "center_horizontal" | "fill_horizontal"
                    | "center" | "fill" | "clip_vertical" | "clip_horizontal"]
    android:mipMap=["true" | "false"]
    android:tileMode=["disabled" | "clamp" | "repeat" | "mirror"] />
```

#### elements:

##### `<bitmap>`

Defines the bitmap source and its properties.

attributes:

##### `xmlns:android`

*String.* Defines the XML namespace, which must be `"http://schemas.android.com/apk/res/android"`. This is required only if the `<bitmap>` is the root element—it is not needed when the `<bitmap>` is nested inside an `<item>`.

##### `android:src`

*Drawable resource.* **Required.** Reference to a drawable resource.

##### `android:antialias`

*Boolean.* Enables or disables antialiasing.

##### `android:dither`

*Boolean.* Enables or disables dithering of the bitmap if the bitmap does not have the same pixel configuration as the screen (for instance: a ARGB 8888 bitmap with an RGB 565 screen).

##### `android:filter`

*Boolean.* Enables or disables bitmap filtering. Filtering is used when the bitmap is shrunk or stretched to smooth its appearance.

## **android:gravity**

*Keyword.* Defines the gravity for the bitmap. The gravity indicates where to position the drawable in its container if the bitmap is smaller than the container.

Must be one or more (separated by '|') of the following constant values:

<b>Value</b>	<b>Description</b>
top	Put the object at the top of its container, not changing its size.
bottom	Put the object at the bottom of its container, not changing its size.
left	Put the object at the left edge of its container, not changing its size.
right	Put the object at the right edge of its container, not changing its size.
center_vertical	Place object in the vertical center of its container, not changing its size.
fill_vertical	Grow the vertical size of the object if needed so it completely fills its container.
center_horizontal	Place object in the horizontal center of its container, not changing its size.
fill_horizontal	Grow the horizontal size of the object if needed so it completely fills its container.
center	Place the object in the center of its container in both the vertical and horizontal axis, not changing its size.
fill	Grow the horizontal and vertical size of the object if needed so it completely fills its container. This is the default.
clip_vertical	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 is based on the vertical gravity: a top gravity clips the bottom edge, a bottom gravity clips the top edge, and neither clips both edges.
clip_horizontal	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 is based on the horizontal gravity: a left gravity clips the right edge, a right gravity clips the left edge, and neither clips both edges.

## **android:mipMap**

*Boolean.* Enables or disables the mipmap hint. See [setHasMipMap\(\)](#) for more information. Default value is false.

## **android:tileMode**

*Keyword.* Defines the tile mode. When the tile mode is enabled, the bitmap is repeated. Gravity is ignored when the tile mode is enabled.

Must be one of the following constant values:

<b>Value</b>	<b>Description</b>
disabled	Do not tile the bitmap. This is the default value.
clamp	Replicates the edge color if the shader draws outside of its original bounds
repeat	Repeats the shader's image horizontally and vertically.
mirror	Repeats the shader's image horizontally and vertically, alternating mirror images so that adjacent images always seam.

## **example:**

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

```
    android:src="@drawable/icon"
    android:tileMode="repeat" />
```

**see also:**

- [BitmapDrawable](#)
- [Creating alias resources](#)

## Nine-Patch

A [NinePatch](#) is a PNG image in which you can define stretchable regions that Android scales when content within the View exceeds the normal image bounds. You typically assign this type of image as the background of a View that has at least one dimension set to "wrap\_content", and when the View grows to accommodate the content, the Nine-Patch image is also scaled to match the size of the View. An example use of a Nine-Patch image is the background used by Android's standard [Button](#) widget, which must stretch to accommodate the text (or image) inside the button.

Same as with a normal [bitmap](#), you can reference a Nine-Patch file directly or from a resource defined by XML.

For a complete discussion about how to create a Nine-Patch file with stretchable regions, see the [2D Graphics](#) document.

## Nine-Patch File

**file location:**

res/drawable/*filename*.9.png

The filename is used as the resource ID.

**compiled resource datatype:**

Resource pointer to a [NinePatchDrawable](#).

**resource reference:**

In Java: R.drawable.*filename*

In XML: @ [package:]drawable/*filename*

**example:**

With an image saved at res/drawable/myninepatch.9.png, this layout XML applies the Nine-Patch to a View:

```
<Button
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:background="@drawable/myninepatch" />
```

**see also:**

- [2D Graphics](#)
- [NinePatchDrawable](#)

## XML Nine-Patch

An XML Nine-Patch is a resource defined in XML that points to a Nine-Patch file. The XML can specify dithering for the image.

**file location:**

res/drawable/*filename*.xml

The filename is used as the resource ID.

**compiled resource datatype:**

Resource pointer to a [NinePatchDrawable](#).

**resource reference:**

In Java: R.drawable.*filename*

In XML: @ [package:]drawable/*filename*

**syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<nine-patch
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:src="@[package:]drawable/drawable_resource"
    android:dither=["true" | "false"] />
```

**elements:****<nine-patch>**

Defines the Nine-Patch source and its properties.

attributes:

**xmlns:android**

*String. Required.* Defines the XML namespace, which must be "http://schemas.android.com/apk/res/android".

**android:src**

*Drawable resource. Required.* Reference to a Nine-Patch file.

**android:dither**

*Boolean.* Enables or disables dithering of the bitmap if the bitmap does not have the same pixel configuration as the screen (for instance: a ARGB 8888 bitmap with an RGB 565 screen).

**example:**

```
<?xml version="1.0" encoding="utf-8"?>
<nine-patch xmlns:android="http://schemas.android.com/apk/res/android"
    android:src="@drawable/myninepatch"
    android:dither="false" />
```

## Layer List

A [LayerDrawable](#) is a drawable object that manages an array of other drawables. Each drawable in the list is drawn in the order of the list—the last drawable in the list is drawn on top.

Each drawable is represented by an `<item>` element inside a single `<layer-list>` element.

**file location:**

res/drawable/*filename*.xml

The filename is used as the resource ID.

## compiled resource datatype:

Resource pointer to a [LayerDrawable](#).

## resource reference:

In Java: R.drawable.*filename*

In XML: @ [package:]drawable/*filename*

## syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list
    xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:drawable="@ [package:]drawable/drawable_resource"
        android:id="@ [+][package:]id/resource_name"
        android:top="dimension"
        android:right="dimension"
        android:bottom="dimension"
        android:left="dimension" />
</layer-list>
```

## elements:

<layer-list>

**Required.** This must be the root element. Contains one or more <item> elements.

attributes:

**xmlns:android**

*String. Required.* Defines the XML namespace, which must be "http://schemas.android.com/apk/res/android".

<item>

Defines a drawable to place in the layer drawable, in a position defined by its attributes. Must be a child of a <selector> element. Accepts child <bitmap> elements.

attributes:

**android:drawable**

*Drawable resource. Required.* Reference to a drawable resource.

**android:id**

*Resource ID.* A unique resource ID for this drawable. To create a new resource ID for this item, use the form: "@+id/*name*". The plus symbol indicates that this should be created as a new ID. You can use this identifier to retrieve and modify the drawable with [View.findViewById\(\)](#) or [Activity.findViewById\(\)](#).

**android:top**

*Integer.* The top offset in pixels.

**android:right**

*Integer.* The right offset in pixels.

**android:bottom**

*Integer.* The bottom offset in pixels.

### **android:left**

*Integer.* The left offset in pixels.

All drawable items are scaled to fit the size of the containing View, by default. Thus, placing your images in a layer list at different positions might increase the size of the View and some images scale as appropriate. To avoid scaling items in the list, use a `<bitmap>` element inside the `<item>` element to specify the drawable and define the gravity to something that does not scale, such as "center". For example, the following `<item>` defines an item that scales to fit its container View:

```
<item android:drawable="@drawable/image" />
```

To avoid scaling, the following example uses a `<bitmap>` element with centered gravity:

```
<item>
    <bitmap android:src="@drawable/image"
            android:gravity="center" />
</item>
```

### **example:**

XML file saved at `res/drawable/layers.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item>
        <bitmap android:src="@drawable/android_red"
                android:gravity="center" />
    </item>
    <item android:top="10dp" android:left="10dp">
        <bitmap android:src="@drawable/android_green"
                android:gravity="center" />
    </item>
    <item android:top="20dp" android:left="20dp">
        <bitmap android:src="@drawable/android_blue"
                android:gravity="center" />
    </item>
</layer-list>
```

Notice that this example uses a nested `<bitmap>` element to define the drawable resource for each item with a "center" gravity. This ensures that none of the images are scaled to fit the size of the container, due to resizing caused by the offset images.

This layout XML applies the drawable to a View:

```
<ImageView
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:src="@drawable/layers" />
```

The result is a stack of increasingly offset images:



#### see also:

- [LayerDrawable](#)

## State List

A [StateListDrawable](#) is a drawable object defined in XML that uses several different images to represent the same graphic, depending on the state of the object. For example, a [Button](#) widget can exist in one of several different states (pressed, focused, or neither) and, using a state list drawable, you can provide a different background image for each state.

You can describe the state list in an XML file. Each graphic is represented by an `<item>` element inside a single `<selector>` element. Each `<item>` uses various attributes to describe the state in which it should be used as the graphic for the drawable.

During each state change, the state list is traversed top to bottom and the first item that matches the current state is used—the selection is *not* based on the "best match," but simply the first item that meets the minimum criteria of the state.

#### file location:

`res/drawable/filename.xml`

The filename is used as the resource ID.

#### compiled resource datatype:

Resource pointer to a [StateListDrawable](#).

#### resource reference:

In Java: `R.drawable.filename`

In XML: `@[package:]drawable/filename`

#### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android"
    android:constantSize=["true" | "false"]
    android:dither=["true" | "false"]
    android:variablePadding=["true" | "false"] >
    <item
        android:drawable="@[package:]drawable/drawable_resource"
        android:state_pressed=["true" | "false"]
        android:state_focused=["true" | "false"]
        android:state_hovered=["true" | "false"]
        android:state_selected=["true" | "false"]
        android:state_checkable=["true" | "false"]
        android:state_checked=["true" | "false"]
        android:state_enabled=["true" | "false"]
        android:state_ACTIVATED=["true" | "false"]
```

```
        android:state_window_focused=["true" | "false"] />
</selector>
```

## elements:

### <selector>

**Required.** This must be the root element. Contains one or more <item> elements.

attributes:

#### **xmlns:android**

*String. Required.* Defines the XML namespace, which must be "http://schemas.android.com/apk/res/android".

#### **android:constantSize**

*Boolean.* "true" if the drawable's reported internal size remains constant as the state changes (the size is the maximum of all of the states); "false" if the size varies based on the current state. Default is false.

#### **android:dither**

*Boolean.* "true" to enable dithering of the bitmap if the bitmap does not have the same pixel configuration as the screen (for instance, an ARGB 8888 bitmap with an RGB 565 screen); "false" to disable dithering. Default is true.

#### **android:variablePadding**

*Boolean.* "true" if the drawable's padding should change based on the current state that is selected; "false" if the padding should stay the same (based on the maximum padding of all the states). Enabling this feature requires that you deal with performing layout when the state changes, which is often not supported. Default is false.

### <item>

Defines a drawable to use during certain states, as described by its attributes. Must be a child of a <selector> element.

attributes:

#### **android:drawable**

*Drawable resource. Required.* Reference to a drawable resource.

#### **android:state\_pressed**

*Boolean.* "true" if this item should be used when the object is pressed (such as when a button is touched(clicked)); "false" if this item should be used in the default, non-pressed state.

#### **android:state\_focused**

*Boolean.* "true" if this item should be used when the object has input focus (such as when the user selects a text input); "false" if this item should be used in the default, non-focused state.

#### **android:state\_hovered**

*Boolean.* "true" if this item should be used when the object is being hovered by a cursor; "false" if this item should be used in the default, non-hovered state. Often, this drawable may be the same drawable used for the "focused" state.

Introduced in API level 14.

**`android:state_selected`**

*Boolean.* "true" if this item should be used when the object is the current user selection when navigating with a directional control (such as when navigating through a list with a d-pad); "false" if this item should be used when the object is not selected.

The selected state is used when focus (`android:state_focused`) is not sufficient (such as when list view has focus and an item within it is selected with a d-pad).

**`android:state_checkable`**

*Boolean.* "true" if this item should be used when the object is checkable; "false" if this item should be used when the object is not checkable. (Only useful if the object can transition between a checkable and non-checkable widget.)

**`android:state_checked`**

*Boolean.* "true" if this item should be used when the object is checked; "false" if it should be used when the object is un-checked.

**`android:state_enabled`**

*Boolean.* "true" if this item should be used when the object is enabled (capable of receiving touch/click events); "false" if it should be used when the object is disabled.

**`android:state_activated`**

*Boolean.* "true" if this item should be used when the object is activated as the persistent selection (such as to "highlight" the previously selected list item in a persistent navigation view); "false" if it should be used when the object is not activated.

Introduced in API level 11.

**`android:state_window.Focused`**

*Boolean.* "true" if this item should be used when the application window has focus (the application is in the foreground), "false" if this item should be used when the application window does not have focus (for example, if the notification shade is pulled down or a dialog appears).

**Note:** Remember that Android applies the first item in the state list that matches the current state of the object. So, if the first item in the list contains none of the state attributes above, then it is applied every time, which is why your default value should always be last (as demonstrated in the following example).

**example:**

XML file saved at `res/drawable/button.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
          android:drawable="@drawable/button_pressed" /> <!-- pressed -->
    <item android:state_focused="true"
          android:drawable="@drawable/button_focused" /> <!-- focused -->
    <item android:state_hovered="true"
          android:drawable="@drawable/button_focused" /> <!-- hovered -->
    <item android:drawable="@drawable/button_normal" /> <!-- default -->
</selector>
```

This layout XML applies the state list drawable to a Button:

```
<Button  
    android:layout_height="wrap_content"  
    android:layout_width="wrap_content"  
    android:background="@drawable/button" />
```

#### see also:

- [StateListDrawable](#)

## Level List

A Drawable that manages a number of alternate Drawables, each assigned a maximum numerical value. Setting the level value of the drawable with [setLevel\(\)](#) loads the drawable resource in the level list that has a `android:maxLevel` value greater than or equal to the value passed to the method.

#### file location:

`res/drawable/filename.xml`

The filename is used as the resource ID.

#### compiled resource datatype:

Resource pointer to a [LevelListDrawable](#).

#### resource reference:

In Java: `R.drawable.filename`

In XML: `@[package:]drawable/filename`

#### syntax:

```
<?xml version="1.0" encoding="utf-8"?>  
<level-list  
    xmlns:android="http://schemas.android.com/apk/res/android" >  
    <item  
        android:drawable="@drawable/drawable_resource"  
        android:maxLevel="integer"  
        android:minLevel="integer" />  
</level-list>
```

#### elements:

##### **<level-list>**

This must be the root element. Contains one or more `<item>` elements.

attributes:

##### **xmlns:android**

*String. Required.* Defines the XML namespace, which must be `"http://schemas.android.com/apk/res/android"`.

##### **<item>**

Defines a drawable to use at a certain level.

attributes:

##### **android:drawable**

*Drawable resource. Required.* Reference to a drawable resource to be inset.

**android:maxLevel**

*Integer*. The maximum level allowed for this item.

**android:minLevel**

*Integer*. The minimum level allowed for this item.

**example:**

```
<?xml version="1.0" encoding="utf-8"?>
<level-list xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:drawable="@drawable/status_off"
        android:maxLevel="0" />
    <item
        android:drawable="@drawable/status_on"
        android:maxLevel="1" />
</level-list>
```

Once this is applied to a [View](#), the level can be changed with [setLevel\(\)](#) or [setImageLevel\(\)](#).

**see also:**

- [LevelListDrawable](#)

## Transition Drawable

A [TransitionDrawable](#) is a drawable object that can cross-fade between the two drawable resources.

Each drawable is represented by an `<item>` element inside a single `<transition>` element. No more than two items are supported. To transition forward, call [startTransition\(\)](#). To transition backward, call [reverseTransition\(\)](#).

**file location:**

`res/drawable/filename.xml`

The filename is used as the resource ID.

**compiled resource datatype:**

Resource pointer to a [TransitionDrawable](#).

**resource reference:**

In Java: `R.drawable.filename`

In XML: `@[package:]drawable/filename`

**syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<transition
    xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:drawable="@[package:]drawable/drawable_resource"
        android:id="@[+][package:]id/resource_name"
        android:top="dimension"
        android:right="dimension"
        android:bottom="dimension"
```

```
        android:left="dimension" />
</transition>
```

## elements:

### <transition>

**Required.** This must be the root element. Contains one or more <item> elements.

attributes:

#### **xmlns:android**

*String. Required.* Defines the XML namespace, which must be "http://schemas.android.com/apk/res/android".

### <item>

Defines a drawable to use as part of the drawable transition. Must be a child of a <transition> element. Accepts child <bitmap> elements.

attributes:

#### **android:drawable**

*Drawable resource. Required.* Reference to a drawable resource.

#### **android:id**

*Resource ID.* A unique resource ID for this drawable. To create a new resource ID for this item, use the form: "@+id/*name*". The plus symbol indicates that this should be created as a new ID. You can use this identifier to retrieve and modify the drawable with [View.findViewById\(\)](#) or [Activity.findViewById\(\)](#).

#### **android:top**

*Integer.* The top offset in pixels.

#### **android:right**

*Integer.* The right offset in pixels.

#### **android:bottom**

*Integer.* The bottom offset in pixels.

#### **android:left**

*Integer.* The left offset in pixels.

## example:

XML file saved at res/drawable/transition.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<transition xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/on" />
    <item android:drawable="@drawable/off" />
</transition>
```

This layout XML applies the drawable to a View:

```
<ImageButton
    android:id="@+id/button"
    android:layout_height="wrap_content"
```

```
    android:layout_width="wrap_content"
    android:src="@drawable/transition" />
```

And the following code performs a 500ms transition from the first item to the second:

```
ImageButton button = (ImageButton) findViewById(R.id.button);
TransitionDrawable drawable = (TransitionDrawable) button.getDrawable();
drawable.startTransition(500);
```

**see also:**

- [TransitionDrawable](#)

## Inset Drawable

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

**file location:**

res/drawable/*filename*.xml

The filename is used as the resource ID.

**compiled resource datatype:**

Resource pointer to a [InsetDrawable](#).

**resource reference:**

In Java: R.drawable.*filename*

In XML: @ [package:]drawable/*filename*

**syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<inset
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/drawable_resource"
    android:insetTop="dimension"
    android:insetRight="dimension"
    android:insetBottom="dimension"
    android:insetLeft="dimension" />
```

**elements:**

**<inset>**

Defines the inset drawable. This must be the root element.

attributes:

**xmlns:android**

*String. Required.* Defines the XML namespace, which must be "http://schemas.android.com/apk/res/android".

**android:drawable**

*Drawable resource. Required.* Reference to a drawable resource to be inset.

**android:insetTop**

*Dimension.* The top inset, as a dimension value or [dimension resource](#)

**android:insetRight**

*Dimension.* The right inset, as a dimension value or [dimension resource](#)

**android:insetBottom**

*Dimension.* The bottom inset, as a dimension value or [dimension resource](#)

**android:insetLeft**

*Dimension.* The left inset, as a dimension value or [dimension resource](#)

**example:**

```
<?xml version="1.0" encoding="utf-8"?>
<inset xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/background"
    android:insetTop="10dp"
    android:insetLeft="10dp" />
```

**see also:**

- [InsetDrawable](#)

## Clip Drawable

A drawable defined in XML that clips another drawable based on this Drawable's current level. You can control how much the child drawable gets clipped in width and height based on the level, as well as a gravity to control where it is placed in its overall container. Most often used to implement things like progress bars.

**file location:**

res/drawable/*filename*.xml

The filename is used as the resource ID.

**compiled resource datatype:**

Resource pointer to a [ClipDrawable](#).

**resource reference:**

In Java: R.drawable.*filename*

In XML: @ [package:]drawable/*filename*

**syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<clip
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/drawable_resource"
    android:clipOrientation=["horizontal" | "vertical"]
    android:gravity=["top" | "bottom" | "left" | "right" | "center_vertical"
                    | "fill_vertical" | "center_horizontal" | "fill_horizontal"
                    | "center" | "fill" | "clip_vertical" | "clip_horizontal"]>
```

**elements:**
**<clip>**

Defines the clip drawable. This must be the root element.

attributes:

**`xmlns:android`**

*String. Required.* Defines the XML namespace, which must be "<http://schemas.android.com/apk/res/android>".

**`android:drawable`**

*Drawable resource. Required.* Reference to a drawable resource to be clipped.

**`android:clipOrientation`**

*Keyword.* The orientation for the clip.

Must be one of the following constant values:

<b>Value</b>	<b>Description</b>
horizontal	Clip the drawable horizontally.
vertical	Clip the drawable vertically.

**`android:gravity`**

*Keyword.* Specifies where to clip within the drawable.

Must be one or more (separated by '|') of the following constant values:

<b>Value</b>	<b>Description</b>
top	Put the object at the top of its container, not changing its size. When <code>clipOrientation</code> is "vertical", clipping occurs at the bottom of the drawable.
bottom	Put the object at the bottom of its container, not changing its size. When <code>clipOrientation</code> is "vertical", clipping occurs at the top of the drawable.
left	Put the object at the left edge of its container, not changing its size. This is the default. When <code>clipOrientation</code> is "horizontal", clipping occurs at the right side of the drawable. This is the default.
right	Put the object at the right edge of its container, not changing its size. When <code>clipOrientation</code> is "horizontal", clipping occurs at the left side of the drawable.
center_vertical	Place object in the vertical center of its container, not changing its size. Clipping behaves the same as when <code>gravity</code> is "center".
fill_vertical	Grow the vertical size of the object if needed so it completely fills its container. When <code>clipOrientation</code> is "vertical", no clipping occurs because the drawable fills the vertical space (unless the drawable level is 0, in which case it's not visible).
center_horizontal	Place object in the horizontal center of its container, not changing its size. Clipping behaves the same as when <code>gravity</code> is "center".
fill_horizontal	Grow the horizontal size of the object if needed so it completely fills its container. When <code>clipOrientation</code> is "horizontal", no clipping occurs because the drawable fills the horizontal space (unless the drawable level is 0, in which case it's not visible).
center	Place the object in the center of its container in both the vertical and horizontal axis, not changing its size. When <code>clipOrientation</code> is "horizontal", clipping occurs on the left and right. When <code>clipOrientation</code> is "vertical", clipping occurs on the top and bottom.

fill

clip\_vertical

clip\_horizontal

Grow the horizontal and vertical size of the object if needed so it completely fills its container. No clipping occurs because the drawable fills the horizontal and vertical space (unless the drawable level is 0, in which case it's not visible).

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 is based on the vertical gravity: a top gravity clips the bottom edge, a bottom gravity clips the top edge, and neither clips both edges.

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 is based on the horizontal gravity: a left gravity clips the right edge, a right gravity clips the left edge, and neither clips both edges.

### example:

XML file saved at res/drawable/clip.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<clip xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/android"
    android:clipOrientation="horizontal"
    android:gravity="left" />
```

The following layout XML applies the clip drawable to a View:

```
<ImageView
    android:id="@+id/image"
    android:background="@drawable/clip"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content" />
```

The following code gets the drawable and increases the amount of clipping in order to progressively reveal the image:

```
ImageView imageview = (ImageView) findViewById(R.id.image);
ClipDrawable drawable = (ClipDrawable) imageview.getDrawable();
drawable.setLevel(drawable.getLevel() + 1000);
```

Increasing the level reduces the amount of clipping and slowly reveals the image. Here it is at a level of 7000:



**Note:** The default level is 0, which is fully clipped so the image is not visible. When the level is 10,000, the image is not clipped and completely visible.

### see also:

- [ClipDrawable](#)

# Scale Drawable

A drawable defined in XML that changes the size of another drawable based on its current level.

## file location:

res/drawable/*filename*.xml

The filename is used as the resource ID.

## compiled resource datatype:

Resource pointer to a [ScaleDrawable](#).

## resource reference:

In Java: R.drawable.*filename*

In XML: @ [package:]drawable/*filename*

## syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<scale
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/drawable_resource"
    android:scaleGravity=["top" | "bottom" | "left" | "right" | "center_vertical"
                        | "fill_vertical" | "center_horizontal" | "fill_horizontal"
                        | "center" | "fill" | "clip_vertical" | "clip_horizontal"]
    android:scaleHeight="percentage"
    android:scaleWidth="percentage" />
```

## elements:

### <scale>

Defines the scale drawable. This must be the root element.

attributes:

#### xmlns:android

*String. Required.* Defines the XML namespace, which must be "http://schemas.android.com/apk/res/android".

#### android:drawable

*Drawable resource. Required.* Reference to a drawable resource.

#### android:scaleGravity

*Keyword.* Specifies the gravity position after scaling.

Must be one or more (separated by '|') of the following constant values:

Value	Description
top	Put the object at the top of its container, not changing its size.
bottom	Put the object at the bottom of its container, not changing its size.
left	Put the object at the left edge of its container, not changing its size. This is the default.
right	Put the object at the right edge of its container, not changing its size.
center_vertical	Place object in the vertical center of its container, not changing its size.

<code>fill_vertical</code>	Grow the vertical size of the object if needed so it completely fills its container.
<code>center_horizontal</code>	Place object in the horizontal center of its container, not changing its size.
<code>fill_horizontal</code>	Grow the horizontal size of the object if needed so it completely fills its container.
<code>center</code>	Place the object in the center of its container in both the vertical and horizontal axis, not changing its size.
<code>fill</code>	Grow the horizontal and vertical size of the object if needed so it completely fills its container.
<code>clip_vertical</code>	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 is based on the vertical gravity: a top gravity clips the bottom edge, a bottom gravity clips the top edge, and neither clips both edges.
<code>clip_horizontal</code>	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 is based on the horizontal gravity: a left gravity clips the right edge, a right gravity clips the left edge, and neither clips both edges.

#### **android:scaleHeight**

*Percentage.* The scale height, expressed as a percentage of the drawable's bound. The value's format is XX%. For instance: 100%, 12.5%, etc.

#### **android:scaleWidth**

*Percentage.* The scale width, expressed as a percentage of the drawable's bound. The value's format is XX%. For instance: 100%, 12.5%, etc.

#### **example:**

```
<?xml version="1.0" encoding="utf-8"?>
<scale xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/logo"
    android:scaleGravity="center_vertical|center_horizontal"
    android:scaleHeight="80%"
    android:scaleWidth="80%" />
```

#### **see also:**

- [ScaleDrawable](#)

## **Shape Drawable**

This is a generic shape defined in XML.

#### **file location:**

`res/drawable/filename.xml`

The filename is used as the resource ID.

#### **compiled resource datatype:**

Resource pointer to a [GradientDrawable](#).

#### **resource reference:**

In Java: `R.drawable.filename`

In XML: `@[package:]drawable/filename`

## syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape=["rectangle" | "oval" | "line" | "ring"] >
    <corners
        android:radius="integer"
        android:topLeftRadius="integer"
        android:topRightRadius="integer"
        android:bottomLeftRadius="integer"
        android:bottomRightRadius="integer" />
    <gradient
        android:angle="integer"
        android:centerX="integer"
        android:centerY="integer"
        android:centerColor="integer"
        android:endColor="color"
        android:gradientRadius="integer"
        android:startColor="color"
        android:type=["linear" | "radial" | "sweep"]
        android:useLevel=["true" | "false"] />
    <padding
        android:left="integer"
        android:top="integer"
        android:right="integer"
        android:bottom="integer" />
    <size
        android:width="integer"
        android:height="integer" />
    <solid
        android:color="color" />
    <stroke
        android:width="integer"
        android:color="color"
        android:dashWidth="integer"
        android:dashGap="integer" />
</shape>
```

## elements:

### <shape>

The shape drawable. This must be the root element.

attributes:

#### **xmlns:android**

*String. Required.* Defines the XML namespace, which must be "http://schemas.android.com/apk/res/android".

#### **android:shape**

*Keyword.* Defines the type of shape. Valid values are:

<b>Value</b>	<b>Description</b>
"rectangle"	A rectangle that fills the containing View. This is the default shape.
"oval"	An oval shape that fits the dimensions of the containing View.

"line"	A horizontal line that spans the width of the containing View. This shape requires the <stroke> element to define the width of the line.
"ring"	A ring shape.

The following attributes are used only when `android:shape="ring"`:

#### **android:innerRadius**

*Dimension*. The radius for the inner part of the ring (the hole in the middle), as a dimension value or [dimension resource](#).

#### **android:innerRadiusRatio**

*Float*. The radius for the inner part of the ring, expressed as a ratio of the ring's width. For instance, if `android:innerRadiusRatio="5"`, then the inner radius equals the ring's width divided by 5. This value is overridden by `android:innerRadius`. Default value is 9.

#### **android:thickness**

*Dimension*. The thickness of the ring, as a dimension value or [dimension resource](#).

#### **android:thicknessRatio**

*Float*. The thickness of the ring, expressed as a ratio of the ring's width. For instance, if `android:thicknessRatio="2"`, then the thickness equals the ring's width divided by 2. This value is overridden by `android:innerRadius`. Default value is 3.

#### **android:useLevel**

*Boolean*. "true" if this is used as a [LevelListDrawable](#). This should normally be "false" or your shape may not appear.

### **<corners>**

Creates rounded corners for the shape. Applies only when the shape is a rectangle.

attributes:

#### **android:radius**

*Dimension*. The radius for all corners, as a dimension value or [dimension resource](#). This is overridden for each corner by the following attributes.

#### **android:topLeftRadius**

*Dimension*. The radius for the top-left corner, as a dimension value or [dimension resource](#).

#### **android:topRightRadius**

*Dimension*. The radius for the top-right corner, as a dimension value or [dimension resource](#).

#### **android:bottomLeftRadius**

*Dimension*. The radius for the bottom-left corner, as a dimension value or [dimension resource](#).

#### **android:bottomRightRadius**

*Dimension*. The radius for the bottom-right corner, as a dimension value or [dimension resource](#).

**Note:** Every corner must (initially) be provided a corner radius greater than 1, or else no corners are rounded. If you want specific corners to *not* be rounded, a work-around is to use `android:radius` to set a default corner radius greater than 1, but then override each and every corner with the values you really want, providing zero ("0dp") where you don't want rounded corners.

## <gradient>

Specifies a gradient color for the shape.

attributes:

### **android:angle**

*Integer.* The angle for the gradient, in degrees. 0 is left to right, 90 is bottom to top. It must be a multiple of 45. Default is 0.

### **android:centerX**

*Float.* The relative X-position for the center of the gradient (0 - 1.0).

### **android:centerY**

*Float.* The relative Y-position for the center of the gradient (0 - 1.0).

### **android:centerColor**

*Color.* Optional color that comes between the start and end colors, as a hexadecimal value or [color resource](#).

### **android:endColor**

*Color.* The ending color, as a hexadecimal value or [color resource](#).

### **android:gradientRadius**

*Float.* The radius for the gradient. Only applied when android:type="radial".

### **android:startColor**

*Color.* The starting color, as a hexadecimal value or [color resource](#).

### **android:type**

*Keyword.* The type of gradient pattern to apply. Valid values are:

<b>Value</b>	<b>Description</b>
"linear"	A linear gradient. This is the default.
"radial"	A radial gradient. The start color is the center color.
"sweep"	A sweeping line gradient.

### **android:useLevel**

*Boolean.* "true" if this is used as a [LevelListDrawable](#).

## <padding>

Padding to apply to the containing View element (this pads the position of the View content, not the shape).

attributes:

### **android:left**

*Dimension.* Left padding, as a dimension value or [dimension resource](#).

### **android:top**

*Dimension.* Top padding, as a dimension value or [dimension resource](#).

### **android:right**

*Dimension.* Right padding, as a dimension value or [dimension resource](#).

### **android:bottom**

*Dimension.* Bottom padding, as a dimension value or [dimension resource](#).

## <size>

The size of the shape.

attributes:

### **android:height**

*Dimension*. The height of the shape, as a dimension value or [dimension resource](#).

### **android:width**

*Dimension*. The width of the shape, as a dimension value or [dimension resource](#).

**Note:** The shape scales to the size of the container View proportionate to the dimensions defined here, by default. When you use the shape in an [ImageView](#), you can restrict scaling by setting the [android:scaleType](#) to "center".

## <solid>

A solid color to fill the shape.

attributes:

### **android:color**

*Color*. The color to apply to the shape, as a hexadecimal value or [color resource](#).

## <stroke>

A stroke line for the shape.

attributes:

### **android:width**

*Dimension*. The thickness of the line, as a dimension value or [dimension resource](#).

### **android:color**

*Color*. The color of the line, as a hexadecimal value or [color resource](#).

### **android:dashGap**

*Dimension*. The distance between line dashes, as a dimension value or [dimension resource](#). Only valid if `android:dashWidth` is set.

### **android:dashWidth**

*Dimension*. The size of each dash line, as a dimension value or [dimension resource](#). Only valid if `android:dashGap` is set.

## example:

XML file saved at `res/drawable/gradient_box.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <gradient
        android:startColor="#FFFF0000"
        android:endColor="#80FF00FF"
        android:angle="45"/>
    <padding android:left="7dp"
        android:top="7dp"
        android:right="7dp"
```

```
        android:bottom="7dp" />
    <corners android:radius="8dp" />
</shape>
```

This layout XML applies the shape drawable to a View:

```
<TextView
    android:background="@drawable/gradient_box"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content" />
```

This application code gets the shape drawable and applies it to a View:

```
Resources res = getResources\(\);
Drawable shape = res. getDrawable(R.drawable.gradient_box);

TextView tv = (TextView) findViewById(R.id.textview);
tv.setBackground(shape);
```

**see also:**

- [ShapeDrawable](#)

# Layout Resource

## See also

1. [Layouts](#)

A layout resource defines the architecture for the UI in an Activity or a component of a UI.

### file location:

`res/layout/filename.xml`

The filename will be used as the resource ID.

### compiled resource datatype:

Resource pointer to a [View](#) (or subclass) resource.

### resource reference:

In Java: `R.layout.filename`

In XML: `@[package:]layout/filename`

### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<ViewGroup xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@[+] [package:]id/resource_name"
    android:layout_height=["dimension" | "fill_parent" | "wrap_content"]
    android:layout_width=["dimension" | "fill_parent" | "wrap_content"]
    [ViewGroup-specific attributes] >
    <View
        android:id="@[+] [package:]id/resource_name"
        android:layout_height=["dimension" | "fill_parent" | "wrap_content"]
        android:layout_width=["dimension" | "fill_parent" | "wrap_content"]
        [View-specific attributes] >
        <requestFocus/>
    </View>
    <ViewGroup >
        <View />
    </ViewGroup>
    <include layout="@layout/layout_resource"/>
</ViewGroup>
```

**Note:** The root element can be either a [ViewGroup](#), a [View](#), or a [<merge>](#) element, but there must be only one root element and it must contain the `xmlns:android` attribute with the android namespace as shown.

### elements:

#### `<ViewGroup>`

A container for other [View](#) elements. There are many different kinds of [ViewGroup](#) objects and each one lets you specify the layout of the child elements in different ways. Different kinds of [ViewGroup](#) objects include [LinearLayout](#), [RelativeLayout](#), and [FrameLayout](#).

You should not assume that any derivation of [ViewGroup](#) will accept nested [Views](#). Some [ViewGroups](#) are implementations of the [AdapterView](#) class, which determines its children only from an [Adapter](#).

attributes:

#### **android:id**

*Resource ID.* A unique resource name for the element, which you can use to obtain a reference to the [ViewGroup](#) from your application. See more about the [value for android:id](#) below.

#### **android:layout\_height**

*Dimension or keyword. Required.* The height for the group, as a dimension value (or [dimension resource](#)) or a keyword ("fill\_parent" or "wrap\_content"). See the [valid values](#) below.

#### **android:layout\_width**

*Dimension or keyword. Required.* The width for the group, as a dimension value (or [dimension resource](#)) or a keyword ("fill\_parent" or "wrap\_content"). See the [valid values](#) below.

More attributes are supported by the [ViewGroup](#) base class, and many more are supported by each implementation of [ViewGroup](#). For a reference of all available attributes, see the corresponding reference documentation for the [ViewGroup](#) class (for example, the [LinearLayout XML attributes](#)).

### **<View>**

An individual UI component, generally referred to as a "widget". Different kinds of [View](#) objects include [TextView](#), [Button](#), and [CheckBox](#).

attributes:

#### **android:id**

*Resource ID.* A unique resource name for the element, which you can use to obtain a reference to the [View](#) from your application. See more about the [value for android:id](#) below.

#### **android:layout\_height**

*Dimension or keyword. Required.* The height for the element, as a dimension value (or [dimension resource](#)) or a keyword ("fill\_parent" or "wrap\_content"). See the [valid values](#) below.

#### **android:layout\_width**

*Dimension or keyword. Required.* The width for the element, as a dimension value (or [dimension resource](#)) or a keyword ("fill\_parent" or "wrap\_content"). See the [valid values](#) below.

More attributes are supported by the [View](#) base class, and many more are supported by each implementation of [View](#). Read [Layouts](#) for more information. For a reference of all available attributes, see the corresponding reference documentation (for example, the [TextView XML attributes](#)).

### **<requestFocus>**

Any element representing a [View](#) object can include this empty element, which gives its parent initial focus on the screen. You can have only one of these elements per file.

### **<include>**

Includes a layout file into this layout.

attributes:

#### **layout**

*Layout resource. Required.* Reference to a layout resource.

**android:id**

*Resource ID.* Overrides the ID given to the root view in the included layout.

**android:layout\_height**

*Dimension or keyword.* Overrides the height given to the root view in the included layout. Only effective if android:layout\_width is also declared.

**android:layout\_width**

*Dimension or keyword.* Overrides the width given to the root view in the included layout. Only effective if android:layout\_height is also declared.

You can include any other layout attributes in the <include> that are supported by the root element in the included layout and they will override those defined in the root element.

**Caution:** If you want to override layout attributes using the <include> tag, you must override both android:layout\_height and android:layout\_width in order for other layout attributes to take effect.

Another way to include a layout is to use [ViewStub](#). It is a lightweight View that consumes no layout space until you explicitly inflate it, at which point, it includes a layout file defined by its android:layout attribute. For more information about using [ViewStub](#), read [Loading Views On Demand](#).

**<merge>**

An alternative root element that is not drawn in the layout hierarchy. Using this as the root element is useful when you know that this layout will be placed into a layout that already contains the appropriate parent View to contain the children of the <merge> element. This is particularly useful when you plan to include this layout in another layout file using <include> and this layout doesn't require a different [viewGroup](#) container. For more information about merging layouts, read [Re-using Layouts with <include/>](#).

**Value for android:id**

For the ID value, you should usually use this syntax form: "@+id/*name*". The plus symbol, +, indicates that this is a new resource ID and the aapt tool will create a new resource integer in the R.java class, if it doesn't already exist. For example:

```
<TextView android:id="@+id/nameTextbox"/>
```

The nameTextbox name is now a resource ID attached to this element. You can then refer to the [TextView](#) to which the ID is associated in Java:

```
findViewById(R.id.nameTextbox);
```

This code returns the [TextView](#) object.

However, if you have already defined an [ID resource](#) (and it is not already used), then you can apply that ID to a [View](#) element by excluding the plus symbol in the android:id value.

**Value for android:layout\_height and android:layout\_width:**

The height and width value can be expressed using any of the [dimension units](#) supported by Android (px, dp, sp, pt, in, mm) or with the following keywords:

Value	Description
<code>match_parent</code>	Sets the dimension to match that of the parent element. Added in API Level 8 to deprecate <code>fill_parent</code> .
<code>fill_parent</code>	Sets the dimension to match that of the parent element.
<code>wrap_content</code>	Sets the dimension only to the size required to fit the content of this element.

## Custom View elements

You can create your own custom [View](#) and [ViewGroup](#) elements and apply them to your layout the same as a standard layout element. You can also specify the attributes supported in the XML element. To learn more, see the [Custom Components](#) developer guide.

### example:

XML file saved at `res/layout/main_activity.xml`:

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

This application code will load the layout for an [Activity](#), in the [onCreate\(\)](#) method:

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

### see also:

- [Layouts](#)
- [View](#)
- [ViewGroup](#)

# Menu Resource

## See also

1. [Menus](#)

A menu resource defines an application menu (Options Menu, Context Menu, or submenu) that can be inflated with [MenuInflater](#).

For a guide to using menus, see the [Menus](#) developer guide.

### file location:

`res/menu/filename.xml`

The filename will be used as the resource ID.

### compiled resource datatype:

Resource pointer to a [Menu](#) (or subclass) resource.

### resource reference:

In Java: `R.menu.filename`

In XML: `@[package:]menu.filename`

### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@[+] [package:]id/resource_name"
          android:title="string"
          android:titleCondensed="string"
          android:icon="@[package:]drawable/drawable_resource_name"
          android:onClick="method name"
          android:showAsAction=["ifRoom" | "never" | "withText" | "always" | "neverNest"]
          android:actionLayout="@[package:]layout/layout_resource_name"
          android:actionViewClass="class name"
          android:actionProviderClass="class name"
          android:alphabeticShortcut="string"
          android:numericShortcut="string"
          android:checkable=["true" | "false"]
          android:visible=["true" | "false"]
          android:enabled=["true" | "false"]
          android:menuCategory=["container" | "system" | "secondary" | "alter"]
          android:orderInCategory="integer" />
    <group android:id="@[+] [package:]id/resource_name"
           android:checkableBehavior=["none" | "all" | "single"]
           android:visible=["true" | "false"]
           android:enabled=["true" | "false"]
           android:menuCategory=["container" | "system" | "secondary" | "alter"]
           android:orderInCategory="integer" >
        <item />
    </group>
    <item >
        <menu>
            <item />
        </menu>
    </item >
</menu>
```

```
</menu>
</item>
</menu>
```

## elements:

### <menu>

**Required.** This must be the root node. Contains <item> and/or <group> elements.

attributes:

#### **xmlns:android**

*XML namespace.* **Required.** Defines the XML namespace, which must be "http://schemas.android.com/apk/res/android".

### <item>

A menu item. May contain a <menu> element (for a Sub Menu). Must be a child of a <menu> or <group> element.

attributes:

#### **android:id**

*Resource ID.* A unique resource ID. To create a new resource ID for this item, use the form: "@+id/*name*". The plus symbol indicates that this should be created as a new ID.

#### **android:title**

*String resource.* The menu title as a string resource or raw string.

#### **android:titleCondensed**

*String resource.* A condensed title as a string resource or a raw string. This title is used for situations in which the normal title is too long.

#### **android:icon**

*Drawable resource.* An image to be used as the menu item icon.

#### **android:onClick**

*Method name.* The method to call when this menu item is clicked. The method must be declared in the activity as public and accept a [MenuItem](#) as its only parameter, which indicates the item clicked. This method takes precedence over the standard callback to [onOptionsItemSelected\(\)](#). See the example at the bottom.

**Warning:** If you obfuscate your code using [ProGuard](#) (or a similar tool), be sure to exclude the method you specify in this attribute from renaming, because it can break the functionality.

Introduced in API Level 11.

#### **android:showAsAction**

*Keyword.* When and how this item should appear as an action item in the Action Bar. A menu item can appear as an action item only when the activity includes an [ActionBar](#) (introduced in API Level 11). Valid values:

Value	Description
ifRoom	Only place this item in the Action Bar if there is room for it.
withText	Also include the title text (defined by <code>android:title</code> ) with the action item. You can include this value along with one of the others as a flag set, by separating them with a pipe   .

never	Never place this item in the Action Bar.
always	Always place this item in the Action Bar. Avoid using this unless it's critical that the item always appear in the action bar. Setting multiple items to always appear as action items can result in them overlapping with other UI in the action bar.
<code>collapseActionBar</code>	The action view associated with this action item (as declared by <code>android:actionLayout</code> or <code>android:actionViewClass</code> ) is collapsible. Introduced in API Level 14.

See the [Action Bar](#) developer guide for more information.

Introduced in API Level 11.

#### **android:actionLayout**

*Layout resource.* A layout to use as the action view.

See the [Action Bar](#) developer guide for more information.

Introduced in API Level 11.

#### **android:actionViewClass**

*Class name.* A fully-qualified class name for the [View](#) to use as the action view. For example, "android.widget.SearchView" to use [SearchView](#) as an action view.

See the [Action Bar](#) developer guide for more information.

**Warning:** If you obfuscate your code using [ProGuard](#) (or a similar tool), be sure to exclude the class you specify in this attribute from renaming, because it can break the functionality.

Introduced in API Level 11.

#### **android:actionProviderClass**

*Class name.* A fully-qualified class name for the [ActionProvider](#) to use in place of the action item. For example, "android.widget.ShareActionProvider" to use [ShareActionProvider](#).

See the [Action Bar](#) developer guide for more information.

**Warning:** If you obfuscate your code using [ProGuard](#) (or a similar tool), be sure to exclude the class you specify in this attribute from renaming, because it can break the functionality.

Introduced in API Level 14.

#### **android:alphabeticShortcut**

*Char.* A character for the alphabetic shortcut key.

#### **android:numericShortcut**

*Integer.* A number for the numeric shortcut key.

#### **android:checkable**

*Boolean.* "true" if the item is checkable.

**android:checked**

*Boolean.* "true" if the item is checked by default.

**android:visible**

*Boolean.* "true" if the item is visible by default.

**android:enabled**

*Boolean.* "true" if the item is enabled by default.

**android:menuCategory**

*Keyword.* Value corresponding to [Menu CATEGORY\\_\\*](#) constants, which define the item's priority. Valid values:

Value	Description
container	For items that are part of a container.
system	For items that are provided by the system.
secondary	For items that are user-supplied secondary (infrequently used) options.
alternative	For items that are alternative actions on the data that is currently displayed.

**android:orderInCategory**

*Integer.* The order of "importance" of the item, within a group.

**<group>**

A menu group (to create a collection of items that share traits, such as whether they are visible, enabled, or checkable). Contains one or more <item> elements. Must be a child of a <menu> element.

attributes:

**android:id**

*Resource ID.* A unique resource ID. To create a new resource ID for this item, use the form: "@+id/*name*". The plus symbol indicates that this should be created as a new ID.

**android:checkableBehavior**

*Keyword.* The type of checkable behavior for the group. Valid values:

Value	Description
none	Not checkable
all	All items can be checked (use checkboxes)
single	Only one item can be checked (use radio buttons)

**android:visible**

*Boolean.* "true" if the group is visible.

**android:enabled**

*Boolean.* "true" if the group is enabled.

**android:menuCategory**

*Keyword.* Value corresponding to [Menu CATEGORY\\_\\*](#) constants, which define the group's priority. Valid values:

Value	Description
container	For groups that are part of a container.
system	For groups that are provided by the system.
secondary	For groups that are user-supplied secondary (infrequently used) options.
alternative	For groups that are alternative actions on the data that is currently displayed.

**android:orderInCategory**

*Integer.* The default order of the items within the category.

**example:**

XML file saved at res/menu/example\_menu.xml:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/item1"
          android:title="@string/item1"
          android:icon="@drawable/group_item1_icon"
          android:showAsAction="ifRoom|withText"/>
    <group android:id="@+id/group">
        <item android:id="@+id/group_item1"
              android:onClick="onGroupItemClick"
              android:title="@string/group_item1"
              android:icon="@drawable/group_item1_icon" />
        <item android:id="@+id/group_item2"
              android:onClick="onGroupItemClick"
              android:title="@string/group_item2"
              android:icon="@drawable/group_item2_icon" />
    </group>
    <item android:id="@+id/submenu"
          android:title="@string/submenu_title"
          android:showAsAction="ifRoom|withText" >
        <menu>
            <item android:id="@+id_submenu_item1"
                  android:title="@string/submenu_item1" />
        </menu>
    </item>
</menu>
```

The following application code inflates the menu from the [onCreateOptionsMenu \(Menu\)](#) callback and also declares the on-click callback for two of the items:

```
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.example_menu, menu);
    return true;
}

public void onGroupItemClick(MenuItem item) {
    // One of the group items (using the onClick attribute) was clicked
    // The item parameter passed here indicates which item it is
    // All other menu item clicks are handled by onOptionsItemSelected\(\)
}
```

**Note:** The android:showAsAction attribute is available only on Android 3.0 (API Level 11) and greater.

# String Resources

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

## String

XML resource that provides a single string.

## String Array

XML resource that provides an array of strings.

## Quantity Strings (Plurals)

XML resource that carries different strings for pluralization.

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

## String

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

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

### file location:

res/values/*filename*.xml

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

### compiled resource datatype:

Resource pointer to a [String](#).

### resource reference:

In Java: R.string.*string\_name*

In XML:@string/*string\_name*

### syntax:

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

### elements:

<resources>

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

No attributes.

## **<string>**

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

attributes:

### **name**

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

### **example:**

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

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

This layout XML applies a string to a View:

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello" />
```

This application code retrieves a string:

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

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

# String Array

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

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

### **file location:**

`res/values/filename.xml`

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

### **compiled resource datatype:**

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

### **resource reference:**

In Java: `R.array.string_array_name`

### **syntax:**

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

```
<string-array
    name="string_array_name">
    <item>
        <text_string></item>
    </string-array>
</resources>
```

#### elements:

<resources>

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

No attributes.

<string-array>

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

attributes:

**name**

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

<item>

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

No attributes.

#### example:

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

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

This application code retrieves a string array:

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

## Quantity Strings (Plurals)

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

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

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

The selection of which string to use is made solely based on grammatical *necessity*. In English, a string for `zero` will be ignored even if the quantity is 0, because 0 isn't grammatically different from 2, or any other number except 1 ("zero books", "one book", "two books", and so on).

Don't be misled either by the fact that, say, `two` sounds like it could only apply to the quantity 2: a language may require that 2, 12, 102 (and so on) are all treated like one another but differently to other quantities. Rely on your translator to know what distinctions their language actually insists upon.

It's often possible to avoid quantity strings by using quantity-neutral formulations such as "Books: 1". This will make your life and your translators' lives easier, if it's a style that's in keeping with your application.

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

#### file location:

`res/values/filename.xml`

The filename is arbitrary. The `<plurals>` element's name will be used as the resource ID.

#### resource reference:

In Java: `R.plurals_plural_name`

#### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <plurals
        name="plural_name">
        <item
            quantity=["zero" | "one" | "two" | "few" | "many" | "other"]>text_string</item>
    </plurals>
</resources>
```

#### elements:

`<resources>`

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

No attributes.

`<plurals>`

A collection of strings, of which, one string is provided depending on the amount of something. Contains one or more `<item>` elements.

attributes:

**name**

*String.* A name for the pair of strings. This name will be used as the resource ID.

**<item>**

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

attributes:

**quantity**

*Keyword.* A value indicating when this string should be used. Valid values, with non-exhaustive examples in parentheses:

Value	Description
zero	When the language requires special treatment of the number 0 (as in Arabic).
one	When the language requires special treatment of numbers like one (as with the number 1 in English and most other languages; in Russian, any number ending in 1 but not ending in 11 is in this class).
two	When the language requires special treatment of numbers like two (as with 2 in Welsh, or 102 in Slovenian).
few	When the language requires special treatment of "small" numbers (as with 2, 3, and 4 in Czech; or numbers ending 2, 3, or 4 but not 12, 13, or 14 in Polish).
many	When the language requires special treatment of "large" numbers (as with numbers ending 11-99 in Maltese).
other	When the language does not require special treatment of the given quantity (as with all numbers in Chinese, or 42 in English).

**example:**

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <plurals name="numberOfSongsAvailable">
        <item quantity="one">One song found.</item>
        <item quantity="other">%d songs found.</item>
    </plurals>
</resources>
```

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <plurals name="numberOfSongsAvailable">
        <item quantity="one">Znaleziono jedną piosenkę.</item>
        <item quantity="few">Znaleziono %d piosenki.</item>
        <item quantity="other">Znaleziono %d piosenek.</item>
    </plurals>
</resources>
```

Java code:

```
int count = getNumberofsongsAvailable();
Resources res = getResources();
String songsFound = res.getQuantityString(R.plurals.numberOfSongsAvailable,
```

When using the [getQuantityString\(\)](#) method, you need to pass the count twice if your string includes [string formatting](#) with a number. For example, for the string %d songs found, the first count parameter selects the appropriate plural string and the second count parameter is inserted into the %d placeholder. If your plural strings do not include string formatting, you don't need to pass the third parameter to [getQuantityString](#).

## Formatting and Styling

Here are a few important things you should know about how to properly format and style your string resources.

### Escaping apostrophes and quotes

If you have an apostrophe or a quote in your string, you must either escape it or enclose the whole string in the other type of enclosing quotes. For example, here are some strings that do and don't work:

```
<string name="good_example">This'll work</string>
<string name="good_example_2">This\'ll also work</string>
<string name="bad_example">This doesn't work</string>
<string name="bad_example_2">XML encodings don't work</string>
```

### Formatting strings

If you need to format your strings using [String.format\(String, Object...\)](#), then you can do so by putting your format arguments in the string resource. For example, with the following resource:

```
<string name="welcome_messages">Hello, %1$s! You have %2$d new messages.</string>
```

In this example, the format string has two arguments: %1\$s is a string and %2\$d is a decimal number. You can format the string with arguments from your application like this:

```
Resources res = getResources\(\);
String text = String.format(res.getString(R.string.welcome_messages), username,
```

### Styling with HTML markup

You can add styling to your strings with HTML markup. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="welcome">Welcome to <b>Android</b>!</string>
</resources>
```

Supported HTML elements include:

- `<b>` for **bold** text.
- `<i>` for *italic* text.
- `<u>` for underline text.

Sometimes you may want to create a styled text resource that is also used as a format string. Normally, this won't work because the [String.format\(String, Object...\)](#) method will strip all the style information from the string. The work-around to this is to write the HTML tags with escaped entities, which are then recovered with [fromHtml\(String\)](#), after the formatting takes place. For example:

1. Store your styled text resource as an HTML-escaped string:

```
<resources>
    <string name="welcome_messages">Hello, %1$s! You have &lt;b>%2$d new me
</resources>
```

In this formatted string, a `<b>` element is added. Notice that the opening bracket is HTML-escaped, using the `&lt;` notation.

2. Then format the string as usual, but also call `fromHtml(String)` to convert the HTML text into styled text:

```
Resources res = getResources();
String text = String.format(res.getString(R.string.welcome_messages), use
CharSequence styledText = Html.fromHtml(text);
```

Because the `fromHtml(String)` method will format all HTML entities, be sure to escape any possible HTML characters in the strings you use with the formatted text, using `htmlEncode(String)`. For instance, if you'll be passing a string argument to `String.format()` that may contain characters such as "<" or "&", then they must be escaped before formatting, so that when the formatted string is passed through `fromHtml(String)`, the characters come out the way they were originally written. For example:

```
String escapedUsername = TextUtil.htmlEncode(username);
```

```
Resources res = getResources();
String text = String.format(res.getString(R.string.welcome_messages), escapedUs
CharSequence styledText = Html.fromHtml(text);
```

# Style Resource

## See also

1. [Styles and Themes](#)

A style resource defines the format and look for a UI. A style can be applied to an individual [View](#) (from within a layout file) or to an entire [Activity](#) or application (from within the manifest file).

For more information about creating and applying styles, please read [Styles and Themes](#).

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

### file location:

`res/values/filename.xml`

The filename is arbitrary. The element's name will be used as the resource ID.

### resource reference:

In XML: `@[package:]style/style_name`

### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style
        name="style_name"
        parent="@[package:]style/style_to_inherit">
        <item
            name="[package:]style_property_name"
            >style_value</item>
    </style>
</resources>
```

### elements:

`<resources>`

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

No attributes.

`<style>`

Defines a single style. Contains `<item>` elements.

attributes:

`name`

*String.* **Required.** A name for the style, which is used as the resource ID to apply the style to a View, Activity, or application.

`parent`

*Style resource.* Reference to a style from which this style should inherit style properties.

## <item>

Defines a single property for the style. Must be a child of a <style> element.

attributes:

### name

*Attribute resource.* **Required.** The name of the style property to be defined, with a package prefix if necessary (for example android:textColor).

## example:

### XML file for the style (saved in `res/values/`):

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CustomText" parent="@style/Text">
        <item name="android:textSize">20sp</item>
        <item name="android:textColor">#008</item>
    </style>
</resources>
```

### XML file that applies the style to a [TextView](#) (saved in `res/layout/`):

```
<?xml version="1.0" encoding="utf-8"?>
<EditText
    style="@style/CustomText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Hello, World!" />
```

# More Resource Types

This page defines more types of resources you can externalize, including:

## Bool

XML resource that carries a boolean value.

## Color

XML resource that carries a color value (a hexadecimal color).

## Dimension

XML resource that carries a dimension value (with a unit of measure).

## ID

XML resource that provides a unique identifier for application resources and components.

## Integer

XML resource that carries an integer value.

## Integer Array

XML resource that provides an array of integers.

## Typed Array

XML resource that provides a [TypedArray](#) (which you can use for an array of drawables).

# Bool

A boolean value defined in XML.

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

### file location:

`res/values/filename.xml`

The filename is arbitrary. The <bool> element's name will be used as the resource ID.

### resource reference:

In Java: `R.bool.bool_name`

In XML: `@[package:]bool/bool_name`

### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool
        name="bool_name"
        >[true | false]</bool>
</resources>
```

### elements:

<**resources**>

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

No attributes.

### <bool>

A boolean value: true or false.

attributes:

#### **name**

*String*. A name for the bool value. This will be used as the resource ID.

### example:

XML file saved at res/values-small/bools.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="screen_small">true</bool>
    <bool name="adjust_view_bounds">true</bool>
</resources>
```

This application code retrieves the boolean:

```
Resources res = getResources\(\);
boolean screenIsSmall = res.getBoolean(R.bool.screen_small);
```

This layout XML uses the boolean for an attribute:

```
<ImageView
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    android:src="@drawable/logo"
    android:adjustViewBounds="@bool/adjust_view_bounds" />
```

## Color

A color value defined in XML. The color is specified with an RGB value and alpha channel. You can use a color resource anywhere that accepts a hexadecimal color value. You can also use a color resource when a drawable resource is expected in XML (for example, android:drawable="@color/green").

The value always begins with a pound (#) character and then followed by the Alpha-Red-Green-Blue information in one of the following formats:

- #RGB
- #ARGB
- #RRGGBB
- #AARRGGBB

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

### file location:

res/values/colors.xml

The filename is arbitrary. The <color> element's name will be used as the resource ID.

## **resource reference:**

In Java: R.color.color\_name

In XML: @*[package:]color/color\_name*

## **syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color>
        name="color_name"
        >hex_color</color>
</resources>
```

## **elements:**

**<resources>**

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

No attributes.

**<color>**

A color expressed in hexadecimal, as described above.

attributes:

**name**

*String.* A name for the color. This will be used as the resource ID.

## **example:**

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="opaque_red">#f00</color>
    <color name="translucent_red">#80ff0000</color>
</resources>
```

This application code retrieves the color resource:

```
Resources res = getResources();
int color = res.getColor(R.color.opaque_red);
```

This layout XML applies the color to an attribute:

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="@color/translucent_red"
    android:text="Hello"/>
```

# **Dimension**

A dimension value defined in XML. A dimension is specified with a number followed by a unit of measure. For example: 10px, 2in, 5sp. The following units of measure are supported by Android:

## **dp**

Density-independent Pixels - An abstract unit that is based on the physical density of the screen. These units are relative to a 160 dpi (dots per inch) screen, on which 1dp is roughly equal to 1px. When running on a higher density screen, the number of pixels used to draw 1dp is scaled up by a factor appropriate for the screen's dpi. Likewise, when on a lower density screen, the number of pixels used for 1dp is scaled down. The ratio of dp-to-pixel will change with the screen density, but not necessarily in direct proportion. Using dp units (instead of px units) is a simple solution to making the view dimensions in your layout resize properly for different screen densities. In other words, it provides consistency for the real-world sizes of your UI elements across different devices.

## **sp**

Scale-independent Pixels - This is like the dp unit, but it is also scaled by the user's font size preference. It is recommended you use this unit when specifying font sizes, so they will be adjusted for both the screen density and the user's preference.

## **pt**

Points - 1/72 of an inch based on the physical size of the screen.

## **px**

Pixels - Corresponds to actual pixels on the screen. This unit of measure is not recommended because the actual representation can vary across devices; each device may have a different number of pixels per inch and may have more or fewer total pixels available on the screen.

## **mm**

Millimeters - Based on the physical size of the screen.

## **in**

Inches - Based on the physical size of the screen.

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

### **file location:**

`res/values/filename.xml`

The filename is arbitrary. The `<dimen>` element's name will be used as the resource ID.

### **resource reference:**

In Java: `R.dimen.dimension_name`

In XML: `@[package:]dimen/dimension_name`

### **syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen
        name="dimension_name"
        >dimension</dimen>
</resources>
```

### **elements:**

`<resources>`

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

No attributes.

## <dimen>

A dimension, represented by a float, followed by a unit of measurement (dp, sp, pt, px, mm, in), as described above.

attributes:

### **name**

*String*. A name for the dimension. This will be used as the resource ID.

### **example:**

XML file saved at `res/values/dimens.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="textview_height">25dp</dimen>
    <dimen name="textview_width">150dp</dimen>
    <dimen name="ball_radius">30dp</dimen>
    <dimen name="font_size">16sp</dimen>
</resources>
```

This application code retrieves a dimension:

```
Resources res = getResources\(\);
float fontSize = res.getDimension(R.dimen.font_size);
```

This layout XML applies dimensions to attributes:

```
<TextView
    android:layout_height="@dimen/textview_height"
    android:layout_width="@dimen/textview_width"
    android:textSize="@dimen/font_size"/>
```

## ID

A unique resource ID defined in XML. Using the name you provide in the `<item>` element, the Android developer tools create a unique integer in your project's `R.java` class, which you can use as an identifier for an application resources (for example, a `View` in your UI layout) or a unique integer for use in your application code (for example, as an ID for a dialog or a result code).

**Note:** An ID is a simple resource that is referenced using the value provided in the `name` attribute (not the name of the XML file). As such, you can combine ID resources with other simple resources in the one XML file, under one `<resources>` element. Also, remember that an ID resources does not reference an actual resource item; it is simply a unique ID that you can attach to other resources or use as a unique integer in your application.

### **file location:**

`res/values/filename.xml`

The filename is arbitrary.

### **resource reference:**

In Java: `R.id.name`

In XML: `@[package:]id/name`

## syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item
        type="id"
        name="id_name" />
</resources>
```

## elements:

### <resources>

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

No attributes.

### <item>

Defines a unique ID. Takes no value, only attributes.

attributes:

#### **type**

Must be "id".

#### **name**

*String.* A unique name for the ID.

## example:

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item type="id" name="button_ok" />
    <item type="id" name="dialog_exit" />
</resources>
```

Then, this layout snippet uses the "button\_ok" ID for a Button widget:

```
<Button android:id="@+id/button_ok"
        style="@style/button_style" />
```

Notice that the android:id value does not include the plus sign in the ID reference, because the ID already exists, as defined in the ids.xml example above. (When you specify an ID to an XML resource using the plus sign—in the format android:id="@+id/name"—it means that the "name" ID does not exist and should be created.)

As another example, the following code snippet uses the "dialog\_exit" ID as a unique identifier for a dialog:

```
showDialog(R.id.dialog_exit);
```

In the same application, the "dialog\_exit" ID is compared when creating a dialog:

```
protected Dialog onCreateDialog(int) (int id) {
    Dialog dialog;
```

```

switch(id) {
    case R.id.dialog_exit:
        ...
        break;
    default:
        dialog = null;
    }
    return dialog;
}

```

## Integer

An integer defined in XML.

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

### file location:

`res/values/filename.xml`

The filename is arbitrary. The `<integer>` element's name will be used as the resource ID.

### resource reference:

In Java: `R.integer.integer_name`

In XML: `@[package:]integer/integer_name`

### syntax:

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <integer
        name="integer_name"
        >integer</integer>
</resources>

```

### elements:

`<resources>`

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

No attributes.

`<integer>`

An integer.

attributes:

`name`

*String*. A name for the integer. This will be used as the resource ID.

### example:

XML file saved at `res/values/integers.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <integer name="max_speed">75</integer>
    <integer name="min_speed">5</integer>
</resources>
```

This application code retrieves an integer:

```
Resources res = getResources();
int maxSpeed = res.getInteger(R.integer.max_speed);
```

## Integer Array

An array of integers defined in XML.

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

### file location:

res/values/*filename*.xml

The filename is arbitrary. The <integer-array> element's name will be used as the resource ID.

### compiled resource datatype:

Resource pointer to an array of integers.

### resource reference:

In Java: R.array.integer\_array\_name

In XML: @ [package:]array.integer\_array\_name

### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <integer-array
        name="integer_array_name">
        <item
            >integer</item>
    </integer-array>
</resources>
```

### elements:

<resources>

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

No attributes.

<integer-array>

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

attributes:

**android:name**

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

### <item>

An integer. The value can be a reference to another integer resource. Must be a child of a <integer-array> element.

No attributes.

#### example:

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <integer-array name="bits">
        <item>4</item>
        <item>8</item>
        <item>16</item>
        <item>32</item>
    </integer-array>
</resources>
```

This application code retrieves the integer array:

```
Resources res = getResources\(\);
int[] bits = res.getIntArray(R.array.bits);
```

## Typed Array

A [TypedArray](#) defined in XML. You can use this to create an array of other resources, such as drawables. Note that the array is not required to be homogeneous, so you can create an array of mixed resource types, but you must be aware of what and where the data types are in the array so that you can properly obtain each item with the [TypedArray](#)'s get...() methods.

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

#### file location:

res/values/*filename*.xml

The filename is arbitrary. The <array> element's name will be used as the resource ID.

#### compiled resource datatype:

Resource pointer to a [TypedArray](#).

#### resource reference:

In Java: R.array.array\_name

In XML: @ [package:]array.array\_name

#### syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <array
        name="integer_array_name">
            <item>resource</item>
```

```
</array>
</resources>
```

#### elements:

##### **<resources>**

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

No attributes.

##### **<array>**

Defines an array. Contains one or more child **<item>** elements.

attributes:

###### **android:name**

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

##### **<item>**

A generic resource. The value can be a reference to a resource or a simple data type. Must be a child of an **<array>** element.

No attributes.

#### example:

XML file saved at `res/values/arrays.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <array name="icons">
        <item>@drawable/home</item>
        <item>@drawable/settings</item>
        <item>@drawable/logout</item>
    </array>
    <array name="colors">
        <item>#FFFF0000</item>
        <item>#FF00FF00</item>
        <item>#FF0000FF</item>
    </array>
</resources>
```

This application code retrieves each array and then obtains the first entry in each array:

```
Resources res = getResources();
TypedArray icons = res.obtainTypedArray(R.array.icons);
Drawable drawable = icons.getDrawable(0);

TypedArray colors = res.obtainTypedArray(R.array.colors);
int color = colors.getColor(0, 0);
```

# Animation and Graphics

Make your apps look and perform their best using Android's powerful graphics features such as OpenGL, hardware acceleration, and built-in UI animations.

## Blog Articles

### [Android 4.0 Graphics and Animations](#)

Earlier this year, Android 3.0 launched with a new 2D rendering pipeline designed to support hardware acceleration on tablets. With this new pipeline, all drawing operations performed by the UI toolkit are carried out using the GPU. You'll be happy to hear that Android 4.0, Ice Cream Sandwich, brings an improved version of the hardware-accelerated 2D rendering pipeline.

### [Introducing ViewPropertyAnimator](#)

This new animation system makes it easy to animate any kind of property on any object, including the new properties added to the View class in 3.0. In the 3.1 release, we added a small utility class that makes animating these properties even easier.

### [Android 3.0 Hardware Acceleration](#)

Hardware accelerated graphics is nothing new to the Android platform, it has always been used for windows composition or OpenGL games for instance, but with this new rendering pipeline applications can benefit from an extra boost in performance.

## Training

### [Displaying Bitmaps Efficiently](#)

This class covers some common techniques for processing and loading Bitmap objects in a way that keeps your user interface (UI) components responsive and avoids exceeding your application memory limit.

# Animation and Graphics Overview

Android provides a variety of powerful APIs for applying animation to UI elements and drawing custom 2D and 3D graphics. The sections below provide an overview of the APIs and system capabilities available and help you decide which approach is best for your needs.

## Animation

The Android framework provides two animation systems: property animation (introduced in Android 3.0) and view animation. Both animation systems are viable options, but the property animation system, in general, is the preferred method to use, because it is more flexible and offers more features. In addition to these two systems, you can utilize Drawable animation, which allows you to load drawable resources and display them one frame after another.

### Property Animation

Introduced in Android 3.0 (API level 11), the property animation system lets you animate properties of any object, including ones that are not rendered to the screen. The system is extensible and lets you animate properties of custom types as well.

### View Animation

View Animation is the older system and can only be used for Views. It is relatively easy to setup and offers enough capabilities to meet many application's needs.

### Drawable Animation

Drawable animation involves displaying [Drawable](#) resources one after another, like a roll of film. This method of animation is useful if you want to animate things that are easier to represent with Drawable resources, such as a progression of bitmaps.

## 2D and 3D Graphics

When writing an application, it's important to consider exactly what your graphical demands will be. Varying graphical tasks are best accomplished with varying techniques. For example, graphics and animations for a rather static application should be implemented much differently than graphics and animations for an interactive game. Here, we'll discuss a few of the options you have for drawing graphics on Android and which tasks they're best suited for.

### Canvas and Drawables

Android provides a set of [View](#) widgets that provide general functionality for a wide array of user interfaces. You can also extend these widgets to modify the way they look or behave. In addition, you can do your own custom 2D rendering using the various drawing methods contained in the [Canvas](#) class or create [Drawable](#) objects for things such as textured buttons or frame-by-frame animations.

### Hardware Acceleration

Beginning in Android 3.0, you can hardware accelerate the majority of the drawing done by the Canvas APIs to further increase their performance.

### OpenGL

Android supports OpenGL ES 1.0 and 2.0, with Android framework APIs as well as natively with the Native Development Kit (NDK). Using the framework APIs is desirable when you want to add a few graphical enhancements to your application that are not supported with the Canvas APIs, or if you desire platform independence and don't demand high performance. There is a performance hit in using the framework APIs compared to the NDK, so for many graphic intensive applications such as games, using the NDK is beneficial (It is important to note though that you can still get adequate performance using the framework

APIs. For example, the Google Body app is developed entirely using the framework APIs). OpenGL with the NDK is also useful if you have a lot of native code that you want to port over to Android. For more information about using the NDK, read the docs in the `docs` / directory of the [NDK download](#).

# Property Animation

## In this document

1. [How Property Animation Works](#)
2. [Animating with ValueAnimator](#)
3. [Animating with ObjectAnimator](#)
4. [Choreographing Multiple Animations with AnimatorSet](#)
5. [Animation Listeners](#)
6. [Using a TypeEvaluator](#)
7. [Using Interpolators](#)
8. [Specifying Keyframes](#)
9. [Animating Layout Changes to ViewGroups](#)
10. [Animating Views](#)
  1. [ViewPropertyAnimator](#)
11. [Declaring Animations in XML](#)

## Key classes

1. [ValueAnimator](#)
2. [ObjectAnimator](#)
3. [TypeEvaluator](#)

## Related samples

1. [API Demos](#)

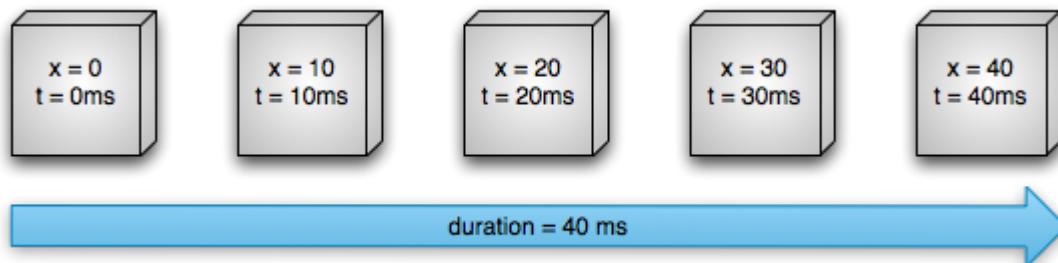
The property animation system is a robust framework that allows you to animate almost anything. You can define an animation to change any object property over time, regardless of whether it draws to the screen or not. A property animation changes a property's (a field in an object) value over a specified length of time. To animate something, you specify the object property that you want to animate, such as an object's position on the screen, how long you want to animate it for, and what values you want to animate between.

The property animation system lets you define the following characteristics of an animation:

- Duration: You can specify the duration of an animation. The default length is 300 ms.
- Time interpolation: You can specify how the values for the property are calculated as a function of the animation's current elapsed time.
- Repeat count and behavior: You can specify whether or not to have an animation repeat when it reaches the end of a duration and how many times to repeat the animation. You can also specify whether you want the animation to play back in reverse. Setting it to reverse plays the animation forwards then backwards repeatedly, until the number of repeats is reached.
- Animator sets: You can group animations into logical sets that play together or sequentially or after specified delays.
- Frame refresh delay: You can specify how often to refresh frames of your animation. The default is set to refresh every 10 ms, but the speed in which your application can refresh frames is ultimately dependent on how busy the system is overall and how fast the system can service the underlying timer.

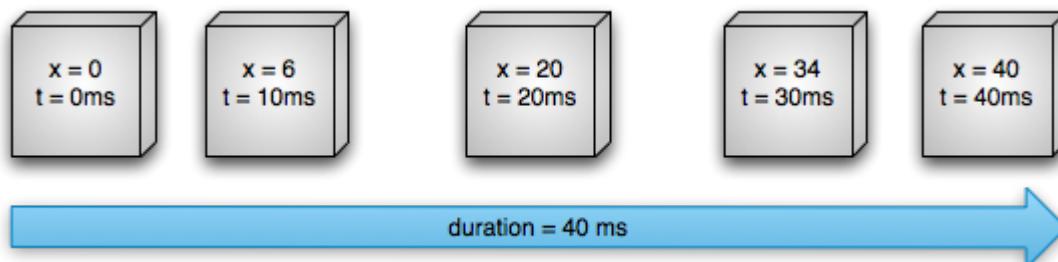
# How Property Animation Works

First, let's go over how an animation works with a simple example. Figure 1 depicts a hypothetical object that is animated with its `x` property, which represents its horizontal location on a screen. The duration of the animation is set to 40 ms and the distance to travel is 40 pixels. Every 10 ms, which is the default frame refresh rate, the object moves horizontally by 10 pixels. At the end of 40ms, the animation stops, and the object ends at horizontal position 40. This is an example of an animation with linear interpolation, meaning the object moves at a constant speed.



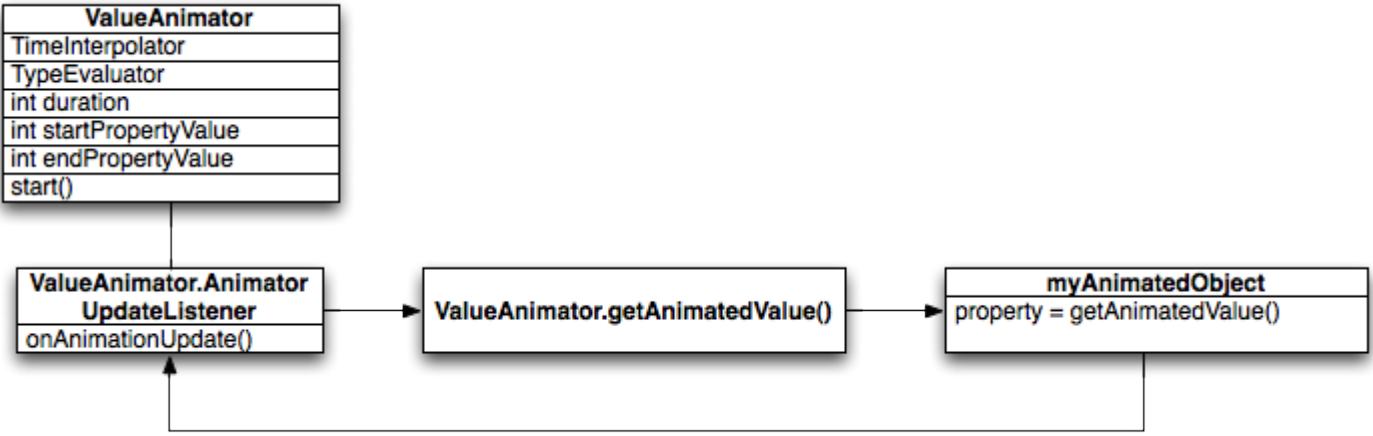
**Figure 1.** Example of a linear animation

You can also specify animations to have a non-linear interpolation. Figure 2 illustrates a hypothetical object that accelerates at the beginning of the animation, and decelerates at the end of the animation. The object still moves 40 pixels in 40 ms, but non-linearly. In the beginning, this animation accelerates up to the halfway point then decelerates from the halfway point until the end of the animation. As Figure 2 shows, the distance traveled at the beginning and end of the animation is less than in the middle.



**Figure 2.** Example of a non-linear animation

Let's take a detailed look at how the important components of the property animation system would calculate animations like the ones illustrated above. Figure 3 depicts how the main classes work with one another.



**Figure 3.** How animations are calculated

The `ValueAnimator` object keeps track of your animation's timing, such as how long the animation has been running, and the current value of the property that it is animating.

The `ValueAnimator` encapsulates a `TimeInterpolator`, which defines animation interpolation, and a `TypeEvaluator`, which defines how to calculate values for the property being animated. For example, in Figure 2, the `TimeInterpolator` used would be `AccelerateDecelerateInterpolator` and the `TypeEvaluator` would be `IntEvaluator`.

To start an animation, create a `ValueAnimator` and give it the starting and ending values for the property that you want to animate, along with the duration of the animation. When you call `start()` the animation begins. During the whole animation, the `ValueAnimator` calculates an *elapsed fraction* between 0 and 1, based on the duration of the animation and how much time has elapsed. The elapsed fraction represents the percentage of time that the animation has completed, 0 meaning 0% and 1 meaning 100%. For example, in Figure 1, the elapsed fraction at  $t = 10$  ms would be .25 because the total duration is  $t = 40$  ms.

When the `ValueAnimator` is done calculating an elapsed fraction, it calls the `TimeInterpolator` that is currently set, to calculate an *interpolated fraction*. An interpolated fraction maps the elapsed fraction to a new fraction that takes into account the time interpolation that is set. For example, in Figure 2, because the animation slowly accelerates, the interpolated fraction, about .15, is less than the elapsed fraction, .25, at  $t = 10$  ms. In Figure 1, the interpolated fraction is always the same as the elapsed fraction.

When the interpolated fraction is calculated, `ValueAnimator` calls the appropriate `TypeEvaluator`, to calculate the value of the property that you are animating, based on the interpolated fraction, the starting value, and the ending value of the animation. For example, in Figure 2, the interpolated fraction was .15 at  $t = 10$  ms, so the value for the property at that time would be  $.15 \times (40 - 0)$ , or 6.

The `com.example.android.apis.animation` package in the [API Demos](#) sample project provides many examples on how to use the property animation system.

## How Property Animation Differs from View Animation

The view animation system provides the capability to only animate `View` objects, so if you wanted to animate non-`View` objects, you have to implement your own code to do so. The view animation system is also constrained in the fact that it only exposes a few aspects of a `View` object to animate, such as the scaling and rotation of a `View` but not the background color, for instance.

Another disadvantage of the view animation system is that it only modified where the View was drawn, and not the actual View itself. For instance, if you animated a button to move across the screen, the button draws correctly, but the actual location where you can click the button does not change, so you have to implement your own logic to handle this.

With the property animation system, these constraints are completely removed, and you can animate any property of any object (Views and non-Views) and the object itself is actually modified. The property animation system is also more robust in the way it carries out animation. At a high level, you assign animators to the properties that you want to animate, such as color, position, or size and can define aspects of the animation such as interpolation and synchronization of multiple animators.

The view animation system, however, takes less time to setup and requires less code to write. If view animation accomplishes everything that you need to do, or if your existing code already works the way you want, there is no need to use the property animation system. It also might make sense to use both animation systems for different situations if the use case arises.

## API Overview

You can find most of the property animation system's APIs in [android.animation](#). Because the view animation system already defines many interpolators in [android.view.animation](#), you can use those interpolators in the property animation system as well. The following tables describe the main components of the property animation system.

The [Animator](#) class provides the basic structure for creating animations. You normally do not use this class directly as it only provides minimal functionality that must be extended to fully support animating values. The following subclasses extend [Animator](#):

**Table 1.** Animators

Class	Description
<a href="#">ValueAnimator</a>	The main timing engine for property animation that also computes the values for the property to be animated. It has all of the core functionality that calculates animation values and contains the timing details of each animation, information about whether an animation repeats, listeners that receive update events, and the ability to set custom types to evaluate. There are two pieces to animating properties: calculating the animated values and setting those values on the object and property that is being animated. <a href="#">ValueAnimator</a> does not carry out the second piece, so you must listen for updates to values calculated by the <a href="#">ValueAnimator</a> and modify the objects that you want to animate with your own logic. See the section about <a href="#">Animating with ValueAnimator</a> for more information.
<a href="#">ObjectAnimator</a>	A subclass of <a href="#">ValueAnimator</a> that allows you to set a target object and object property to animate. This class updates the property accordingly when it computes a new value for the animation. You want to use <a href="#">ObjectAnimator</a> most of the time, because it makes the process of animating values on target objects much easier. However, you sometimes want to use <a href="#">ValueAnimator</a> directly because <a href="#">ObjectAnimator</a> has a few more restrictions, such as requiring specific accessor methods to be present on the target object.
<a href="#">AnimatorSet</a>	Provides a mechanism to group animations together so that they run in relation to one another. You can set animations to play together, sequentially, or after a specified delay. See the section about <a href="#">Choreographing multiple animations with Animator Sets</a> for more information.

Evaluators tell the property animation system how to calculate values for a given property. They take the timing data that is provided by an [Animator](#) class, the animation's start and end value, and calculate the animated values of the property based on this data. The property animation system provides the following evaluators:

**Table 2.** Evaluators

Class/Interface	Description
<a href="#">IntEvaluator</a>	The default evaluator to calculate values for <code>int</code> properties.
<a href="#">FloatEvaluator</a>	The default evaluator to calculate values for <code>float</code> properties.
<a href="#">ArgbEvaluator</a>	The default evaluator to calculate values for color properties that are represented as hex-decimal values.
<a href="#">TypeEvaluator</a>	An interface that allows you to create your own evaluator. If you are animating an object property that is <i>not</i> an <code>int</code> , <code>float</code> , or color, you must implement the <a href="#">TypeEvaluator</a> interface to specify how to compute the object property's animated values. You can also specify a custom <a href="#">TypeEvaluator</a> for <code>int</code> , <code>float</code> , and color values as well, if you want to process those types differently than the default behavior. See the section about <a href="#">Using a TypeEvaluator</a> for more information on how to write a custom evaluator.

A time interpolator defines how specific values in an animation are calculated as a function of time. For example, you can specify animations to happen linearly across the whole animation, meaning the animation moves evenly the entire time, or you can specify animations to use non-linear time, for example, accelerating at the beginning and decelerating at the end of the animation. Table 3 describes the interpolators that are contained in [android.view.animation](#). If none of the provided interpolators suits your needs, implement the [TimeInterpolator](#) interface and create your own. See [Using interpolators](#) for more information on how to write a custom interpolator.

**Table 3.** Interpolators

Class/Interface	Description
<a href="#">AccelerateDecelerateInterpolator</a>	An interpolator whose rate of change starts and ends slowly but accelerates through the middle.
<a href="#">AccelerateInterpolator</a>	An interpolator whose rate of change starts out slowly and then accelerates.
<a href="#">AnticipateInterpolator</a>	An interpolator whose change starts backward then flings forward.
<a href="#">AnticipateOvershootInterpolator</a>	An interpolator whose change starts backward, flings forward and overshoots the target value, then finally goes back to the final value.
<a href="#">BounceInterpolator</a>	An interpolator whose change bounces at the end.
<a href="#">CycleInterpolator</a>	An interpolator whose animation repeats for a specified number of cycles.
<a href="#">DecelerateInterpolator</a>	An interpolator whose rate of change starts out quickly and then decelerates.
<a href="#">LinearInterpolator</a>	An interpolator whose rate of change is constant.
<a href="#">OvershootInterpolator</a>	An interpolator whose change flings forward and overshoots the last value then comes back.
<a href="#">TimeInterpolator</a>	An interface that allows you to implement your own interpolator.

## Animating with ValueAnimator

The [ValueAnimator](#) class lets you animate values of some type for the duration of an animation by specifying a set of int, float, or color values to animate through. You obtain a [ValueAnimator](#) by calling one of its factory methods: [ofInt\(\)](#), [ofFloat\(\)](#), or [ofObject\(\)](#). For example:

```
ValueAnimator animation = ValueAnimator.ofFloat(0f, 1f);  
animation.setDuration(1000);  
animation.start();
```

In this code, the [ValueAnimator](#) starts calculating the values of the animation, between 0 and 1, for a duration of 1000 ms, when the [start\(\)](#) method runs.

You can also specify a custom type to animate by doing the following:

```
ValueAnimator animation = ValueAnimator.ofObject(new MyTypeEvaluator(), startPr  
animation.setDuration(1000);  
animation.start();
```

In this code, the [ValueAnimator](#) starts calculating the values of the animation, between [startProperty-Value](#) and [endProperty-Value](#) using the logic supplied by [MyTypeEvaluator](#) for a duration of 1000 ms, when the [start\(\)](#) method runs.

The previous code snippets, however, has no real effect on an object, because the [ValueAnimator](#) does not operate on objects or properties directly. The most likely thing that you want to do is modify the objects that you want to animate with these calculated values. You do this by defining listeners in the [ValueAnimator](#) to appropriately handle important events during the animation's lifespan, such as frame updates. When implementing the listeners, you can obtain the calculated value for that specific frame refresh by calling [getAnimated-Value\(\)](#). For more information on listeners, see the section about [Animation Listeners](#).

## Animating with ObjectAnimator

The [ObjectAnimator](#) is a subclass of the [ValueAnimator](#) (discussed in the previous section) and combines the timing engine and value computation of [ValueAnimator](#) with the ability to animate a named property of a target object. This makes animating any object much easier, as you no longer need to implement the [ValueAnimator.AnimatorUpdateListener](#), because the animated property updates automatically.

Instantiating an [ObjectAnimator](#) is similar to a [ValueAnimator](#), but you also specify the object and the name of that object's property (as a String) along with the values to animate between:

```
ObjectAnimator anim = ObjectAnimator.ofFloat(foo, "alpha", 0f, 1f);  
anim.setDuration(1000);  
anim.start();
```

To have the [ObjectAnimator](#) update properties correctly, you must do the following:

- The object property that you are animating must have a setter function (in camel case) in the form of `set<propertyName>()`. Because the [ObjectAnimator](#) automatically updates the property during animation, it must be able to access the property with this setter method. For example, if the property name is `foo`, you need to have a `setFoo()` method. If this setter method does not exist, you have three options:
  - Add the setter method to the class if you have the rights to do so.

- Use a wrapper class that you have rights to change and have that wrapper receive the value with a valid setter method and forward it to the original object.
- Use [ValueAnimator](#) instead.
- If you specify only one value for the `values...` parameter in one of the [ObjectAnimator](#) factory methods, it is assumed to be the ending value of the animation. Therefore, the object property that you are animating must have a getter function that is used to obtain the starting value of the animation. The getter function must be in the form of `get<propertyName>()`. For example, if the property name is `foo`, you need to have a `getFoo()` method.
- The getter (if needed) and setter methods of the property that you are animating must operate on the same type as the starting and ending values that you specify to [ObjectAnimator](#). For example, you must have `targetObject.setPropName(float)` and `targetObject.getPropName(float)` if you construct the following [ObjectAnimator](#):

```
ObjectAnimator.ofFloat(targetObject, "propName", 1f)
```

- Depending on what property or object you are animating, you might need to call the [invalidate\(\)](#) method on a View to force the screen to redraw itself with the updated animated values. You do this in the [onAnimationUpdate\(\)](#) callback. For example, animating the color property of a Drawable object only cause updates to the screen when that object redraws itself. All of the property setters on View, such as [setAlpha\(\)](#) and [setTranslationX\(\)](#) invalidate the View properly, so you do not need to invalidate the View when calling these methods with new values. For more information on listeners, see the section about [Animation Listeners](#).

## Choreographing Multiple Animations with AnimatorSet

In many cases, you want to play an animation that depends on when another animation starts or finishes. The Android system lets you bundle animations together into an [AnimatorSet](#), so that you can specify whether to start animations simultaneously, sequentially, or after a specified delay. You can also nest [AnimatorSet](#) objects within each other.

The following sample code taken from the [Bouncing Balls](#) sample (modified for simplicity) plays the following [Animator](#) objects in the following manner:

1. Plays `bounceAnim`.
2. Plays `squashAnim1`, `squashAnim2`, `stretchAnim1`, and `stretchAnim2` at the same time.
3. Plays `bounceBackAnim`.
4. Plays `fadeAnim`.

```
AnimatorSet bouncer = new AnimatorSet();
bouncer.play(bounceAnim).before(squashAnim1);
bouncer.play(squashAnim1).with(squashAnim2);
bouncer.play(squashAnim1).with(stretchAnim1);
bouncer.play(squashAnim1).with(stretchAnim2);
bouncer.play(bounceBackAnim).after(stretchAnim2);
ValueAnimator fadeAnim = ObjectAnimator.ofFloat(newBall, "alpha", 1f, 0f);
fadeAnim.setDuration(250);
AnimatorSet animatorSet = new AnimatorSet();
animatorSet.play(bouncer).before(fadeAnim);
animatorSet.start();
```

For a more complete example on how to use animator sets, see the [Bouncing Balls](#) sample in APIDemos.

# Animation Listeners

You can listen for important events during an animation's duration with the listeners described below.

- [Animator.AnimatorListener](#)
  - [onAnimationStart\(\)](#) - Called when the animation starts.
  - [onAnimationEnd\(\)](#) - Called when the animation ends.
  - [onAnimationRepeat\(\)](#) - Called when the animation repeats itself.
  - [onAnimationCancel\(\)](#) - Called when the animation is canceled. A cancelled animation also calls [onAnimationEnd\(\)](#), regardless of how they were ended.
- [ValueAnimator.AnimatorUpdateListener](#)
  - [onAnimationUpdate\(\)](#) - called on every frame of the animation. Listen to this event to use the calculated values generated by [ValueAnimator](#) during an animation. To use the value, query the [ValueAnimator](#) object passed into the event to get the current animated value with the [getAnimatedValue\(\)](#) method. Implementing this listener is required if you use [ValueAnimator](#).

Depending on what property or object you are animating, you might need to call [invalidate\(\)](#) on a View to force that area of the screen to redraw itself with the new animated values. For example, animating the color property of a Drawable object only cause updates to the screen when that object redraws itself. All of the property setters on View, such as [setAlpha\(\)](#) and [setTranslationX\(\)](#) invalidate the View properly, so you do not need to invalidate the View when calling these methods with new values.

You can extend the [AnimatorListenerAdapter](#) class instead of implementing the [Animator.AnimatorListener](#) interface, if you do not want to implement all of the methods of the [Animator.AnimatorListener](#) interface. The [AnimatorListenerAdapter](#) class provides empty implementations of the methods that you can choose to override.

For example, the [Bouncing Balls](#) sample in the API demos creates an [AnimatorListenerAdapter](#) for just the [onAnimationEnd\(\)](#) callback:

```
ValueAnimator fadeAnim = ObjectAnimator.ofFloat(newBall, "alpha", 1f, 0f);
fadeAnim.setDuration(250);
fadeAnim.addListener(new AnimatorListenerAdapter() {
    public void onAnimationEnd(Animator animation) {
        balls.remove(((ObjectAnimator)animation).getTarget());
    }
})
```

# Animating Layout Changes to ViewGroups

The property animation system provides the capability to animate changes to ViewGroup objects as well as provide an easy way to animate View objects themselves.

You can animate layout changes within a ViewGroup with the [LayoutTransition](#) class. Views inside a ViewGroup can go through an appearing and disappearing animation when you add them to or remove them from a ViewGroup or when you call a View's [setVisibility\(\)](#) method with [VISIBLE](#), [android.view.View#INVISIBLE](#), or [GONE](#). The remaining Views in the ViewGroup can also animate into their new positions when you add or remove Views. You can define the following animations in a [LayoutTransition](#) object by calling [setAnimator\(\)](#) and passing in an [Animator](#) object with one of the following [LayoutTransition](#) constants:

- APPEARING - A flag indicating the animation that runs on items that are appearing in the container.
- CHANGE\_APPEARING - A flag indicating the animation that runs on items that are changing due to a new item appearing in the container.
- DISAPPEARING - A flag indicating the animation that runs on items that are disappearing from the container.
- CHANGE\_DISAPPEARING - A flag indicating the animation that runs on items that are changing due to an item disappearing from the container.

You can define your own custom animations for these four types of events to customize the look of your layout transitions or just tell the animation system to use the default animations.

The [LayoutAnimations](#) sample in API Demos shows you how to define animations for layout transitions and then set the animations on the View objects that you want to animate.

The [LayoutAnimationsByDefault](#) and its corresponding [layout\\_animations\\_by\\_default.xml](#) layout resource file show you how to enable the default layout transitions for ViewGroups in XML. The only thing that you need to do is to set the `android:animateLayoutChanges` attribute to `true` for the ViewGroup. For example:

```
<LinearLayout
    android:orientation="vertical"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:id="@+id/verticalContainer"
    android:animateLayoutChanges="true" />
```

Setting this attribute to true automatically animates Views that are added or removed from the ViewGroup as well as the remaining Views in the ViewGroup.

## Using a TypeEvaluator

If you want to animate a type that is unknown to the Android system, you can create your own evaluator by implementing the [TypeEvaluator](#) interface. The types that are known by the Android system are `int`, `float`, or a color, which are supported by the [IntEvaluator](#), [FloatEvaluator](#), and [ArgbEvaluator](#) type evaluators.

There is only one method to implement in the [TypeEvaluator](#) interface, the `evaluate()` method. This allows the animator that you are using to return an appropriate value for your animated property at the current point of the animation. The [FloatEvaluator](#) class demonstrates how to do this:

```
public class FloatEvaluator implements TypeEvaluator {

    public Object evaluate(float fraction, Object startValue, Object endValue)
        float startFloat = ((Number) startValue).floatValue();
        return startFloat + fraction * (((Number) endValue).floatValue() - startFloat);
}
```

**Note:** When [ValueAnimator](#) (or [ObjectAnimator](#)) runs, it calculates a current elapsed fraction of the animation (a value between 0 and 1) and then calculates an interpolated version of that depending on what interpolator that you are using. The interpolated fraction is what your [TypeEvaluator](#) receives through the `fraction` parameter, so you do not have to take into account the interpolator when calculating animated values.

# Using Interpolators

An interpolator defines how specific values in an animation are calculated as a function of time. For example, you can specify animations to happen linearly across the whole animation, meaning the animation moves evenly the entire time, or you can specify animations to use non-linear time, for example, using acceleration or deceleration at the beginning or end of the animation.

Interpolators in the animation system receive a fraction from Animators that represent the elapsed time of the animation. Interpolators modify this fraction to coincide with the type of animation that it aims to provide. The Android system provides a set of common interpolators in the [android.view.animation package](#). If none of these suit your needs, you can implement the [TimeInterpolator](#) interface and create your own.

As an example, how the default interpolator [AccelerateDecelerateInterpolator](#) and the [LinearInterpolator](#) calculate interpolated fractions are compared below. The [LinearInterpolator](#) has no effect on the elapsed fraction. The [AccelerateDecelerateInterpolator](#) accelerates into the animation and decelerates out of it. The following methods define the logic for these interpolators:

## AccelerateDecelerateInterpolator

```
public float getInterpolation(float input) {
    return (float)(Math.cos((input + 1) * Math.PI) / 2.0f) + 0.5f;
}
```

## LinearInterpolator

```
public float getInterpolation(float input) {
    return input;
}
```

The following table represents the approximate values that are calculated by these interpolators for an animation that lasts 1000ms:

ms elapsed	Elapsed fraction/Interpolated fraction (Linear)	Interpolated fraction (Accelerate/Decelerate)
0	0	0
200	.2	.1
400	.4	.345
600	.6	.8
800	.8	.9
1000	1	1

As the table shows, the [LinearInterpolator](#) changes the values at the same speed, .2 for every 200ms that passes. The [AccelerateDecelerateInterpolator](#) changes the values faster than [LinearInterpolator](#) between 200ms and 600ms and slower between 600ms and 1000ms.

# Specifying Keyframes

A [Keyframe](#) object consists of a time/value pair that lets you define a specific state at a specific time of an animation. Each keyframe can also have its own interpolator to control the behavior of the animation in the interval between the previous keyframe's time and the time of this keyframe.

To instantiate a [Keyframe](#) object, you must use one of the factory methods, [ofInt\(\)](#), [ofFloat\(\)](#), or [ofObject\(\)](#) to obtain the appropriate type of [Keyframe](#). You then call the [ofKeyframe\(\)](#) factory

method to obtain a [PropertyValuesHolder](#) object. Once you have the object, you can obtain an animator by passing in the [PropertyValuesHolder](#) object and the object to animate. The following code snippet demonstrates how to do this:

```
Keyframe kf0 = Keyframe.ofFloat(0f, 0f);
Keyframe kf1 = Keyframe.ofFloat(.5f, 360f);
Keyframe kf2 = Keyframe.ofFloat(1f, 0f);
PropertyValuesHolder pvhRotation = PropertyValuesHolder.ofKeyframe("rotation",
ObjectAnimator rotationAnim = ObjectAnimator.ofPropertyValuesHolder(target, pvh,
rotationAnim.setDuration(5000ms);
```

For a more complete example on how to use keyframes, see the [MultiPropertyAnimation](#) sample in APIDemos.

## Animating Views

The property animation system allow streamlined animation of View objects and offers a few advantages over the view animation system. The view animation system transformed View objects by changing the way that they were drawn. This was handled in the container of each View, because the View itself had no properties to manipulate. This resulted in the View being animated, but caused no change in the View object itself. This led to behavior such as an object still existing in its original location, even though it was drawn on a different location on the screen. In Android 3.0, new properties and the corresponding getter and setter methods were added to eliminate this drawback.

The property animation system can animate Views on the screen by changing the actual properties in the View objects. In addition, Views also automatically call the [invalidate\(\)](#) method to refresh the screen whenever its properties are changed. The new properties in the [View](#) class that facilitate property animations are:

- `translationX` and `translationY`: These properties control where the View is located as a delta from its left and top coordinates which are set by its layout container.
- `rotation`, `rotationX`, and `rotationY`: These properties control the rotation in 2D (`rotation` property) and 3D around the pivot point.
- `scaleX` and `scaleY`: These properties control the 2D scaling of a View around its pivot point.
- `pivotX` and `pivotY`: These properties control the location of the pivot point, around which the rotation and scaling transforms occur. By default, the pivot point is located at the center of the object.
- `x` and `y`: These are simple utility properties to describe the final location of the View in its container, as a sum of the left and top values and `translationX` and `translationY` values.
- `alpha`: Represents the alpha transparency on the View. This value is 1 (opaque) by default, with a value of 0 representing full transparency (not visible).

To animate a property of a View object, such as its color or rotation value, all you need to do is create a property animator and specify the View property that you want to animate. For example:

```
ObjectAnimator.ofFloat(myView, "rotation", 0f, 360f);
```

For more information on creating animators, see the sections on animating with [ValueAnimator](#) and [ObjectAnimator](#).

## Animating with ViewPropertyAnimator

The [ViewPropertyAnimator](#) provides a simple way to animate several properties of a [View](#) in parallel, using a single underlying [Animator](#) object. It behaves much like an [ObjectAnimator](#), because it modifies the actual values of the view's properties, but is more efficient when animating many properties at once. In addition, the code for using the [ViewPropertyAnimator](#) is much more concise and easier to read. The fol-

lowing code snippets show the differences in using multiple [ObjectAnimator](#) objects, a single [ObjectAnimator](#), and the [ViewPropertyAnimator](#) when simultaneously animating the x and y property of a view.

## Multiple ObjectAnimator objects

```
ObjectAnimator animX = ObjectAnimator.ofFloat(myView, "x", 50f);
ObjectAnimator animY = ObjectAnimator.ofFloat(myView, "y", 100f);
AnimatorSet animSetXY = new AnimatorSet();
animSetXY.playTogether(animX, animY);
animSetXY.start();
```

## One ObjectAnimator

```
PropertyValuesHolder pvhX = PropertyValuesHolder.ofFloat("x", 50f);
PropertyValuesHolder pvhY = PropertyValuesHolder.ofFloat("y", 100f);
ObjectAnimator.ofPropertyValuesHolder(myView, pvhX, pvhY).start();
```

## ViewPropertyAnimator

```
myView.animate().x(50f).y(100f);
```

For more detailed information about [ViewPropertyAnimator](#), see the corresponding Android Developers [blog post](#).

## Declaring Animations in XML

The property animation system lets you declare property animations with XML instead of doing it programmatically. By defining your animations in XML, you can easily reuse your animations in multiple activities and more easily edit the animation sequence.

To distinguish animation files that use the new property animation APIs from those that use the legacy [view animation](#) framework, starting with Android 3.1, you should save the XML files for property animations in the `res/animator/` directory (instead of `res/anim/`). Using the `animator` directory name is optional, but necessary if you want to use the layout editor tools in the Eclipse ADT plugin (ADT 11.0.0+), because ADT only searches the `res/animator/` directory for property animation resources.

The following property animation classes have XML declaration support with the following XML tags:

- [ValueAnimator](#) - `<animator>`
- [ObjectAnimator](#) - `<objectAnimator>`
- [AnimatorSet](#) - `<set>`

The following example plays the two sets of object animations sequentially, with the first nested set playing two object animations together:

```
<set android:ordering="sequentially">
    <set>
        <objectAnimator
            android:propertyName="x"
            android:duration="500"
            android:valueTo="400"
            android:valueType="intType"/>
        <objectAnimator
```

```
        android:propertyName="y"
        android:duration="500"
        android:valueTo="300"
        android:valueType="intType"/>
    
```

```
</set>
<objectAnimator
    android:propertyName="alpha"
    android:duration="500"
    android:valueTo="1f"/>

```

```
</set>
```

In order to run this animation, you must inflate the XML resources in your code to an [AnimatorSet](#) object, and then set the target objects for all of the animations before starting the animation set. Calling [setTarget\(\)](#) sets a single target object for all children of the [AnimatorSet](#) as a convenience. The following code shows how to do this:

```
AnimatorSet set = (AnimatorSet) AnimatorInflater.loadAnimator(myContext,
    R.anim.property_animator);
set.setTarget(myObject);
set.start();
```

For information about the XML syntax for defining property animations, see [Animation Resources](#).

# View Animation

You can use the view animation system to perform tweened animation on Views. Tween animation calculates the animation with information such as the start point, end point, size, rotation, and other common aspects of an animation.

A tween animation can perform a series of simple transformations (position, size, rotation, and transparency) on the contents of a View object. So, if you have a [TextView](#) object, you can move, rotate, grow, or shrink the text. If it has a background image, the background image will be transformed along with the text. The [animation package](#) provides all the classes used in a tween animation.

A sequence of animation instructions defines the tween animation, defined by either XML or Android code. As with defining a layout, an XML file is recommended because it's more readable, reusable, and swappable than hard-coding the animation. In the example below, we use XML. (To learn more about defining an animation in your application code, instead of XML, refer to the [AnimationSet](#) class and other [Animation](#) subclasses.)

The animation instructions define the transformations that you want to occur, when they will occur, and how long they should take to apply. Transformations can be sequential or simultaneous - for example, you can have the contents of a TextView move from left to right, and then rotate 180 degrees, or you can have the text move and rotate simultaneously. Each transformation takes a set of parameters specific for that transformation (starting size and ending size for size change, starting angle and ending angle for rotation, and so on), and also a set of common parameters (for instance, start time and duration). To make several transformations happen simultaneously, give them the same start time; to make them sequential, calculate the start time plus the duration of the preceding transformation.

The animation XML file belongs in the `res/anim/` directory of your Android project. The file must have a single root element: this will be either a single `<alpha>`, `<scale>`, `<translate>`, `<rotate>`, interpolator element, or `<set>` element that holds groups of these elements (which may include another `<set>`). By default, all animation instructions are applied simultaneously. To make them occur sequentially, you must specify the `startOffset` attribute, as shown in the example below.

The following XML from one of the ApiDemos is used to stretch, then simultaneously spin and rotate a View object.

```
<set android:shareInterpolator="false">
    <scale
        android:interpolator="@android:anim/accelerate_decelerate_interpolator"
        android:fromXScale="1.0"
        android:toXScale="1.4"
        android:fromYScale="1.0"
        android:toYScale="0.6"
        android:pivotX="50%"
        android:pivotY="50%"
        android:fillAfter="false"
        android:duration="700" />
    <set android:interpolator="@android:anim/decelerate_interpolator">
        <scale
            android:fromXScale="1.4"
            android:toXScale="0.0"
            android:fromYScale="0.6"
            android:toYScale="0.0"
            android:pivotX="50%"
            android:pivotY="50%"
```

```

        android:startOffset="700"
        android:duration="400"
        android:fillBefore="false" />
    <rotate
        android:fromDegrees="0"
        android:toDegrees="-45"
        android:toYScale="0.0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:startOffset="700"
        android:duration="400" />
    </set>
</set>
```

Screen coordinates (not used in this example) are (0,0) at the upper left hand corner, and increase as you go down and to the right.

Some values, such as pivotX, can be specified relative to the object itself or relative to the parent. Be sure to use the proper format for what you want ("50" for 50% relative to the parent, or "50%" for 50% relative to itself).

You can determine how a transformation is applied over time by assigning an [Interpolator](#). Android includes several Interpolator subclasses that specify various speed curves: for instance, [AccelerateInterpolator](#) tells a transformation to start slow and speed up. Each one has an attribute value that can be applied in the XML.

With this XML saved as `hyperspace_jump.xml` in the `res/anim/` directory of the project, the following code will reference it and apply it to an [ImageView](#) object from the layout.

```
ImageView spaceshipImage = (ImageView) findViewById(R.id.spaceshipImage);
Animation hyperspaceJumpAnimation = AnimationUtils.loadAnimation(this, R.anim.hyperspace_jump);
spaceshipImage.startAnimation(hyperspaceJumpAnimation);
```

As an alternative to `startAnimation()`, you can define a starting time for the animation with [Animation.setStartTime\(\)](#), then assign the animation to the View with [View.setAnimation\(\)](#).

For more information on the XML syntax, available tags and attributes, see [Animation Resources](#).

**Note:** Regardless of how your animation may move or resize, the bounds of the View that holds your animation will not automatically adjust to accommodate it. Even so, the animation will still be drawn beyond the bounds of its View and will not be clipped. However, clipping *will occur* if the animation exceeds the bounds of the parent View.

# Drawable Animation

Drawable animation lets you load a series of Drawable resources one after another to create an animation. This is a traditional animation in the sense that it is created with a sequence of different images, played in order, like a roll of film. The [AnimationDrawable](#) class is the basis for Drawable animations.

While you can define the frames of an animation in your code, using the [AnimationDrawable](#) class API, it's more simply accomplished with a single XML file that lists the frames that compose the animation. The XML file for this kind of animation belongs in the `res/drawable/` directory of your Android project. In this case, the instructions are the order and duration for each frame of the animation.

The XML file consists of an `<animation-list>` element as the root node and a series of child `<item>` nodes that each define a frame: a drawable resource for the frame and the frame duration. Here's an example XML file for a Drawable animation:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="true">
    <item android:drawable="@drawable/rocket_thrust1" android:duration="200" />
    <item android:drawable="@drawable/rocket_thrust2" android:duration="200" />
    <item android:drawable="@drawable/rocket_thrust3" android:duration="200" />
</animation-list>
```

This animation runs for just three frames. By setting the `android:oneshot` attribute of the list to *true*, it will cycle just once then stop and hold on the last frame. If it is set *false* then the animation will loop. With this XML saved as `rocket_thrust.xml` in the `res/drawable/` directory of the project, it can be added as the background image to a View and then called to play. Here's an example Activity, in which the animation is added to an [ImageView](#) and then animated when the screen is touched:

```
AnimationDrawable rocketAnimation;

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ImageView rocketImage = (ImageView) findViewById(R.id.rocket_image);
    rocketImage.setBackgroundResource(R.drawable.rocket_thrust);
    rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
}

public boolean onTouchEvent(MotionEvent event) {
    if (event.getAction() == MotionEvent.ACTION_DOWN) {
        rocketAnimation.start();
        return true;
    }
    return super.onTouchEvent(event);
}
```

It's important to note that the `start()` method called on the `AnimationDrawable` cannot be called during the `onCreate()` method of your Activity, because the `AnimationDrawable` is not yet fully attached to the window. If you want to play the animation immediately, without requiring interaction, then you might want to call it from the [onWindowFocusChanged\(\)](#) method in your Activity, which will get called when Android brings your window into focus.

For more information on the XML syntax, available tags and attributes, see [Animation Resources](#).

# Canvas and Drawables

## In this document

1. [Draw with a Canvas](#)
  1. [On a View](#)
  2. [On a SurfaceView](#)
2. [Drawables](#)
  1. [Creating from resource images](#)
  2. [Creating from resource XML](#)
3. [Shape Drawable](#)
4. [Nine-patch](#)

## See also

1. [OpenGL with the Framework APIs](#)
2. [RenderScript](#)

The Android framework APIs provides a set 2D drawing APIs that allow you to render your own custom graphics onto a canvas or to modify existing Views to customize their look and feel. When drawing 2D graphics, you'll typically do so in one of two ways:

- a. Draw your graphics or animations into a View object from your layout. In this manner, the drawing of your graphics is handled by the system's normal View hierarchy drawing process — you simply define the graphics to go inside the View.
- b. Draw your graphics directly to a Canvas. This way, you personally call the appropriate class's [on-Draw\(\)](#) method (passing it your Canvas), or one of the Canvas `draw...()` methods (like [draw-Picture\(\)](#)). In doing so, you are also in control of any animation.

Option "a," drawing to a View, is your best choice when you want to draw simple graphics that do not need to change dynamically and are not part of a performance-intensive game. For example, you should draw your graphics into a View when you want to display a static graphic or predefined animation, within an otherwise static application. Read [Drawables](#) for more information.

Option "b," drawing to a Canvas, is better when your application needs to regularly re-draw itself. Applications such as video games should be drawing to the Canvas on its own. However, there's more than one way to do this:

- In the same thread as your UI Activity, wherein you create a custom View component in your layout, call [invalidate\(\)](#) and then handle the [onDraw\(\)](#) callback.
- Or, in a separate thread, wherein you manage a [SurfaceView](#) and perform draws to the Canvas as fast as your thread is capable (you do not need to request [invalidate\(\)](#)).

## Draw with a Canvas

When you're writing an application in which you would like to perform specialized drawing and/or control the animation of graphics, you should do so by drawing through a [Canvas](#). A Canvas works for you as a pretense, or interface, to the actual surface upon which your graphics will be drawn — it holds all of your "draw" calls. Via the Canvas, your drawing is actually performed upon an underlying [Bitmap](#), which is placed into the window.

In the event that you're drawing within the `onDraw()` callback method, the Canvas is provided for you and you need only place your drawing calls upon it. You can also acquire a Canvas from `SurfaceHolder.lockCanvas()`, when dealing with a SurfaceView object. (Both of these scenarios are discussed in the following sections.) However, if you need to create a new Canvas, then you must define the `Bitmap` upon which drawing will actually be performed. The Bitmap is always required for a Canvas. You can set up a new Canvas like this:

```
Bitmap b = Bitmap.createBitmap(100, 100, Bitmap.Config.ARGB_8888);  
Canvas c = new Canvas(b);
```

Now your Canvas will draw onto the defined Bitmap. After drawing upon it with the Canvas, you can then carry your Bitmap to another Canvas with one of the `Canvas.drawBitmap(Bitmap, ...)` methods. It's recommended that you ultimately draw your final graphics through a Canvas offered to you by `View.onDraw()` or `SurfaceHolder.lockCanvas()` (see the following sections).

The `Canvas` class has its own set of drawing methods that you can use, like `drawBitmap(...)`, `drawRect(...)`, `drawText(...)`, and many more. Other classes that you might use also have `draw()` methods. For example, you'll probably have some `Drawable` objects that you want to put on the Canvas. Drawable has its own `draw()` method that takes your Canvas as an argument.

## On a View

If your application does not require a significant amount of processing or frame-rate speed (perhaps for a chess game, a snake game, or another slowly-animated application), then you should consider creating a custom View component and drawing with a Canvas in `View.onDraw()`. The most convenient aspect of doing so is that the Android framework will provide you with a pre-defined Canvas to which you will place your drawing calls.

To start, extend the `View` class (or descendant thereof) and define the `onDraw()` callback method. This method will be called by the Android framework to request that your View draw itself. This is where you will perform all your calls to draw through the `Canvas`, which is passed to you through the `onDraw()` callback.

The Android framework will only call `onDraw()` as necessary. Each time that your application is prepared to be drawn, you must request your View be invalidated by calling `invalidate()`. This indicates that you'd like your View to be drawn and Android will then call your `onDraw()` method (though is not guaranteed that the callback will be instantaneous).

Inside your View component's `onDraw()`, use the Canvas given to you for all your drawing, using various `Canvas.draw...()` methods, or other class `draw()` methods that take your Canvas as an argument. Once your `onDraw()` is complete, the Android framework will use your Canvas to draw a Bitmap handled by the system.

**Note:** In order to request an invalidate from a thread other than your main Activity's thread, you must call `postInvalidate()`.

For information about extending the `View` class, read [Building Custom Components](#).

For a sample application, see the Snake game, in the SDK samples folder: <your-sdk-directory>/samples/Snake/.

## On a SurfaceView

The `SurfaceView` is a special subclass of View that offers a dedicated drawing surface within the View hierarchy. The aim is to offer this drawing surface to an application's secondary thread, so that the application isn't

required to wait until the system's View hierarchy is ready to draw. Instead, a secondary thread that has reference to a SurfaceView can draw to its own Canvas at its own pace.

To begin, you need to create a new class that extends [SurfaceView](#). The class should also implement [SurfaceHolder.Callback](#). This subclass is an interface that will notify you with information about the underlying [Surface](#), such as when it is created, changed, or destroyed. These events are important so that you know when you can start drawing, whether you need to make adjustments based on new surface properties, and when to stop drawing and potentially kill some tasks. Inside your SurfaceView class is also a good place to define your secondary Thread class, which will perform all the drawing procedures to your Canvas.

Instead of handling the Surface object directly, you should handle it via a [SurfaceHolder](#). So, when your SurfaceView is initialized, get the SurfaceHolder by calling [getHolder\(\)](#). You should then notify the SurfaceHolder that you'd like to receive SurfaceHolder callbacks (from [SurfaceHolder.Callback](#)) by calling [addCallback\(\)](#) (pass it *this*). Then override each of the [SurfaceHolder.Callback](#) methods inside your SurfaceView class.

In order to draw to the Surface Canvas from within your second thread, you must pass the thread your SurfaceHandler and retrieve the Canvas with [lockCanvas\(\)](#). You can now take the Canvas given to you by the SurfaceHolder and do your necessary drawing upon it. Once you're done drawing with the Canvas, call [unlockCanvasAndPost\(\)](#), passing it your Canvas object. The Surface will now draw the Canvas as you left it. Perform this sequence of locking and unlocking the canvas each time you want to redraw.

**Note:** On each pass you retrieve the Canvas from the SurfaceHolder, the previous state of the Canvas will be retained. In order to properly animate your graphics, you must re-paint the entire surface. For example, you can clear the previous state of the Canvas by filling in a color with [drawColor\(\)](#) or setting a background image with [drawBitmap\(\)](#). Otherwise, you will see traces of the drawings you previously performed.

For a sample application, see the Lunar Lander game, in the SDK samples folder: <your-sdk-directory>/samples/LunarLander/. Or, browse the source in the [Sample Code](#) section.

## Drawables

Android offers a custom 2D graphics library for drawing shapes and images. The [android.graphics.drawable](#) package is where you'll find the common classes used for drawing in two-dimensions.

This document discusses the basics of using Drawable objects to draw graphics and how to use a couple subclasses of the Drawable class. For information on using Drawables to do frame-by-frame animation, see [Drawable Animation](#).

A [Drawable](#) is a general abstraction for "something that can be drawn." You'll discover that the Drawable class extends to define a variety of specific kinds of drawable graphics, including [BitmapDrawable](#), [ShapeDrawable](#), [PictureDrawable](#), [LayerDrawable](#), and several more. Of course, you can also extend these to define your own custom Drawable objects that behave in unique ways.

There are three ways to define and instantiate a Drawable: using an image saved in your project resources; using an XML file that defines the Drawable properties; or using the normal class constructors. Below, we'll discuss each the first two techniques (using constructors is nothing new for an experienced developer).

## Creating from resource images

A simple way to add graphics to your application is by referencing an image file from your project resources. Supported file types are PNG (preferred), JPG (acceptable) and GIF (discouraged). This technique would obviously be preferred for application icons, logos, or other graphics such as those used in a game.

To use an image resource, just add your file to the `res/drawable/` directory of your project. From there, you can reference it from your code or your XML layout. Either way, it is referred using a resource ID, which is the file name without the file type extension (E.g., `my_image.png` is referenced as `my_image`).

**Note:** Image resources placed in `res/drawable/` may be automatically optimized with lossless image compression by the `aapt` tool during the build process. For example, a true-color PNG that does not require more than 256 colors may be converted to an 8-bit PNG with a color palette. This will result in an image of equal quality but which requires less memory. So be aware that the image binaries placed in this directory can change during the build. If you plan on reading an image as a bit stream in order to convert it to a bitmap, put your images in the `res/raw/` folder instead, where they will not be optimized.

### Example code

The following code snippet demonstrates how to build an [ImageView](#) that uses an image from drawable resources and add it to the layout.

```
LinearLayout mLinearLayout;

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

    // Create a LinearLayout in which to add the ImageView
    mLinearLayout = new LinearLayout(this);

    // Instantiate an ImageView and define its properties
    ImageView i = new ImageView(this);
    i.setImageResource(R.drawable.my_image);
    i.setAdjustViewBounds(true); // set the ImageView bounds to match the Drawable
    i.setLayoutParams(new Gallery.LayoutParams(LayoutParams.WRAP_CONTENT,
        LayoutParams.WRAP_CONTENT));

    // Add the ImageView to the layout and set the layout as the content view
    mLinearLayout.addView(i);
    setContentView(mLinearLayout);
}
```

In other cases, you may want to handle your image resource as a [Drawable](#) object. To do so, create a Drawable from the resource like so:

```
Resources res = mContext.getResources();
Drawable myImage = res.getDrawable(R.drawable.my_image);
```

**Note:** Each unique resource in your project can maintain only one state, no matter how many different objects you may instantiate for it. For example, if you instantiate two Drawable objects from the same image resource, then change a property (such as the alpha) for one of the Drawables, then it will also affect the other. So when dealing with multiple instances of an image resource, instead of directly transforming the Drawable, you should perform a [tween animation](#).

## Example XML

The XML snippet below shows how to add a resource Drawable to an [ImageView](#) in the XML layout (with some red tint just for fun).

```
<ImageView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:tint="#55ff0000"  
    android:src="@drawable/my_image"/>
```

For more information on using project resources, read about [Resources and Assets](#).

## Creating from resource XML

By now, you should be familiar with Android's principles of developing a [User Interface](#). Hence, you understand the power and flexibility inherent in defining objects in XML. This philosophy carries over from Views to Drawables. If there is a Drawable object that you'd like to create, which is not initially dependent on variables defined by your application code or user interaction, then defining the Drawable in XML is a good option. Even if you expect your Drawable to change its properties during the user's experience with your application, you should consider defining the object in XML, as you can always modify properties once it is instantiated.

Once you've defined your Drawable in XML, save the file in the `res/drawable/` directory of your project. Then, retrieve and instantiate the object by calling [Resources.getDrawable\(\)](#), passing it the resource ID of your XML file. (See the [example below](#).)

Any Drawable subclass that supports the `inflate()` method can be defined in XML and instantiated by your application. Each Drawable that supports XML inflation utilizes specific XML attributes that help define the object properties (see the class reference to see what these are). See the class documentation for each Drawable subclass for information on how to define it in XML.

## Example

Here's some XML that defines a TransitionDrawable:

```
<transition xmlns:android="http://schemas.android.com/apk/res/android">  
    <item android:drawable="@drawable/image_expand">  
    <item android:drawable="@drawable/image_collapse">  
    </transition>
```

With this XML saved in the file `res/drawable/expand_collapse.xml`, the following code will instantiate the TransitionDrawable and set it as the content of an ImageView:

```
Resources res = mContext.getResources();  
TransitionDrawable transition = (TransitionDrawable)  
res.getDrawable(R.drawable.expand_collapse);  
ImageView image = (ImageView) findViewById(R.id.toggle_image);  
image.setImageDrawable(transition);
```

Then this transition can be run forward (for 1 second) with:

```
transition.startTransition(1000);
```

Refer to the Drawable classes listed above for more information on the XML attributes supported by each.

## Shape Drawable

When you want to dynamically draw some two-dimensional graphics, a [ShapeDrawable](#) object will probably suit your needs. With a ShapeDrawable, you can programmatically draw primitive shapes and style them in any way imaginable.

A ShapeDrawable is an extension of [Drawable](#), so you can use one where ever a Drawable is expected — perhaps for the background of a View, set with [setBackgroundDrawable\(\)](#). Of course, you can also draw your shape as its own custom [View](#), to be added to your layout however you please. Because the ShapeDrawable has its own `draw()` method, you can create a subclass of View that draws the ShapeDrawable during the `View.onDraw()` method. Here's a basic extension of the View class that does just this, to draw a ShapeDrawable as a View:

```
public class CustomDrawableView extends View {  
    private ShapeDrawable mDrawable;  
  
    public CustomDrawableView(Context context) {  
        super(context);  
  
        int x = 10;  
        int y = 10;  
        int width = 300;  
        int height = 50;  
  
        mDrawable = new ShapeDrawable(new OvalShape());  
        mDrawable.getPaint().setColor(0xff74AC23);  
        mDrawable.setBounds(x, y, x + width, y + height);  
    }  
  
    protected void onDraw(Canvas canvas) {  
        mDrawable.draw(canvas);  
    }  
}
```

In the constructor, a ShapeDrawable is defines as an [OvalShape](#). It's then given a color and the bounds of the shape are set. If you do not set the bounds, then the shape will not be drawn, whereas if you don't set the color, it will default to black.

With the custom View defined, it can be drawn any way you like. With the sample above, we can draw the shape programmatically in an Activity:

```
CustomDrawableView mCustomDrawableView;  
  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    mCustomDrawableView = new CustomDrawableView(this);  
  
    setContentView(mCustomDrawableView);
```

```
}
```

If you'd like to draw this custom drawable from the XML layout instead of from the Activity, then the CustomDrawable class must override the [View\(Context, AttributeSet\)](#) constructor, which is called when instantiating a View via inflation from XML. Then add a CustomDrawable element to the XML, like so:

```
<com.example.shapeddrawable.CustomButtonView  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
/>
```

The ShapeDrawable class (like many other Drawable types in the [android.graphics.drawable](#) package) allows you to define various properties of the drawable with public methods. Some properties you might want to adjust include alpha transparency, color filter, dither, opacity and color.

You can also define primitive drawable shapes using XML. For more information, see the section about Shape Drawables in the [Drawable Resources](#) document.

## Nine-patch

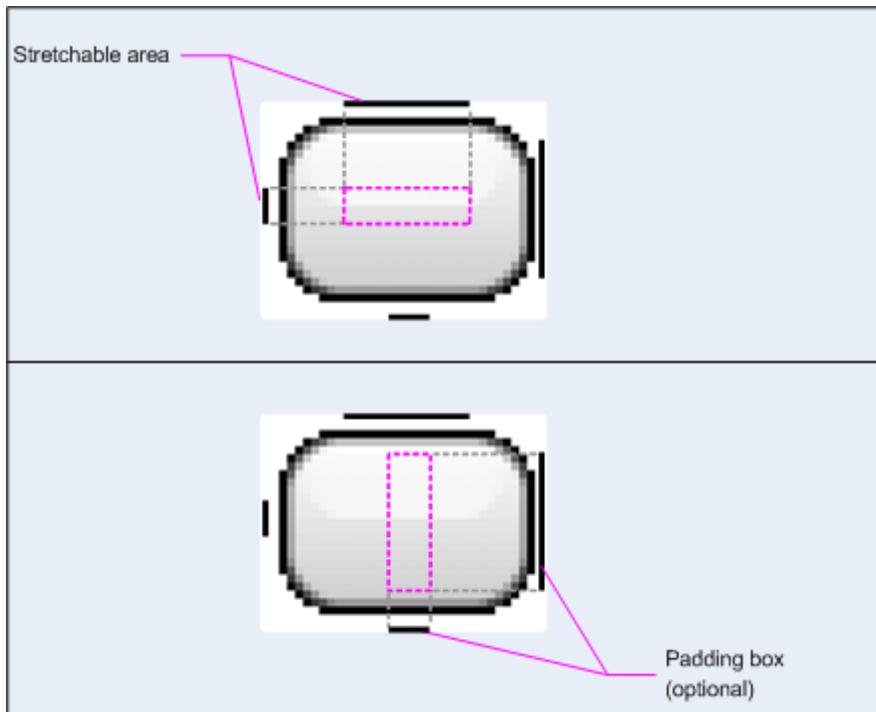
A [NinePatchDrawable](#) graphic is a stretchable bitmap image, which Android will automatically resize to accommodate the contents of the View in which you have placed it as the background. An example use of a NinePatch is the backgrounds used by standard Android buttons — buttons must stretch to accommodate strings of various lengths. A NinePatch drawable is a standard PNG image that includes an extra 1-pixel-wide border. It must be saved with the extension `.9.png`, and saved into the `res/drawable/` directory of your project.

The border is used to define the stretchable and static areas of the image. You indicate a stretchable section by drawing one (or more) 1-pixel-wide black line(s) in the left and top part of the border (the other border pixels should be fully transparent or white). You can have as many stretchable sections as you want: their relative size stays the same, so the largest sections always remain the largest.

You can also define an optional drawable section of the image (effectively, the padding lines) by drawing a line on the right and bottom lines. If a View object sets the NinePatch as its background and then specifies the View's text, it will stretch itself so that all the text fits inside only the area designated by the right and bottom lines (if included). If the padding lines are not included, Android uses the left and top lines to define this drawable area.

To clarify the difference between the different lines, the left and top lines define which pixels of the image are allowed to be replicated in order to stretch the image. The bottom and right lines define the relative area within the image that the contents of the View are allowed to lie within.

Here is a sample NinePatch file used to define a button:



This NinePatch defines one stretchable area with the left and top lines and the drawable area with the bottom and right lines. In the top image, the dotted grey lines identify the regions of the image that will be replicated in order to stretch the image. The pink rectangle in the bottom image identifies the region in which the contents of the View are allowed. If the contents don't fit in this region, then the image will be stretched so that they do.

The [Draw 9-patch](#) tool offers an extremely handy way to create your NinePatch images, using a WYSIWYG graphics editor. It even raises warnings if the region you've defined for the stretchable area is at risk of producing drawing artifacts as a result of the pixel replication.

## Example XML

Here's some sample layout XML that demonstrates how to add a NinePatch image to a couple of buttons. (The NinePatch image is saved as `res/drawable/my_button_background.9.png`

```
<Button id="@+id/tiny"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerInParent="true"
        android:text="Tiny"
        android:textSize="8sp"
        android:background="@drawable/my_button_background"/>
```

```
<Button id="@+id/big"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerInParent="true"
        android:text="Biiiiiiig text!"
        android:textSize="30sp"
        android:background="@drawable/my_button_background"/>
```

Note that the width and height are set to "wrap\_content" to make the button fit neatly around the text.

Below are the two buttons rendered from the XML and NinePatch image shown above. Notice how the width and height of the button varies with the text, and the background image stretches to accommodate it.



# OpenGL ES

## In this document

1. [The Basics](#)
  1. [OpenGL ES packages](#)
2. [Declaring OpenGL Requirements](#)
3. [Mapping Coordinates for Drawn Objects](#)
  1. [Projection and camera in ES 1.0](#)
  2. [Projection and camera in ES 2.0 and higher](#)
4. [Shape Faces and Winding](#)
5. [OpenGL Versions and Device Compatibility](#)
  1. [Texture compression support](#)
  2. [Determining OpenGL extensions](#)
  3. [Checking OpenGL ES Version](#)
6. [Choosing an OpenGL API Version](#)

## Key classes

1. [GLSurfaceView](#)
2. [GLSurfaceView.Renderer](#)

## Related samples

1. [GLSurfaceViewActivity](#)
2. [GLES20Activity](#)
3. [TouchRotateActivity](#)
4. [Compressed Textures](#)

## See also

1. [Displaying Graphics with OpenGL ES](#)
2. [OpenGL ES](#)
3. [OpenGL ES 1.x Specification](#)
4. [OpenGL ES 2.x specification](#)
5. [OpenGL ES 3.x specification](#)

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.

**Caution:** Support of the OpenGL ES 3.0 API on a device requires an implementation of this graphics pipeline provided by the device manufacturer. A device running Android 4.3 or higher *may not support* the OpenGL ES 3.0 API. For information on checking what version of OpenGL ES is supported at run time, see [Checking OpenGL ES Version](#).

**Note:** The specific API provided by the Android framework is similar to the J2ME JSR239 OpenGL ES API, but is not identical. If you are familiar with J2ME JSR239 specification, be alert for variations.

## The Basics

Android supports OpenGL both through its framework API and the Native Development Kit (NDK). This topic focuses on the Android framework interfaces. For more information about the NDK, see the [Android NDK](#).

There are two foundational classes in the Android framework that let you create and manipulate graphics with the OpenGL ES API: [GLSurfaceView](#) and [GLSurfaceView.Renderer](#). If your goal is to use OpenGL in your Android application, understanding how to implement these classes in an activity should be your first objective.

### [GLSurfaceView](#)

This class is a [View](#) where you can draw and manipulate objects using OpenGL API calls and is similar in function to a [SurfaceView](#). You can use this class by creating an instance of [GLSurfaceView](#) and adding your [Renderer](#) to it. However, if you want to capture touch screen events, you should extend the [GLSurfaceView](#) class to implement the touch listeners, as shown in OpenGL training lesson, [Responding to Touch Events](#).

### [GLSurfaceView.Renderer](#)

This interface defines the methods required for drawing graphics in a [GLSurfaceView](#). You must provide an implementation of this interface as a separate class and attach it to your [GLSurfaceView](#) instance using [GLSurfaceView.setRenderer\(\)](#).

The [GLSurfaceView.Renderer](#) interface requires that you implement the following methods:

- [onSurfaceCreated\(\)](#) : The system calls this method once, when creating the [GLSurfaceView](#). Use this method to perform actions that need to happen only once, such as setting OpenGL environment parameters or initializing OpenGL graphic objects.
- [onDrawFrame\(\)](#) : The system calls this method on each redraw of the [GLSurfaceView](#). Use this method as the primary execution point for drawing (and re-drawing) graphic objects.
- [onSurfaceChanged\(\)](#) : The system calls this method when the [GLSurfaceView](#) geometry changes, including changes in size of the [GLSurfaceView](#) or orientation of the device screen. For example, the system calls this method when the device changes from portrait to landscape orientation. Use this method to respond to changes in the [GLSurfaceView](#) container.

## OpenGL ES packages

Once you have established a container view for OpenGL ES using [GLSurfaceView](#) and [GLSurfaceView.Renderer](#), you can begin calling OpenGL APIs using the following classes:

- OpenGL ES 1.0/1.1 API Packages
  - [android.opengl](#) - This package provides a static interface to the OpenGL ES 1.0/1.1 classes and better performance than the [javax.microedition.khronos](#) package interfaces.
    - [GLES10](#)
    - [GLES10Ext](#)
    - [GLES11](#)
    - [GLES11Ext](#)
  - [javax.microedition.khronos.opengles](#) - This package provides the standard implementation of OpenGL ES 1.0/1.1.

- [GL10](#)
- [GL10Ext](#)
- [GL11](#)
- [GL11Ext](#)
- [GL11ExtensionPack](#)

- OpenGL ES 2.0 API Class
  - [android.opengl.GLES20](#) - This package provides the interface to OpenGL ES 2.0 and is available starting with Android 2.2 (API level 8).
- OpenGL ES 3.0 API Class
  - [android.opengl.GLES30](#) - This package provides the interface to OpenGL ES 3.0 and is available starting with Android 4.3 (API level 18).

If you want to start building an app with OpenGL ES right away, follow the [Displaying Graphics with OpenGL ES](#) class.

## Declaring OpenGL Requirements

If your application uses OpenGL features that are not available on all devices, you must include these requirements in your [AndroidManifest.xml](#) file. Here are the most common OpenGL manifest declarations:

- **OpenGL ES version requirements** - If your application only supports OpenGL ES 2.0, you must declare that requirement by adding the following settings to your manifest as shown below.

```
<!-- Tell the system this app requires OpenGL ES 2.0. -->
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

Adding this declaration causes Google Play to restrict your application from being installed on devices that do not support OpenGL ES 2.0. If your application is exclusively for devices that support OpenGL ES 3.0, you can also specify this in your manifest:

```
<!-- Tell the system this app requires OpenGL ES 3.0. -->
<uses-feature android:glEsVersion="0x00030000" android:required="true" />
```

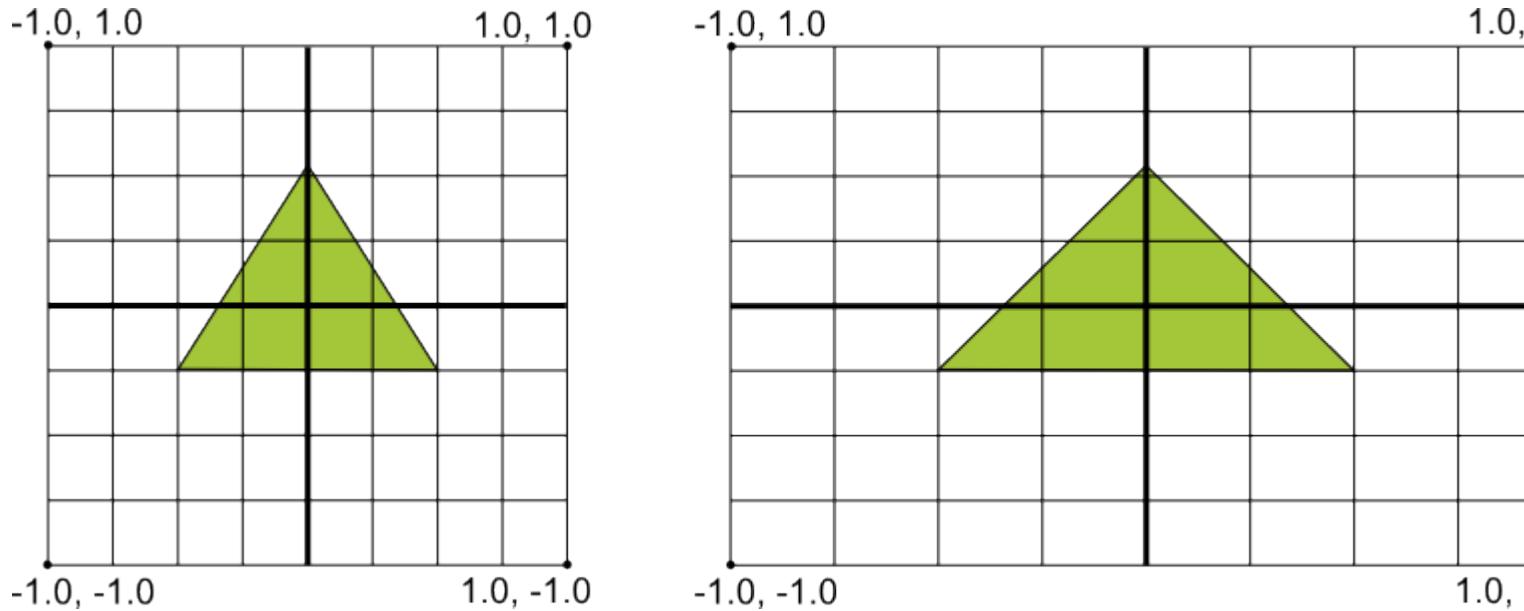
**Note:** The OpenGL ES 3.0 API is backwards-compatible with the 2.0 API, which means you can be more flexible with your implementation of OpenGL ES in your application. By declaring the OpenGL ES 2.0 API as a requirement in your manifest, you can use that API version as a default, check for the availability of the 3.0 API at run time and then use OpenGL ES 3.0 features if the device supports it. For more information about checking the OpenGL ES version supported by a device, see [Checking OpenGL ES Version](#).

- **Texture compression requirements** - If your application uses texture compression formats, you must declare the formats your application supports in your manifest file using [`<supports-gl-texture>`](#). For more information about available texture compression formats, see [Texture compression support](#).

Declaring texture compression requirements in your manifest hides your application from users with devices that do not support at least one of your declared compression types. For more information on how Google Play filtering works for texture compressions, see the [Google Play and texture compression filtering](#) section of the `<supports-gl-texture>` documentation.

# Mapping Coordinates for Drawn Objects

One of the basic problems in displaying graphics on Android devices is that their screens can vary in size and shape. OpenGL assumes a square, uniform coordinate system and, by default, happily draws those coordinates onto your typically non-square screen as if it is perfectly square.



**Figure 1.** Default OpenGL coordinate system (left) mapped to a typical Android device screen (right).

The illustration above shows the uniform coordinate system assumed for an OpenGL frame on the left, and how these coordinates actually map to a typical device screen in landscape orientation on the right. To solve this problem, you can apply OpenGL projection modes and camera views to transform coordinates so your graphic objects have the correct proportions on any display.

In order to apply projection and camera views, you create a projection matrix and a camera view matrix and apply them to the OpenGL rendering pipeline. The projection matrix recalculates the coordinates of your graphics so that they map correctly to Android device screens. The camera view matrix creates a transformation that renders objects from a specific eye position.

## Projection and camera view in OpenGL ES 1.0

In the ES 1.0 API, you apply projection and camera view by creating each matrix and then adding them to the OpenGL environment.

1. **Projection matrix** - Create a projection matrix using the geometry of the device screen in order to recalculate object coordinates so they are drawn with correct proportions. The following example code demonstrates how to modify the [onSurfaceChanged\(\)](#) method of a [GLSurfaceView.Renderer](#) implementation to create a projection matrix based on the screen's aspect ratio and apply it to the OpenGL rendering environment.

```
public void onSurfaceChanged(GL10 gl, int width, int height) {  
    gl.glViewport(0, 0, width, height);  
  
    // make adjustments for screen ratio  
    float ratio = (float) width / height;  
    gl.glMatrixMode(GL10.GL_PROJECTION);           // set matrix to projecti  
    gl.glLoadIdentity();                          // reset the matrix to it
```

```
    gl.glFrustumf(-ratio, ratio, -1, 1, 3, 7); // apply the projection matrix
}
```

2. **Camera transformation matrix** - Once you have adjusted the coordinate system using a projection matrix, you must also apply a camera view. The following example code shows how to modify the [onDrawFrame\(\)](#) method of a [GLSurfaceView.Renderer](#) implementation to apply a model view and use the [GLU.gluLookAt\(\)](#) utility to create a viewing transformation which simulates a camera position.

```
public void onDrawFrame(GL10 gl) {
    ...
    // Set GL_MODELVIEW transformation mode
    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glLoadIdentity(); // reset the matrix to its identity
    ...
    // When using GL_MODELVIEW, you must set the camera view
    GLU.gluLookAt(gl, 0, 0, -5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);
    ...
}
```

## Projection and camera view in OpenGL ES 2.0 and higher

In the ES 2.0 and 3.0 APIs, you apply projection and camera view by first adding a matrix member to the vertex shaders of your graphics objects. With this matrix member added, you can then generate and apply projection and camera viewing matrices to your objects.

1. **Add matrix to vertex shaders** - Create a variable for the view projection matrix and include it as a multiplier of the shader's position. In the following example vertex shader code, the included `uMVPMatrix` member allows you to apply projection and camera viewing matrices to the coordinates of objects that use this shader.

```
private final String vertexShaderCode =
    // This matrix member variable provides a hook to manipulate
    // the coordinates of objects that use this vertex shader.
    "uniform mat4 uMVPMatrix; \n" +
    "attribute vec4 vPosition; \n" +
    "void main(){ \n" +
    // The matrix must be included as part of gl_Position
    // Note that the uMVPMatrix factor *must be first* in order
    // for the matrix multiplication product to be correct.
    " gl_Position = uMVPMatrix * vPosition; \n" +
    "}" \n";
```

**Note:** The example above defines a single transformation matrix member in the vertex shader into which you apply a combined projection matrix and camera view matrix. Depending on your application requirements, you may want to define separate projection matrix and camera viewing matrix members in your vertex shaders so you can change them independently.

2. **Access the shader matrix** - After creating a hook in your vertex shaders to apply projection and camera view, you can then access that variable to apply projection and camera viewing matrices. The fol-

lowing code shows how to modify the `onSurfaceCreated()` method of a `GLSurfaceView.Renderer` implementation to access the matrix variable defined in the vertex shader above.

```
public void onSurfaceCreated(GL10 unused, EGLConfig config) {  
    ...  
    muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");  
    ...  
}
```

3. **Create projection and camera viewing matrices** - Generate the projection and viewing matrices to be applied the graphic objects. The following example code shows how to modify the `onSurfaceCreated()` and `onSurfaceChanged()` methods of a `GLSurfaceView.Renderer` implementation to create camera view matrix and a projection matrix based on the screen aspect ratio of the device.

```
public void onSurfaceCreated(GL10 unused, EGLConfig config) {  
    ...  
    // Create a camera view matrix  
    Matrix.setLookAtM(mVMatrix, 0, 0, 0, -3, 0f, 0f, 0f, 0f, 1.0f, 0.0f);  
}  
  
public void onSurfaceChanged(GL10 unused, int width, int height) {  
    GLES20.glViewport(0, 0, width, height);  
  
    float ratio = (float) width / height;  
  
    // create a projection matrix from device screen geometry  
    Matrix.frustumM(mProjMatrix, 0, -ratio, ratio, -1, 1, 3, 7);  
}
```

4. **Apply projection and camera viewing matrices** - To apply the projection and camera view transformations, multiply the matrices together and then set them into the vertex shader. The following example code shows how modify the `onDrawFrame()` method of a `GLSurfaceView.Renderer` implementation to combine the projection matrix and camera view created in the code above and then apply it to the graphic objects to be rendered by OpenGL.

```
public void onDrawFrame(GL10 unused) {  
    ...  
    // Combine the projection and camera view matrices  
    Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mVMatrix, 0);  
  
    // Apply the combined projection and camera view transformations  
    GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, mMVPMatrix, 0);  
  
    // Draw objects  
    ...  
}
```

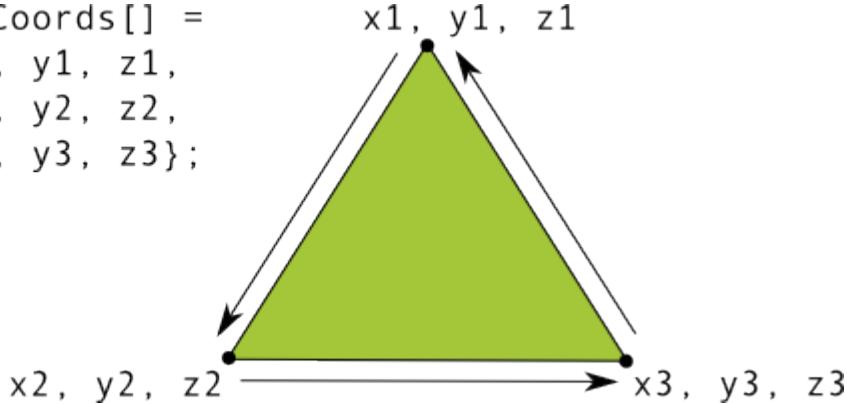
For a complete example of how to apply projection and camera view with OpenGL ES 2.0, see the [Displaying Graphics with OpenGL ES](#) class.

## Shape Faces and Winding

In OpenGL, the face of a shape is a surface defined by three or more points in three-dimensional space. A set of three or more three-dimensional points (called vertices in OpenGL) have a front face and a back face. How do

you know which face is front and which is the back? Good question. The answer has to do with winding, or, the direction in which you define the points of a shape.

```
float triangleCoords[] =  
    {x1, y1, z1,  
     x2, y2, z2,  
     x3, y3, z3};
```



**Figure 1.** Illustration of a coordinate list which translates into a counterclockwise drawing order.

In this example, the points of the triangle are defined in an order such that they are drawn in a counterclockwise direction. The order in which these coordinates are drawn defines the winding direction for the shape. By default, in OpenGL, the face which is drawn counterclockwise is the front face. The triangle shown in Figure 1 is defined so that you are looking at the front face of the shape (as interpreted by OpenGL) and the other side is the back face.

Why is it important to know which face of a shape is the front face? The answer has to do with a commonly used feature of OpenGL, called face culling. Face culling is an option for the OpenGL environment which allows the rendering pipeline to ignore (not calculate or draw) the back face of a shape, saving time, memory and processing cycles:

```
// enable face culling feature  
gl.glEnable(GL10.GL_CULL_FACE);  
// specify which faces to not draw  
gl.glCullFace(GL10.GL_BACK);
```

If you try to use the face culling feature without knowing which sides of your shapes are the front and back, your OpenGL graphics are going to look a bit thin, or possibly not show up at all. So, always define the coordinates of your OpenGL shapes in a counterclockwise drawing order.

**Note:** It is possible to set an OpenGL environment to treat the clockwise face as the front face, but doing so requires more code and is likely to confuse experienced OpenGL developers when you ask them for help. So don't do that.

## OpenGL Versions and Device Compatibility

The OpenGL ES 1.0 and 1.1 API specifications have been supported since Android 1.0. Beginning with Android 2.2 (API level 8), the framework supports the OpenGL ES 2.0 API specification. OpenGL ES 2.0 is supported by most Android devices and is recommended for new applications being developed with OpenGL. OpenGL ES 3.0 is supported with Android 4.3 (API level 18) and higher, on devices that provide an implementation of the OpenGL ES 3.0 API. For information about the relative number of Android-powered devices that support a given version of OpenGL ES, see the [OpenGL ES Version Dashboard](#).

Graphics programming with OpenGL ES 1.0/1.1 API is significantly different than using the 2.0 and higher versions. The 1.x version of the API has more convenience methods and a fixed graphics pipeline, while the OpenGL ES 2.0 and 3.0 APIs provide more direct control of the pipeline through use of OpenGL shaders. You

should carefully consider the graphics requirements and choose the API version that works best for your application. For more information, see [Choosing an OpenGL API Version](#).

The OpenGL ES 3.0 API provides additional features and better performance than the 2.0 API and is also backward compatible. This means that you can potentially write your application targeting OpenGL ES 2.0 and conditionally include OpenGL ES 3.0 graphics features if they are available. For more information on checking for availability of the 3.0 API, see [Checking OpenGL ES Version](#).

## Texture compression support

Texture compression can significantly increase the performance of your OpenGL application by reducing memory requirements and making more efficient use of memory bandwidth. The Android framework provides support for the ETC1 compression format as a standard feature, including a [ETC1Util](#) utility class and the `etc1tool` compression tool (located in the Android SDK at `<sdk>/tools/`). For an example of an Android application that uses texture compression, see the `CompressedTextureActivity` code sample in Android SDK (`<sdk>/samples/<version>/ApiDemos/src/com/example/android/apis/graphics/`).

**Caution:** The ETC1 format is supported by most Android devices, but it is not guaranteed to be available. To check if the ETC1 format is supported on a device, call the [ETC1Util.isETC1Supported\(\)](#) method.

**Note:** The ETC1 texture compression format does not support textures with transparency (alpha channel). If your application requires textures with transparency, you should investigate other texture compression formats available on your target devices.

The ETC2/EAC texture compression formats are guaranteed to be available when using the OpenGL ES 3.0 API. This texture format offers excellent compression ratios with high visual quality and the format also supports transparency (alpha channel).

Beyond the ETC formats, Android devices have varied support for texture compression based on their GPU chipsets and OpenGL implementations. You should investigate texture compression support on the devices you are targeting to determine what compression types your application should support. In order to determine what texture formats are supported on a given device, you must [query the device](#) and review the *OpenGL extension names*, which identify what texture compression formats (and other OpenGL features) are supported by the device. Some commonly supported texture compression formats are as follows:

- **ATITC (ATC)** - ATI texture compression (ATITC or ATC) is available on a wide variety of devices and supports fixed rate compression for RGB textures with and without an alpha channel. This format may be represented by several OpenGL extension names, for example:
  - `GL_AMD_compressed_ATC_texture`
  - `GL_ATI_texture_compression_atitc`
- **PVRTC** - PowerVR texture compression (PVRTC) is available on a wide variety of devices and supports 2-bit and 4-bit per pixel textures with or without an alpha channel. This format is represented by the following OpenGL extension name:
  - `GL_IMG_texture_compression_pvrtc`
- **S3TC (DXTn/DXTc)** - S3 texture compression (S3TC) has several format variations (DXT1 to DX-T5) and is less widely available. The format supports RGB textures with 4-bit alpha or 8-bit alpha channels. This format may be represented by several OpenGL extension names, for example:
  - `GL_OES_texture_compression_S3TC`
  - `GL_EXT_texture_compression_s3tc`
  - `GL_EXT_texture_compression_dxt1`
  - `GL_EXT_texture_compression_dxt3`
  - `GL_EXT_texture_compression_dxt5`

- **3DC** - 3DC texture compression (3DC) is a less widely available format that supports RGB textures with an alpha channel. This format is represented by the following OpenGL extension name:
  - `GL_AMD_compressed_3DC_texture`

**Warning:** These texture compression formats are *not supported* on all devices. Support for these formats can vary by manufacturer and device. For information on how to determine what texture compression formats are on a particular device, see the next section.

**Note:** Once you decide which texture compression formats your application will support, make sure you declare them in your manifest using [`<supports-gl-texture>`](#). Using this declaration enables filtering by external services such as Google Play, so that your app is installed only on devices that support the formats your app requires. For details, see [OpenGL manifest declarations](#).

## Determining OpenGL extensions

Implementations of OpenGL vary by Android device in terms of the extensions to the OpenGL ES API that are supported. These extensions include texture compressions, but typically also include other extensions to the OpenGL feature set.

To determine what texture compression formats, and other OpenGL extensions, are supported on a particular device:

1. Run the following code on your target devices to determine what texture compression formats are supported:

```
String extensions = javax.microedition.khronos.opengles.GL10.glGetString(GL10.GL_EXTENSIONS);
```

**Warning:** The results of this call *vary by device model!* You must run this call on several target devices to determine what compression types are commonly supported.

2. Review the output of this method to determine what OpenGL extensions are supported on the device.

## Checking OpenGL ES Version

There are several versions of the OpenGL ES available on Android devices. You can specify the minimum version of the API your application requires in your [manifest](#), but you may also want to take advantage of features in a newer API at the same time. For example, the OpenGL ES 3.0 API is backward-compatible with the 2.0 version of the API, so you may want to write your application so that it uses OpenGL ES 3.0 features, but falls back to the 2.0 API if the 3.0 API is not available.

Before using OpenGL ES features from a version higher than the minimum required in your application manifest, your application should check the version of the API available on the device. You can do this in one of two ways:

1. Attempt to create the higher-level OpenGL ES context ([EGLContext](#)) and check the result.
2. Create a minimum-supported OpenGL ES context and check the version value.

The following example code demonstrates how to check the available OpenGL ES version by creating an [EGLContext](#) and checking the result. This example shows how to check for OpenGL ES 3.0 version:

```
private static double glVersion = 3.0;  
  
private static class ContextFactory implements GLSurfaceView.EGLContextFactory
```

```

private static int EGL_CONTEXT_CLIENT_VERSION = 0x3098;

public EGLContext createContext(
    EGL10 egl, EGLDisplay display, EGLConfig eglConfig) {

    Log.w(TAG, "creating OpenGL ES " + glVersion + " context");
    int[] attrib_list = {EGL_CONTEXT_CLIENT_VERSION, (int) glVersion,
        EGL10.EGL_NONE };
    // attempt to create a OpenGL ES 3.0 context
    EGLContext context = egl.eglCreateContext(
        display, eglConfig, EGL10.EGL_NO_CONTEXT, attrib_list);
    return context; // returns null if 3.0 is not supported;
}
}

```

If the `createContext()` method shown above returns null, your code should create a OpenGL ES 2.0 context instead and fall back to using only that API.

The following code example demonstrates how to check the OpenGL ES version by creating a minimum supported context first, and then checking the version string:

```

// Create a minimum supported OpenGL ES context, then check:
String version = javax.microedition.khronos.opengles.GL10.glGetString(
    GL10.GL_VERSION);
Log.w(TAG, "Version: " + version );
// The version format is displayed as: "OpenGL ES <major>.<minor>"
// followed by optional content provided by the implementation.

```

With this approach, if you discover that the device supports a higher-level API version, you must destroy the minimum OpenGL ES context and create a new context with the higher available API version.

## Choosing an OpenGL API Version

OpenGL ES 1.0 API version (and the 1.1 extensions), version 2.0, and version 3.0 all provide high performance graphics interfaces for creating 3D games, visualizations and user interfaces. Graphics programming for OpenGL ES 2.0 and 3.0 is largely similar, with version 3.0 representing a superset of the 2.0 API with additional features. Programming for the OpenGL ES 1.0/1.1 API versus OpenGL ES 2.0 and 3.0 differs significantly, and so developers should carefully consider the following factors before starting development with these APIs:

- **Performance** - In general, OpenGL ES 2.0 and 3.0 provide faster graphics performance than the ES 1.0/1.1 APIs. However, the performance difference can vary depending on the Android device your OpenGL application is running on, due to differences in hardware manufacturer's implementation of the OpenGL ES graphics pipeline.
- **Device Compatibility** - Developers should consider the types of devices, Android versions and the OpenGL ES versions available to their customers. For more information on OpenGL compatibility across devices, see the [OpenGL Versions and Device Compatibility](#) section.
- **Coding Convenience** - The OpenGL ES 1.0/1.1 API provides a fixed function pipeline and convenience functions which are not available in the OpenGL ES 2.0 or 3.0 APIs. Developers who are new to OpenGL ES may find coding for version 1.0/1.1 faster and more convenient.
- **Graphics Control** - The OpenGL ES 2.0 and 3.0 APIs provide a higher degree of control by providing a fully programmable pipeline through the use of shaders. With more direct control of the graphics processing pipeline, developers can create effects that would be very difficult to generate using the 1.0/1.1 API.

- **Texture Support** - The OpenGL ES 3.0 API has the best support for texture compression because it guarantees availability of the ETC2 compression format, which supports transparency. The 1.x and 2.0 API implementations usually include support for ETC1, however this texture format does not support transparency and so you must typically provide resources in other compression formats supported by the devices you are targeting. For more information, see [Texture compression support](#).

While performance, compatibility, convenience, control and other factors may influence your decision, you should pick an OpenGL API version based on what you think provides the best experience for your users.

# Hardware Acceleration

## In this document

1. [Controlling Hardware Acceleration](#)
2. [Determining if a View is Hardware Accelerated](#)
3. [Android Drawing Models](#)
  1. [Software-based drawing model](#)
  2. [Hardware accelerated drawing model](#)
4. [Unsupported Drawing Operations](#)
5. [View Layers](#)
  1. [View Layers and Animations](#)
6. [Tips and Tricks](#)

## See also

1. [OpenGL with the Framework APIs](#)
2. [Renderscript](#)

Beginning in Android 3.0 (API level 11), the Android 2D rendering pipeline supports hardware acceleration, meaning that all drawing operations that are performed on a [View](#)'s canvas use the GPU. Because of the increased resources required to enable hardware acceleration, your app will consume more RAM.

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. See [Controlling Hardware Acceleration](#).

If your application performs custom drawing, test your application on actual hardware devices with hardware acceleration turned on to find any problems. The [Unsupported drawing operations](#) section describes known issues with hardware acceleration and how to work around them.

## 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:

```
<application android: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:

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

## Window level

If you need even more fine-grained control, you can enable hardware acceleration for a given window with the following code:

```
getWindow().setFlags(
    WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED,
    WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED);
```

**Note:** You currently cannot disable hardware acceleration at the window level.

## View level

You can disable hardware acceleration for an individual view at runtime with the following code:

```
myView.setLayerType(View.LAYER_TYPE_SOFTWARE, null);
```

**Note:** You currently cannot enable hardware acceleration at the view level. View layers have other functions besides disabling hardware acceleration. See [View layers](#) for more information about their uses.

# 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.

# Android Drawing Models

When hardware acceleration is enabled, the Android framework utilizes a new drawing model that utilizes *display lists* to render your application to the screen. To fully understand display lists and how they might affect your application, it is useful to understand how Android draws views without hardware acceleration as well. The following sections describe the software-based and hardware-accelerated drawing models.

## Software-based drawing model

In the software drawing model, views are drawn with the following two steps:

1. Invalidate the hierarchy
2. Draw the hierarchy

Whenever an application needs to update a part of its UI, it invokes [invalidate\(\)](#) (or one of its variants) on any view that has changed content. The invalidation messages are propagated all the way up the view hierarchy to compute the regions of the screen that need to be redrawn (the dirty region). The Android system then draws any view in the hierarchy that intersects with the dirty region. Unfortunately, there are two drawbacks to this drawing model:

- First, this model requires execution of a lot of code on every draw pass. For example, if your application calls [invalidate\(\)](#) on a button and that button sits on top of another view, the Android system redraws the view even though it hasn't changed.
- The second issue is that the drawing model can hide bugs in your application. Since the Android system redraws views when they intersect the dirty region, a view whose content you changed might be re-drawn even though [invalidate\(\)](#) was not called on it. When this happens, you are relying on another view being invalidated to obtain the proper behavior. This behavior can change every time you modify your application. Because of this, you should always call [invalidate\(\)](#) on your custom views whenever you modify data or state that affects the view's drawing code.

**Note:** Android views automatically call [invalidate\(\)](#) when their properties change, such as the background color or the text in a [TextView](#).

## Hardware accelerated drawing model

The Android system still uses [invalidate\(\)](#) and [draw\(\)](#) to request screen updates and to render views, but handles the actual drawing differently. Instead of executing the drawing commands immediately, the Android system records them inside display lists, which contain the output of the view hierarchy's drawing code. Another optimization is that the Android system only needs to record and update display lists for views marked dirty by an [invalidate\(\)](#) call. Views that have not been invalidated can be redrawn simply by re-issuing the previously recorded display list. The new drawing model contains three stages:

1. Invalidate the hierarchy
2. Record and update display lists
3. Draw the display lists

With this model, you cannot rely on a view intersecting the dirty region to have its [draw\(\)](#) method executed. To ensure that the Android system records a view's display list, you must call [invalidate\(\)](#). Forgetting to do so causes a view to look the same even after it has been changed.

Using display lists also benefits animation performance because setting specific properties, such as alpha or rotation, does not require invalidating the targeted view (it is done automatically). This optimization also applies to views with display lists (any view when your application is hardware accelerated.) For example, assume

there is a [LinearLayout](#) that contains a [ListView](#) above a [Button](#). The display list for the [LinearLayout](#) looks like this:

- DrawDisplayList(ListView)
- DrawDisplayList(Button)

Assume now that you want to change the [ListView](#)'s opacity. After invoking `setAlpha(0.5f)` on the [ListView](#), the display list now contains this:

- SaveLayerAlpha(0.5)
- DrawDisplayList(ListView)
- Restore
- DrawDisplayList(Button)

The complex drawing code of [ListView](#) was not executed. Instead, the system only updated the display list of the much simpler [LinearLayout](#). In an application without hardware acceleration enabled, the drawing code of both the list and its parent are executed again.

## Unsupported Drawing Operations

When hardware accelerated, the 2D rendering pipeline supports the most commonly used [Canvas](#) drawing operations as well as many less-used operations. All of the drawing operations that are used to render applications that ship with Android, default widgets and layouts, and common advanced visual effects such as reflections and tiled textures are supported.

The following table describes the support level of various operations across API levels:

	API level		
	< 16	16	17+8
Canvas			
drawBitmapMesh() (colors array)	XX	X ✓	
drawPicture()	XX	X X	
drawPosText()	X✓	✓ ✓	
drawTextOnPath()	X✓	✓ ✓	
drawVertices()	XX	X X	
setDrawFilter()	X✓	✓ ✓	
clipPath()	XX	X ✓	
clipRegion()	XX	X ✓	
clipRect(Region.Op.XOR)	XX	X ✓	
clipRect(Region.Op.Difference)	XX	X ✓	
clipRect(Region.Op.ReverseDifference)	XX	X ✓	
clipRect() with rotation/perspective	XX	X ✓	
Paint			
setAntiAlias() (for text)	XX	X ✓	
setAntiAlias() (for lines)	X✓	✓ ✓	
setFilterBitmap()	XX	✓ ✓	
setLinearText()	XX	X X	
setMaskFilter()	XX	X X	
setPathEffect() (for lines)	XX	X X	
setRasterizer()	XX	X X	
setShadowLayer() (other than text)	XX	X X	
setStrokeCap() (for lines)	XX	X ✓	

setStrokeCap() (for points)	<b>XX</b>	X X
setSubpixelText()	<b>XX</b>	X X
Xfermode		
AvoidXfermode	<b>XX</b>	X X
PixelXorXfermode	<b>XX</b>	X X
PorterDuff.Mode.DARKEN (framebuffer)	<b>XX</b>	X X
PorterDuff.Mode.LIGHTEN (framebuffer)	<b>XX</b>	X X
PorterDuff.Mode.OVERLAY (framebuffer)	<b>XX</b>	X X
Shader		
ComposeShader inside ComposeShader	<b>XX</b>	X X
Same type shaders inside ComposeShader	<b>XX</b>	X X
Local matrix on ComposeShader	<b>XX</b>	X ✓

## Canvas Scaling

The hardware accelerated 2D rendering pipeline was built first to support unscaled drawing, with some drawing operations degrading quality significantly at higher scale values. These operations are implemented as textures drawn at scale 1.0, transformed by the GPU. In API level <17, using these operations will result in scaling artifacts increasing with scale.

The following table shows when implementation was changed to correctly handle large scales:

API level  
 < 17 17 18

Support for large scale factors

drawText()	<b>XX</b>	✓
drawPosText()	<b>XX</b>	X
drawTextOnPath()	<b>XX</b>	X
Simple Shapes*	<b>XX</b>	✓
Complex Shapes*	<b>XX</b>	X
drawPath()	<b>XX</b>	X
Shadow layer	<b>XX</b>	X

**Note:** 'Simple' shapes are `drawRect()`, `drawCircle()`, `drawOval()`, `drawRoundRect()`, and `drawArc()` (with `useCenter=false`) commands issued with a Paint that doesn't have a PathEffect, and doesn't contain non-default joins (via `setStrokeJoin()` / `setStrokeMiter()`). Other instances of those draw commands fall under 'Complex,' in the above chart.

If your application is affected by any of these missing features or limitations, you can turn off hardware acceleration for just the affected portion of your application by calling [setLayerType \(View.LAYER\\_TYPE\\_SOFTWARE, null\)](#). This way, you can still take advantage of hardware acceleration everywhere else. See [Controlling Hardware Acceleration](#) for more information on how to enable and disable hardware acceleration at different levels in your application.

## View Layers

In all versions of Android, views have had the ability to render into off-screen buffers, either by using a view's drawing cache, or by using [Canvas.saveLayer\(\)](#). Off-screen buffers, or layers, have several uses. You can use them to get better performance when animating complex views or to apply composition effects. For instance, you can implement fade effects using `Canvas.saveLayer()` to temporarily render a view into a layer and then composite it back on screen with an opacity factor.

Beginning in Android 3.0 (API level 11), you have more control on how and when to use layers with the [View.setLayerType\(\)](#) method. This API takes two parameters: the type of layer you want to use and an optional [Paint](#) object that describes how the layer should be composited. You can use the [Paint](#) parameter to apply color filters, special blending modes, or opacity to a layer. A view can use one of three layer types:

- [LAYER\\_TYPE\\_NONE](#): The view is rendered normally and is not backed by an off-screen buffer. This is the default behavior.
- [LAYER\\_TYPE\\_HARDWARE](#): The view is rendered in hardware into a hardware texture if the application is hardware accelerated. If the application is not hardware accelerated, this layer type behaves the same as [LAYER\\_TYPE\\_SOFTWARE](#).
- [LAYER\\_TYPE\\_SOFTWARE](#): The view is rendered in software into a bitmap.

The type of layer you use depends on your goal:

- **Performance**: Use a hardware layer type to render a view into a hardware texture. Once a view is rendered into a layer, its drawing code does not have to be executed until the view calls [invalidate\(\)](#). Some animations, such as alpha animations, can then be applied directly onto the layer, which is very efficient for the GPU to do.
- **Visual effects**: Use a hardware or software layer type and a [Paint](#) to apply special visual treatments to a view. For instance, you can draw a view in black and white using a [ColorMatrixColorFilter](#).
- **Compatibility**: Use a software layer type to force a view to be rendered in software. If a view that is hardware accelerated (for instance, if your whole application is hardware accelerated), is having rendering problems, this is an easy way to work around limitations of the hardware rendering pipeline.

## View layers and animations

Hardware layers can deliver faster and smoother animations when your application is hardware accelerated. Running an animation at 60 frames per second is not always possible when animating complex views that issue a lot of drawing operations. This can be alleviated by using hardware layers to render the view to a hardware texture. The hardware texture can then be used to animate the view, eliminating the need for the view to constantly redraw itself when it is being animated. The view is not redrawn unless you change the view's properties, which calls [invalidate\(\)](#), or if you call [invalidate\(\)](#) manually. If you are running an animation in your application and do not obtain the smooth results you want, consider enabling hardware layers on your animated views.

When a view is backed by a hardware layer, some of its properties are handled by the way the layer is composited on screen. Setting these properties will be efficient because they do not require the view to be invalidated and redrawn. The following list of properties affect the way the layer is composited. Calling the setter for any of these properties results in optimal invalidation and no redrawing of the targeted view:

- `alpha`: Changes the layer's opacity
- `x, y, translationX, translationY`: Changes the layer's position
- `scaleX, scaleY`: Changes the layer's size
- `rotation, rotationX, rotationY`: Changes the layer's orientation in 3D space
- `pivotX, pivotY`: Changes the layer's transformations origin

These properties are the names used when animating a view with an [ObjectAnimator](#). If you want to access these properties, call the appropriate setter or getter. For instance, to modify the `alpha` property, call [setAlpha\(\)](#). The following code snippet shows the most efficient way to rotate a view in 3D around the Y-axis:

```
view.setLayerType(View.LAYER_TYPE_HARDWARE, null);
ObjectAnimator.ofFloat(view, "rotationY", 180).start();
```

Because hardware layers consume video memory, it is highly recommended that you enable them only for the duration of the animation and then disable them after the animation is done. You can accomplish this using animation listeners:

```
View.setLayerType(View.LAYER_TYPE_HARDWARE, null);
ObjectAnimator animator = ObjectAnimator.ofFloat(view, "rotationY", 180);
animator.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        view.setLayerType(View.LAYER_TYPE_NONE, null);
    }
});
animator.start();
```

For more information on property animation, see [Property Animation](#).

## Tips and Tricks

Switching to hardware accelerated 2D graphics can instantly increase performance, but you should still design your application to use the GPU effectively by following these recommendations:

### Reduce the number of views in your application

The more views the system has to draw, the slower it will be. This applies to the software rendering pipeline as well. Reducing views is one of the easiest ways to optimize your UI.

### Avoid overdraw

Do not draw too many layers on top of each other. Remove any views that are completely obscured by other opaque views on top of it. If you need to draw several layers blended on top of each other, consider merging them into a single layer. A good rule of thumb with current hardware is to not draw more than 2.5 times the number of pixels on screen per frame (transparent pixels in a bitmap count!).

### Don't create render objects in draw methods

A common mistake is to create a new [Paint](#) or a new [Path](#) every time a rendering method is invoked. This forces the garbage collector to run more often and also bypasses caches and optimizations in the hardware pipeline.

### Don't modify shapes too often

Complex shapes, paths, and circles for instance, are rendered using texture masks. Every time you create or modify a path, the hardware pipeline creates a new mask, which can be expensive.

### Don't modify bitmaps too often

Every time you change the content of a bitmap, it is uploaded again as a GPU texture the next time you draw it.

### Use alpha with care

When you make a view translucent using [setAlpha\(\)](#), [AlphaAnimation](#), or [ObjectAnimator](#), it is rendered in an off-screen buffer which doubles the required fill-rate. When applying alpha on very large views, consider setting the view's layer type to [LAYER\\_TYPE\\_HARDWARE](#).

# Computation

RenderScript provides a platform-independent computation engine that operates at the native level. Use it to accelerate your apps that require extensive computational horsepower.

## Blog Articles

### [Evolution of RenderScript Performance](#)

It's been a year since the last blog post on RenderScript, and with the release of Android 4.2, it's a good time to talk about the performance work that we've done since then. One of the major goals of this past year was to improve the performance of common image-processing operations with RenderScript.

### [Levels in RenderScript](#)

For ICS, RenderScript (RS) has been updated with several new features to simplify adding compute acceleration to your application. RS is interesting for compute acceleration when you have large buffers of data on which you need to do significant processing. In this example we will look at applying a levels/saturation operation on a bitmap.

### [RenderScript Part 2](#)

In Introducing RenderScript I gave a brief overview of this technology. In this post I'll look at "compute" in more detail. In RenderScript we use "compute" to mean offloading of data processing from Dalvik code to RenderScript code which may run on the same or different processor(s).

# RenderScript

## In this document

1. [Writing a RenderScript Kernel](#)
2. [Using RenderScript from Java Code](#)

## Related Samples

1. [Hello Compute](#)

RenderScript is a framework for running computationally intensive tasks at high performance on Android. RenderScript is primarily oriented for use with data-parallel computation, although serial computationally intensive workloads can benefit as well. The RenderScript runtime will parallelize work across all processors available on a device, such as multi-core CPUs, GPUs, or DSPs, allowing you to focus on expressing algorithms rather than scheduling work or load balancing. RenderScript is especially useful for applications performing image processing, computational photography, or computer vision.

To begin with RenderScript, there are two main concepts you should understand:

- High-performance compute kernels are written in a C99-derived language.
- A Java API is used for managing the lifetime of RenderScript resources and controlling kernel execution.

## Writing a RenderScript Kernel

A RenderScript kernel typically resides in a `.rs` file in the `<project_root>/src/` directory; each `.rs` file is called a script. Every script contains its own set of kernels, functions, and variables. A script can contain:

- A pragma declaration (`#pragma version(1)`) that declares the version of the RenderScript kernel language used in this script. Currently, 1 is the only valid value.
- A pragma declaration (`#pragma rs java_package_name(com.example.app)`) that declares the package name of the Java classes reflected from this script.
- Some number of invokable functions. An invokable function is a single-threaded RenderScript function that you can call from your Java code with arbitrary arguments. These are often useful for initial setup or serial computations within a larger processing pipeline.
- Some number of script globals. A script global is equivalent to a global variable in C. You can access script globals from Java code, and these are often used for parameter passing to RenderScript kernels.
- Some number of compute kernels. A kernel is a parallel function that executes across every [Element](#) within an [Allocation](#).

A simple kernel may look like the following:

```
uchar4 __attribute__((kernel)) invert(uchar4 in, uint32_t x, uint32_t y)
{
    uchar4 out = in;
    out.r = 255 - in.r;
    out.g = 255 - in.g;
    out.b = 255 - in.b;
    return out;
}
```

In most respects, this is identical to a standard C function. The first notable feature is the `__attribute__((kernel))` applied to the function prototype. This denotes that the function is a RenderScript kernel instead of an invokable function. The next feature is the `in` argument and its type. In a RenderScript kernel, this is a special argument that is automatically filled in based on the input [Allocation](#) passed to the kernel launch. By default, the kernel is run across an entire [Allocation](#), with one execution of the kernel body per [Element](#) in the [Allocation](#). The third notable feature is the return type of the kernel. The value returned from the kernel is automatically written to the appropriate location in the output [Allocation](#). The RenderScript runtime checks to ensure that the [Element](#) types of the input and output Allocations match the kernel's prototype; if they do not match, an exception is thrown.

A kernel may have an input [Allocation](#), an output [Allocation](#), or both. A kernel may not have more than one input or one output [Allocation](#). If more than one input or output is required, those objects should be bound to `rs_allocation` script globals and accessed from a kernel or invokable function via `rsGetElementAt_type()` or `rsSetElementAt_type()`.

A kernel may access the coordinates of the current execution using the `x`, `y`, and `z` arguments. These arguments are optional, but the type of the coordinate arguments must be `uint32_t`.

- An optional `init()` function. An `init()` function is a special type of invokable function that is run when the script is first instantiated. This allows for some computation to occur automatically at script creation.
- Some number of static script globals and functions. A static script global is equivalent to a script global except that it cannot be set from Java code. A static function is a standard C function that can be called from any kernel or invokable function in the script but is not exposed to the Java API. If a script global or function does not need to be called from Java code, it is highly recommended that those be declared `static`.

## Setting floating point precision

You can control the required level of floating point precision in a script. This is useful if full IEEE 754-2008 standard (used by default) is not required. The following pragmas can set a different level of floating point precision:

- `#pragma rs_fp_full` (default if nothing is specified): For apps that require floating point precision as outlined by the IEEE 754-2008 standard.
- `#pragma rs_fp_relaxed` - For apps that don't require strict IEEE 754-2008 compliance and can tolerate less precision. This mode enables flush-to-zero for denorms and round-towards-zero.
- `#pragma rs_fp_imprecise` - For apps that don't have stringent precision requirements. This mode enables everything in `rs_fp_relaxed` along with the following:
  - Operations resulting in `-0.0` can return `+0.0` instead.
  - Operations on INF and NAN are undefined.

Most applications can use `rs_fp_relaxed` without any side effects. This may be very beneficial on some architectures due to additional optimizations only available with relaxed precision (such as SIMD CPU instructions).

# Using RenderScript from Java Code

Using RenderScript from Java code relies on the [android.renderscript](#) APIs. Most applications follow the same basic usage patterns:

- Initialize a RenderScript context.** The [RenderScript](#) context, created with [create\(Context\)](#), ensures that RenderScript can be used and provides an object to control the lifetime of all subsequent RenderScript objects. You should consider context creation to be a potentially long-running operation, since it may create resources on different pieces of hardware; it should not be in an application's critical path if at all possible. Typically, an application will have only a single RenderScript context at a time.
- Create at least one [Allocation](#) to be passed to a script.** An [Allocation](#) is a RenderScript object that provides storage for a fixed amount of data. Kernels in scripts take [Allocation](#) objects as their input and output, and [Allocation](#) objects can be accessed in kernels using `rsGetElementAt_type()` and `rsSetElementAt_type()` when bound as script globals. [Allocation](#) objects allow arrays to be passed from Java code to RenderScript code and vice-versa. [Allocation](#) objects are typically created using [createTyped\(RenderScript, Type\)](#) or [createFromBitmap\(RenderScript, Bitmap\)](#).
- Create whatever scripts are necessary.** There are two types of scripts available to you when using RenderScript:
  - **ScriptC:** These are the user-defined scripts as described in [Writing a RenderScript Kernel](#) above. Every script has a Java class reflected by the RenderScript compiler in order to make it easy to access the script from Java code; this class will have the name `ScriptC_filename`. For example, if the kernel above was located in `invert.rs` and a RenderScript context was already located in `mRS`, the Java code to instantiate the script would be:

```
ScriptC_invert invert = new ScriptC_invert(mRenderScript);
```

- **ScriptIntrinsic:** These are built-in RenderScript kernels for common operations, such as Gaussian blur, convolution, and image blending. For more information, see the subclasses of [ScriptIntrinsic](#).

- Populate Allocations with data.** Except for Allocations created with [android.renderscript](#), an Allocation will be populated with empty data when it is first created. To populate an Allocation, use one of the `copy` methods in [Allocation](#).
- Set any necessary script globals.** Globals may be set using methods in the same `ScriptC_filename` class with methods named `set_globalname`. For example, in order to set an `int` named `elements`, use the Java method `set_elements(int)`. RenderScript objects can also be set in kernels; for example, the `rs_allocation` variable named `lookup` can be set with the method `set_lookup(Allocation)`.
- Launch the appropriate kernels.** Methods to launch a given kernel will be reflected in the same `ScriptC_filename` class with methods named `forEach_kernelname()`. These launches are asynchronous, and launches will be serialized in the order in which they are launched. Depending on the arguments to the kernel, the method will take either one or two Allocations. By default, a kernel will execute over the entire input or output Allocation; to execute over a subset of that Allocation, pass an appropriate [Script.LaunchOptions](#) as the last argument to the `forEach` method.

Invoked functions can be launched using the `invoke_functionname` methods reflected in the same `ScriptC_filename` class.

- Copy data out of [Allocation](#) objects.** In order to access data from an [Allocation](#) from Java code, that data must be copied back to Java buffers using one of the `copy` methods in [Allocation](#). These functions will synchronize with asynchronous kernel and function launches as necessary.
- Tear down the RenderScript context.** The RenderScript context can be destroyed with [destroy\(\)](#) or by allowing the RenderScript context object to be garbage collected. This will cause any further use of any object belonging to that context to throw an exception.

# Advanced RenderScript

## In this document

1. [RenderScript Runtime Layer](#)
2. [Reflected Layer](#)
  1. [Functions](#)
  2. [Variables](#)
  3. [Pointers](#)
  4. [Structs](#)
3. [Memory Allocation APIs](#)
4. [Working with Memory](#)
  1. [Allocating and binding memory to the RenderScript](#)
  2. [Reading and writing to memory](#)

Because applications that utilize RenderScript still run inside of the Android VM, you have access to all of the framework APIs that you are familiar with, but can utilize RenderScript when appropriate. To facilitate this interaction between the framework and the RenderScript runtime, an intermediate layer of code is also present to facilitate communication and memory management between the two levels of code. This document goes into more detail about these different layers of code as well as how memory is shared between the Android VM and RenderScript runtime.

## RenderScript Runtime Layer

Your RenderScript code is compiled and executed in a compact and well-defined runtime layer. The RenderScript runtime APIs offer support for intensive computation that is portable and automatically scalable to the amount of cores available on a processor.

**Note:** The standard C functions in the NDK must be guaranteed to run on a CPU, so RenderScript cannot access these libraries, because RenderScript is designed to run on different types of processors.

You define your RenderScript code in `.rs` and `.rsh` files in the `src/` directory of your Android project. The code is compiled to intermediate bytecode by the `llvm` compiler that runs as part of an Android build. When your application runs on a device, the bytecode is then compiled (just-in-time) to machine code by another `llvm` compiler that resides on the device. The machine code is optimized for the device and also cached, so subsequent uses of the RenderScript enabled application does not recompile the bytecode.

Some key features of the RenderScript runtime libraries include:

- Memory allocation request features
- A large collection of math functions with both scalar and vector typed overloaded versions of many common routines. Operations such as adding, multiplying, dot product, and cross product are available as well as atomic arithmetic and comparison functions.
- Conversion routines for primitive data types and vectors, matrix routines, and date and time routines
- Data types and structures to support the RenderScript system such as Vector types for defining two-, three-, or four-vectors.
- Logging functions

See the RenderScript runtime API reference for more information on the available functions.

# Reflected Layer

The reflected layer is a set of classes that the Android build tools generate to allow access to the RenderScript runtime from the Android framework. This layer also provides methods and constructors that allow you to allocate and work with memory for pointers that are defined in your RenderScript code. The following list describes the major components that are reflected:

- Every .rs file that you create is generated into a class named `project_root/gen/package/name/ScriptC_renderscript_filename` of type [ScriptC](#). This file is the .java version of your .rs file, which you can call from the Android framework. This class contains the following items reflected from the .rs file:
  - Non-static functions
  - Non-static, global RenderScript variables. Accessor methods are generated for each variable, so you can read and write the RenderScript variables from the Android framework. If a global variable is initialized at the RenderScript runtime layer, those values are used to initialize the corresponding values in the Android framework layer. If global variables are marked as `const`, then a `set` method is not generated.
  - Global pointers
- A struct is reflected into its own class named `project_root/gen/package/name/ScriptField_struct_name`, which extends [Script.FieldBase](#). This class represents an array of the struct, which allows you to allocate memory for one or more instances of this struct.

## Functions

Functions are reflected into the script class itself, located in `project_root/gen/package/name/ScriptC_renderscript_filename`. For example, if you declare the following function in your RenderScript code:

```
void touch(float x, float y, float pressure, int id) {  
    if (id >= 10) {  
        return;  
    }  
  
    touchPos[id].x = x;  
    touchPos[id].y = y;  
    touchPressure[id] = pressure;  
}
```

then the following code is generated:

```
public void invoke_touch(float x, float y, float pressure, int id) {  
    FieldPacker touch_fp = new FieldPacker(16);  
    touch_fp.addF32(x);  
    touch_fp.addF32(y);  
    touch_fp.addF32(pressure);  
    touch_fp.addI32(id);  
    invoke(mExportFuncIdx_touch, touch_fp);  
}
```

Functions cannot have a return value, because the RenderScript system is designed to be asynchronous. When your Android framework code calls into RenderScript, the call is queued and is executed when possible. This

restriction allows the RenderScript system to function without constant interruption and increases efficiency. If functions were allowed to have return values, the call would block until the value was returned.

If you want the RenderScript code to send a value back to the Android framework, use the [rsSendToClient\(\)](#) function.

## Variables

Variables of supported types are reflected into the script class itself, located in `project_root/gen/package/name/ScriptC_renderscript_filename`. A set of accessor methods are generated for each variable. For example, if you declare the following variable in your RenderScript code:

```
uint32_t unsignedInteger = 1;
```

then the following code is generated:

```
private long mExportVar_unsignedInteger;
public void set_unsignedInteger(long v) {
    mExportVar_unsignedInteger = v;
    setVar(mExportVarIdx_unsignedInteger, v);
}

public long get_unsignedInteger() {
    return mExportVar_unsignedInteger;
}
```

## Structs

Structs are reflected into their own classes, located in `<project_root>/gen/com/example/render-script/ScriptField_struct_name`. This class represents an array of the struct and allows you to allocate memory for a specified number of structs. For example, if you declare the following struct:

```
typedef struct Point {
    float2 position;
    float size;
} Point_t;
```

then the following code is generated in `ScriptField_Point.java`:

```
package com.example.android.rs.hellocompute;

import android.renderscript.*;
import android.content.res.Resources;

/**
 * @hide
 */
public class ScriptField_Point extends android.renderscript.Script.FieldBase {

    static public class Item {
        public static final int sizeof = 12;
        Float2 position;
    }
}
```

```
    float size;

    Item() {
        position = new Float2();
    }
}

private Item mItemArray[];
private FieldPacker mIOBuffer;
public static Element createElement(RenderScript rs) {
    Element.Builder eb = new Element.Builder(rs);
    eb.add(Element.F32_2(rs), "position");
    eb.add(Element.F32(rs), "size");
    return eb.create();
}

public ScriptField_Point(RenderScript rs, int count) {
    mItemArray = null;
    mIOBuffer = null;
    mElement = createElement(rs);
    init(rs, count);
}

public ScriptField_Point(RenderScript rs, int count, int usages) {
    mItemArray = null;
    mIOBuffer = null;
    mElement = createElement(rs);
    init(rs, count, usages);
}

private void copyToArray(Item i, int index) {
    if (mIOBuffer == null) mIOBuffer = new FieldPacker(Item.sizeof * getType().getTypedefSize());
    mIOBuffer.reset(index * Item.sizeof);
    mIOBuffer.addF32(i.position);
    mIOBuffer.addF32(i.size);
}

public void set(Item i, int index, boolean copyNow) {
    if (mItemArray == null) mItemArray = new Item[getType().getX() /* count */];
    mItemArray[index] = i;
    if (copyNow) {
        copyToArray(i, index);
        mAllocation.setFromFieldPacker(index, mIOBuffer);
    }
}

public Item get(int index) {
    if (mItemArray == null) return null;
    return mItemArray[index];
}

public void set_position(int index, Float2 v, boolean copyNow) {
    if (mIOBuffer == null) mIOBuffer = new FieldPacker(Item.sizeof * getType().getTypedefSize());
    mIOBuffer.reset(index * Item.sizeof);
    mIOBuffer.addF32(v.x);
    mIOBuffer.addF32(v.y);
}
```

```
        if (mItemArray == null) mItemArray = new Item[getType().getX() /* count */];
        if (mItemArray[index] == null) mItemArray[index] = new Item();
        mItemArray[index].position = v;
        if (copyNow) {
            mIOBuffer.reset(index * Item.sizeof);
            mIOBuffer.addF32(v);
            FieldPacker fp = new FieldPacker(8);
            fp.addF32(v);
            mAllocation.setFromFieldPacker(index, 0, fp);
        }
    }

public void set_size(int index, float v, boolean copyNow) {
    if (mIOBuffer == null) mIOBuffer = new FieldPacker(Item.sizeof * getType().getX());
    if (mItemArray == null) mItemArray = new Item[getType().getX() /* count */];
    if (mItemArray[index] == null) mItemArray[index] = new Item();
    mItemArray[index].size = v;
    if (copyNow) {
        mIOBuffer.reset(index * Item.sizeof + 8);
        mIOBuffer.addF32(v);
        FieldPacker fp = new FieldPacker(4);
        fp.addF32(v);
        mAllocation.setFromFieldPacker(index, 1, fp);
    }
}

public Float2 get_position(int index) {
    if (mItemArray == null) return null;
    return mItemArray[index].position;
}

public float get_size(int index) {
    if (mItemArray == null) return 0;
    return mItemArray[index].size;
}

public void copyAll() {
    for (int ct = 0; ct < mItemArray.length; ct++) copyToArray(mItemArray[ct], mAllocation.setFromFieldPacker(0, mIOBuffer));
}

public void resize(int newSize) {
    if (mItemArray != null) {
        int oldSize = mItemArray.length;
        int copySize = Math.min(oldSize, newSize);
        if (newSize == oldSize) return;
        Item ni[] = new Item[newSize];
        System.arraycopy(mItemArray, 0, ni, 0, copySize);
        mItemArray = ni;
    }
    mAllocation.resize(newSize);
    if (mIOBuffer != null) mIOBuffer = new FieldPacker(Item.sizeof * getType().getX());
}
```

The generated code is provided to you as a convenience to allocate memory for structs requested by the RenderScript runtime and to interact with `structs` in memory. Each `struct`'s class defines the following methods and constructors:

- Overloaded constructors that allow you to allocate memory. The `ScriptField_struct_name(RenderScript rs, int count)` constructor allows you to define the number of structures that you want to allocate memory for with the `count` parameter. The `ScriptField_struct_name(RenderScript rs, int count, int usages)` constructor defines an extra parameter, `usages`, that lets you specify the memory space of this memory allocation. There are four memory space possibilities:
  - `USAGE_SCRIPT`: Allocates in the script memory space. This is the default memory space if you do not specify a memory space.
  - `USAGE_GRAPHICS_TEXTURE`: Allocates in the texture memory space of the GPU.
  - `USAGE_GRAPHICS_VERTEX`: Allocates in the vertex memory space of the GPU.
  - `USAGE_GRAPHICS_CONSTANTS`: Allocates in the constants memory space of the GPU that is used by the various program objects.

You can specify multiple memory spaces by using the bitwise OR operator. Doing so notifies the RenderScript runtime that you intend on accessing the data in the specified memory spaces. The following example allocates memory for a custom data type in both the script and vertex memory spaces:

```
ScriptField_Point touchPoints = new ScriptField_Point(myRenderScr  
Allocation.USAGE_SCRIPT | Allocation.USAGE_GRAPHICS_VERTEX);
```

- A static nested class, `Item`, allows you to create an instance of the `struct`, in the form of an object. This nested class is useful if it makes more sense to work with the `struct` in your Android code. When you are done manipulating the object, you can push the object to the allocated memory by calling `set(Item i, int index, boolean copyNow)` and setting the `Item` to the desired position in the array. The RenderScript runtime automatically has access to the newly written memory.
- Accessor methods to get and set the values of each field in a struct. Each of these accessor methods have an `index` parameter to specify the `struct` in the array that you want to read or write to. Each setter method also has a `copyNow` parameter that specifies whether or not to immediately sync this memory to the RenderScript runtime. To sync any memory that has not been synced, call `copyAll()`.
- The `createElement()` method creates a description of the struct in memory. This description is used to allocate memory consisting of one or many elements.
- `resize()` works much like a `realloc()` in C, allowing you to expand previously allocated memory, maintaining the current values that were previously created.
- `copyAll()` synchronizes memory that was set on the framework level to the RenderScript runtime. When you call a set accessor method on a member, there is an optional `copyNow` boolean parameter that you can specify. Specifying `true` synchronizes the memory when you call the method. If you specify `false`, you can call `copyAll()` once, and it synchronizes memory for all the properties that are not yet synchronized.

## Pointers

Pointers are reflected into the script class itself, located in `project_root/gen/package/name/ScriptC_renderScript_filename`. You can declare pointers to a `struct` or any of the supported RenderScript types, but a `struct` cannot contain pointers or nested arrays. For example, if you declare the following pointers to a `struct` and `int32_t`

```
typedef struct Point {  
    float2 position;
```

```

    float size;
} Point_t;

Point_t *touchPoints;
int32_t *intPointer;

```

then the following code is generated in:

```

private ScriptField_Point mExportVar_touchPoints;
public void bind_touchPoints(ScriptField_Point v) {
    mExportVar_touchPoints = v;
    if (v == null) bindAllocation(null, mExportVarIdx_touchPoints);
    else bindAllocation(v.getAllocation(), mExportVarIdx_touchPoints);
}

public ScriptField_Point get_touchPoints() {
    return mExportVar_touchPoints;
}

private Allocation mExportVar_intPointer;
public void bind_intPointer(Allocation v) {
    mExportVar_intPointer = v;
    if (v == null) bindAllocation(null, mExportVarIdx_intPointer);
    else bindAllocation(v, mExportVarIdx_intPointer);
}

public Allocation get_intPointer() {
    return mExportVar_intPointer;
}

```

A `get` method and a special method named `bind_pointer_name` (instead of a `set()` method) is generated. This method allows you to bind the memory that is allocated in the Android VM to the RenderScript runtime (you cannot allocate memory in your `.rs` file). For more information, see [Working with Allocated Memory](#).

## Memory Allocation APIs

Applications that use RenderScript still run in the Android VM. The actual RenderScript code, however, runs natively and needs access to the memory allocated in the Android VM. To accomplish this, you must attach the memory that is allocated in the VM to the RenderScript runtime. This process, called binding, allows the RenderScript runtime to seamlessly work with memory that it requests but cannot explicitly allocate. The end result is essentially the same as if you had called `malloc` in C. The added benefit is that the Android VM can carry out garbage collection as well as share memory with the RenderScript runtime layer. Binding is only necessary for dynamically allocated memory. Statically allocated memory is automatically created for your RenderScript code at compile time. See [Figure 1](#) for more information on how memory allocation occurs.

To support this memory allocation system, there are a set of APIs that allow the Android VM to allocate memory and offer similar functionality to a `malloc` call. These classes essentially describe how memory should be allocated and also carry out the allocation. To better understand how these classes work, it is useful to think of them in relation to a simple `malloc` call that can look like this:

```
array = (int *)malloc(sizeof(int)*10);
```

The `malloc` call can be broken up into two parts: the size of the memory being allocated (`sizeof(int)`), along with how many units of that memory should be allocated (10). The Android framework provides classes for these two parts as well as a class to represent `malloc` itself.

The [Element](#) class represents the (`sizeof(int)`) portion of the `malloc` call and encapsulates one cell of a memory allocation, such as a single float value or a struct. The [Type](#) class encapsulates the [Element](#) and the amount of elements to allocate (10 in our example). You can think of a [Type](#) as an array of [Elements](#). The [Allocation](#) class does the actual memory allocation based on a given [Type](#) and represents the actual allocated memory.

In most situations, you do not need to call these memory allocation APIs directly. The reflected layer classes generate code to use these APIs automatically and all you need to do to allocate memory is call a constructor that is declared in one of the reflected layer classes and then bind the resulting memory [Allocation](#) to the RenderScript. There are some situations where you would want to use these classes directly to allocate memory on your own, such as loading a bitmap from a resource or when you want to allocate memory for pointers to primitive types. You can see how to do this in the [Allocating and binding memory to the RenderScript](#) section. The following table describes the three memory management classes in more detail:

Android Object Type	Description
<a href="#">Element</a>	An element describes one cell of a memory allocation and can have two forms: basic or complex.  A basic element contains a single component of data of any valid RenderScript data type. Examples of basic element data types include a single <code>float</code> value, a <code>float4</code> vector, or a single RGB-565 color.
<a href="#">Type</a>	Complex elements contain a list of basic elements and are created from <code>structs</code> that you declare in your RenderScript code. For instance an allocation can contain multiple <code>structs</code> arranged in order in memory. Each struct is considered as its own element, rather than each data type within that struct.  A type is a memory allocation template and consists of an element and one or more dimensions. It describes the layout of the memory (basically an array of <a href="#">Elements</a> ) but does not allocate the memory for the data that it describes.  A type consists of five dimensions: X, Y, Z, LOD (level of detail), and Faces (of a cube map). You can assign the X,Y,Z dimensions to any positive integer value within the constraints of available memory. A single dimension allocation has an X dimension of greater than zero while the Y and Z dimensions are zero to indicate not present. For example, an allocation of x=10, y=1 is considered two dimensional and x=10, y=0 is considered one dimensional. The LOD and Faces dimensions are booleans to indicate present or not present.
<a href="#">Allocation</a>	An allocation provides the memory for applications based on a description of the memory that is represented by a <a href="#">Type</a> . Allocated memory can exist in many memory spaces concurrently. If memory is modified in one space, you must explicitly synchronize the memory, so that it is updated in all the other spaces in which it exists.  Allocation data is uploaded in one of two primary ways: type checked and type unchecked. For simple arrays there are <code>copyFrom()</code> functions that take an array from the Android system and

copy it to the native layer memory store. The unchecked variants allow the Android system to copy over arrays of structures because it does not support structures. For example, if there is an allocation that is an array of n floats, the data contained in a float[n] array or a byte [n\*4] array can be copied.

## Working with Memory

Non-static, global variables that you declare in your RenderScript are allocated memory at compile time. You can work with these variables directly in your RenderScript code without having to allocate memory for them at the Android framework level. The Android framework layer also has access to these variables with the provided accessor methods that are generated in the reflected layer classes. If these variables are initialized at the RenderScript runtime layer, those values are used to initialize the corresponding values in the Android framework layer. If global variables are marked as const, then a set method is not generated.

**Note:** If you are using certain RenderScript structures that contain pointers, such as `rs_program_fragment` and `rs_allocation`, you have to obtain an object of the corresponding Android framework class first and then call the `set` method for that structure to bind the memory to the RenderScript runtime. You cannot directly manipulate these structures at the RenderScript runtime layer. This restriction is not applicable to user-defined structures that contain pointers, because they cannot be exported to a reflected layer class in the first place. A compiler error is generated if you try to declare a non-static, global struct that contains a pointer.

RenderScript also has support for pointers, but you must explicitly allocate the memory in your Android framework code. When you declare a global pointer in your `.rs` file, you allocate memory through the appropriate reflected layer class and bind that memory to the native RenderScript layer. You can interact with this memory from the Android framework layer as well as the RenderScript layer, which offers you the flexibility to modify variables in the most appropriate layer.

### Allocating and binding dynamic memory to the RenderScript

To allocate dynamic memory, you need to call the constructor of a `Script.FieldBase` class, which is the most common way. An alternative is to create an `Allocation` manually, which is required for things such as primitive type pointers. You should use a `Script.FieldBase` class constructor whenever available for simplicity. After obtaining a memory allocation, call the reflected `bind` method of the pointer to bind the allocated memory to the RenderScript runtime.

The example below allocates memory for both a primitive type pointer, `intPointer`, and a pointer to a struct, `touchPoints`. It also binds the memory to the RenderScript:

```
private RenderScript myRenderScript;
private ScriptC_example script;
private Resources resources;

public void init(RenderScript rs, Resources res) {
    myRenderScript = rs;
    resources = res;

    //allocate memory for the struct pointer, calling the constructor
    ScriptField_Point touchPoints = new ScriptField_Point(myRenderScript, 2);

    //Create an element manually and allocate memory for the int pointer
```

```

intPointer = Allocation.createSized(myRenderScript, Element.I32(myRenderScript));
//create an instance of the RenderScript, pointing it to the bytecode resource
mScript = new ScriptC_example(myRenderScript, resources, R.raw.example);

//bind the struct and int pointers to the RenderScript
mScript.bind_touchPoints(touchPoints);
script.bind_intPointer(intPointer);

...
}

```

## Reading and writing to memory

You can read and write to statically and dynamically allocated memory both at the RenderScript runtime and Android framework layer.

Statically allocated memory comes with a one-way communication restriction at the RenderScript runtime level. When RenderScript code changes the value of a variable, it is not communicated back to the Android framework layer for efficiency purposes. The last value that is set from the Android framework is always returned during a call to a `get` method. However, when Android framework code modifies a variable, that change can be communicated to the RenderScript runtime automatically or synchronized at a later time. If you need to send data from the RenderScript runtime to the Android framework layer, you can use the [rsSendToClient\(\)](#) function to overcome this limitation.

When working with dynamically allocated memory, any changes at the RenderScript runtime layer are propagated back to the Android framework layer if you modified the memory allocation using its associated pointer. Modifying an object at the Android framework layer immediately propagates that change back to the RenderScript runtime layer.

## Reading and writing to global variables

Reading and writing to global variables is a straightforward process. You can use the accessor methods at the Android framework level or set them directly in the RenderScript code. Keep in mind that any changes that you make in your RenderScript code are not propagated back to the Android framework layer.

For example, given the following struct declared in a file named `rsfile.rs`:

```

typedef struct Point {
    int x;
    int y;
} Point_t;

Point_t point;

```

You can assign values to the struct like this directly in `rsfile.rs`. These values are not propagated back to the Android framework level:

```

point.x = 1;
point.y = 1;

```

You can assign values to the struct at the Android framework layer like this. These values are propagated back to the RenderScript runtime level:

```
ScriptC_rsfile mScript;  
...  
Item i = new ScriptField_Point.Item();  
i.x = 1;  
i.y = 1;  
mScript.set_point(i);
```

You can read the values in your RenderScript code like this:

```
rsDebug("Printing out a Point", point.x, point.y);
```

You can read the values in the Android framework layer with the following code. Keep in mind that this code only returns a value if one was set at the Android framework level. You will get a null pointer exception if you only set the value at the RenderScript runtime level:

```
Log.i("TAGNAME", "Printing out a Point: " + mScript.get_point().x + " " + mScript.get_point().y);  
System.out.println(point.get_x() + " " + point.get_y());
```

## Reading and writing global pointers

Assuming that memory has been allocated in the Android framework level and bound to the RenderScript runtime, you can read and write memory from the Android framework level by using the `get` and `set` methods for that pointer. In the RenderScript runtime layer, you can read and write to memory with pointers as normal and the changes are propagated back to the Android framework layer, unlike with statically allocated memory.

For example, given the following pointer to a struct in a file named `rsfile.rs`:

```
typedef struct Point {  
    int x;  
    int y;  
} Point_t;  
  
Point_t *point;
```

Assuming you already allocated memory at the Android framework layer, you can access values in the struct as normal. Any changes you make to the struct via its pointer variable are automatically available to the Android framework layer:

```
point[index].x = 1;  
point[index].y = 1;
```

You can read and write values to the pointer at the Android framework layer as well:

```
ScriptField_Point p = new ScriptField_Point(mRS, 1);  
    Item i = new ScriptField_Point.Item();  
    i.x=100;  
    i.y = 100;  
    p.set(i, 0, true);  
    mScript.bind_point(p);  
  
    points.get_x(0);           //read x and y from index 0  
    points.get_x(0);
```

Once memory is already bound, you do not have to rebind the memory to the RenderScript runtime every time you make a change to a value.

# Runtime API Reference

Overview	
Globals	
Structs	

RenderScript is a high-performance runtime that provides compute operations at the native level. RenderScript code is compiled on devices at runtime to allow platform-independence as well. This reference documentation describes the RenderScript runtime APIs, which you can utilize to write RenderScript code in C99. The RenderScript compute header files are automatically included for you.

To use RenderScript, you need to utilize the RenderScript runtime APIs documented here as well as the Android framework APIs for RenderScript. For documentation on the Android framework APIs, see the [android.renderscript](#) package reference. For more information on how to develop with RenderScript and how the runtime and Android framework APIs interact, see the [RenderScript developer guide](#) and the [RenderScript samples](#).

# Media and Camera

Add video, audio, and photo capabilities to your app with Android's robust APIs for playing and recording media.

## Blog Articles

### [Allowing applications to play nice\(r\) with each other: Handling remote control buttons](#)

If your media playback application creates a media playback service, just like Music, that responds to the media button events, how will the user know where those events are going to? Music, or your new application?

### [Making Android Games that Play Nice](#)

Making a game on Android is easy. Making a great game for a mobile, multitasking, often multi-core, multi-purpose system like Android is trickier. Even the best developers frequently make mistakes in the way they interact with the Android system and with other applications

### [More Android Games that Play Nice](#)

Android users get used to using the back key. We expect the volume keys to work in some intuitive fashion. We expect that the home key behaves in a manner consistent with the Android navigation paradigm.

## Training

### [Capturing Photos](#)

This class gets you clicking fast with some super-easy ways of leveraging existing camera applications. In later lessons, you dive deeper and learn how to control the camera hardware directly.

### [Managing Audio Playback](#)

After this class, you will be able to build apps that respond to hardware audio key presses, which request audio focus when playing audio, and which respond appropriately to changes in audio focus caused by the system or other applications.

# Media Playback

## In this document

1. [The Basics](#)
2. [Manifest Declarations](#)
3. [Using MediaPlayer](#)
  1. [Asynchronous Preparation](#)
  2. [Managing State](#)
  3. [Releasing the MediaPlayer](#)
4. [Using a Service with MediaPlayer](#)
  1. [Running asynchronously](#)
  2. [Handling asynchronous errors](#)
  3. [Using wake locks](#)
  4. [Running as a foreground service](#)
  5. [Handling audio focus](#)
  6. [Performing cleanup](#)
5. [Handling the AUDIO\\_BECOMING\\_NOISY Intent](#)
6. [Retrieving Media from a Content Resolver](#)

## Key classes

1. [MediaPlayer](#)
2. [AudioManager](#)
3. [SoundPool](#)

## See also

1. [JetPlayer](#)
2. [Audio Capture](#)
3. [Android Supported Media Formats](#)
4. [Data Storage](#)

The Android multimedia framework includes support for playing variety of common media types, so that you can easily integrate audio, video and images into your applications. You can play audio or video from media files stored in your application's resources (raw resources), from standalone files in the filesystem, or from a data stream arriving over a network connection, all using [MediaPlayer](#) APIs.

This document shows you how to write a media-playing application that interacts with the user and the system in order to obtain good performance and a pleasant user experience.

**Note:** You can play back the audio data only to the standard output device. Currently, that is the mobile device speaker or a Bluetooth headset. You cannot play sound files in the conversation audio during a call.

## The Basics

The following classes are used to play sound and video in the Android framework:

### [MediaPlayer](#)

This class is the primary API for playing sound and video.

## [AudioManager](#)

This class manages audio sources and audio output on a device.

## Manifest Declarations

Before starting development on your application using MediaPlayer, make sure your manifest has the appropriate declarations to allow use of related features.

- **Internet Permission** - If you are using MediaPlayer to stream network-based content, your application must request network access.

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

- **Wake Lock Permission** - If your player application needs to keep the screen from dimming or the processor from sleeping, or uses the [MediaPlayer.setScreenOnWhilePlaying\(\)](#) or [MediaPlayer.setWakeMode\(\)](#) methods, you must request this permission.

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

## Using MediaPlayer

One of the most important components of the media framework is the [MediaPlayer](#) class. An object of this class can fetch, decode, and play both audio and video with minimal setup. It supports several different media sources such as:

- Local resources
- Internal URIs, such as one you might obtain from a Content Resolver
- External URLs (streaming)

For a list of media formats that Android supports, see the [Android Supported Media Formats](#) document.

Here is an example of how to play audio that's available as a local raw resource (saved in your application's res/raw/ directory):

```
MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.sound_file_1);
mediaPlayer.start(); // no need to call prepare(); create() does that for you
```

In this case, a "raw" resource is a file that the system does not try to parse in any particular way. However, the content of this resource should not be raw audio. It should be a properly encoded and formatted media file in one of the supported formats.

And here is how you might play from a URI available locally in the system (that you obtained through a Content Resolver, for instance):

```
Uri myUri = ....; // initialize Uri here
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(getApplicationContext(), myUri);
mediaPlayer.prepare();
mediaPlayer.start();
```

Playing from a remote URL via HTTP streaming looks like this:

```
String url = "http://....."; // your URL here
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(url);
mediaPlayer.prepare(); // might take long! (for buffering, etc)
mediaPlayer.start();
```

**Note:** If you're passing a URL to stream an online media file, the file must be capable of progressive download.

**Caution:** You must either catch or pass [IllegalArgumentException](#) and [IOException](#) when using [setDataSource\(\)](#), because the file you are referencing might not exist.

## Asynchronous Preparation

Using [MediaPlayer](#) can be straightforward in principle. However, it's important to keep in mind that a few more things are necessary to integrate it correctly with a typical Android application. For example, the call to [prepare\(\)](#) can take a long time to execute, because it might involve fetching and decoding media data. So, as is the case with any method that may take long to execute, you should **never call it from your application's UI thread**. Doing that will cause the UI to hang until the method returns, which is a very bad user experience and can cause an ANR (Application Not Responding) error. Even if you expect your resource to load quickly, remember that anything that takes more than a tenth of a second to respond in the UI will cause a noticeable pause and will give the user the impression that your application is slow.

To avoid hanging your UI thread, spawn another thread to prepare the [MediaPlayer](#) and notify the main thread when done. However, while you could write the threading logic yourself, this pattern is so common when using [MediaPlayer](#) that the framework supplies a convenient way to accomplish this task by using the [prepareAsync\(\)](#) method. This method starts preparing the media in the background and returns immediately. When the media is done preparing, the [onPrepared\(\)](#) method of the [MediaPlayer.OnPreparedListener](#), configured through [setOnPreparedListener\(\)](#) is called.

## Managing State

Another aspect of a [MediaPlayer](#) that you should keep in mind is that it's state-based. That is, the [MediaPlayer](#) has an internal state that you must always be aware of when writing your code, because certain operations are only valid when the player is in specific states. If you perform an operation while in the wrong state, the system may throw an exception or cause other undesirable behaviors.

The documentation in the [MediaPlayer](#) class shows a complete state diagram, that clarifies which methods move the [MediaPlayer](#) from one state to another. For example, when you create a new [MediaPlayer](#), it is in the *Idle* state. At that point, you should initialize it by calling [setDataSource\(\)](#), bringing it to the *Initialized* state. After that, you have to prepare it using either the [prepare\(\)](#) or [prepareAsync\(\)](#) method. When the [MediaPlayer](#) is done preparing, it will then enter the *Prepared* state, which means you can call [start\(\)](#) to make it play the media. At that point, as the diagram illustrates, you can move between the *Started*, *Paused* and *PlaybackCompleted* states by calling such methods as [start\(\)](#), [pause\(\)](#), and [seekTo\(\)](#), amongst others. When you call [stop\(\)](#), however, notice that you cannot call [start\(\)](#) again until you prepare the [MediaPlayer](#) again.

Always keep [the state diagram](#) in mind when writing code that interacts with a [MediaPlayer](#) object, because calling its methods from the wrong state is a common cause of bugs.

## Releasing the MediaPlayer

A [MediaPlayer](#) can consume valuable system resources. Therefore, you should always take extra precautions to make sure you are not hanging on to a [MediaPlayer](#) instance longer than necessary. When you are done with it, you should always call [release\(\)](#) to make sure any system resources allocated to it are properly released. For example, if you are using a [MediaPlayer](#) and your activity receives a call to [onStop\(\)](#), you must release the [MediaPlayer](#), because it makes little sense to hold on to it while your activity is not interacting with the user (unless you are playing media in the background, which is discussed in the next section). When your activity is resumed or restarted, of course, you need to create a new [MediaPlayer](#) and prepare it again before resuming playback.

Here's how you should release and then nullify your [MediaPlayer](#):

```
mediaPlayer.release();  
mediaPlayer = null;
```

As an example, consider the problems that could happen if you forgot to release the [MediaPlayer](#) when your activity is stopped, but create a new one when the activity starts again. As you may know, when the user changes the screen orientation (or changes the device configuration in another way), the system handles that by restarting the activity (by default), so you might quickly consume all of the system resources as the user rotates the device back and forth between portrait and landscape, because at each orientation change, you create a new [MediaPlayer](#) that you never release. (For more information about runtime restarts, see [Handling Runtime Changes](#).)

You may be wondering what happens if you want to continue playing "background media" even when the user leaves your activity, much in the same way that the built-in Music application behaves. In this case, what you need is a [MediaPlayer](#) controlled by a [Service](#), as discussed in [Using a Service with MediaPlayer](#).

## Using a Service with MediaPlayer

If you want your media to play in the background even when your application is not onscreen—that is, you want it to continue playing while the user is interacting with other applications—then you must start a [Service](#) and control the [MediaPlayer](#) instance from there. You should be careful about this setup, because the user and the system have expectations about how an application running a background service should interact with the rest of the system. If your application does not fulfil those expectations, the user may have a bad experience. This section describes the main issues that you should be aware of and offers suggestions about how to approach them.

### Running asynchronously

First of all, like an [Activity](#), all work in a [Service](#) is done in a single thread by default—in fact, if you're running an activity and a service from the same application, they use the same thread (the "main thread") by default. Therefore, services need to process incoming intents quickly and never perform lengthy computations when responding to them. If any heavy work or blocking calls are expected, you must do those tasks asynchronously: either from another thread you implement yourself, or using the framework's many facilities for asynchronous processing.

For instance, when using a [MediaPlayer](#) from your main thread, you should call [prepareAsync\(\)](#) rather than [prepare\(\)](#), and implement a [MediaPlayer.OnPreparedListener](#) in order to be notified when the preparation is complete and you can start playing. For example:

```
public class MyService extends Service implements MediaPlayer.OnPreparedListener  
    private static final ACTION_PLAY = "com.example.action.PLAY";
```

```

MediaPlayer mMediaPlayer = null;

public int onStartCommand(Intent intent, int flags, int startId) {
    ...
    if (intent.getAction().equals(ACTION_PLAY)) {
        mMediaPlayer = ... // initialize it here
        mMediaPlayer.setOnPreparedListener(this);
        mMediaPlayer.prepareAsync(); // prepare async to not block main thread
    }
}

/** Called when MediaPlayer is ready */
public void onPrepared(MediaPlayer player) {
    player.start();
}
}

```

## Handling asynchronous errors

On synchronous operations, errors would normally be signaled with an exception or an error code, but whenever you use asynchronous resources, you should make sure your application is notified of errors appropriately. In the case of a [MediaPlayer](#), you can accomplish this by implementing a [MediaPlayer.OnErrorListener](#) and setting it in your [MediaPlayer](#) instance:

```

public class MyService extends Service implements MediaPlayer.OnErrorListener {
    MediaPlayer mMediaPlayer;

    public void initMediaPlayer() {
        // ...initialize the MediaPlayer here...

        mMediaPlayer.setOnErrorListener(this);
    }

    @Override
    public boolean onError(MediaPlayer mp, int what, int extra) {
        // ... react appropriately ...
        // The MediaPlayer has moved to the Error state, must be reset!
    }
}

```

It's important to remember that when an error occurs, the [MediaPlayer](#) moves to the *Error* state (see the documentation for the [MediaPlayer](#) class for the full state diagram) and you must reset it before you can use it again.

## Using wake locks

When designing applications that play media in the background, the device may go to sleep while your service is running. Because the Android system tries to conserve battery while the device is sleeping, the system tries to shut off any of the phone's features that are not necessary, including the CPU and the WiFi hardware. However, if your service is playing or streaming music, you want to prevent the system from interfering with your playback.

In order to ensure that your service continues to run under those conditions, you have to use "wake locks." A wake lock is a way to signal to the system that your application is using some feature that should stay available even if the phone is idle.

**Notice:** You should always use wake locks sparingly and hold them only for as long as truly necessary, because they significantly reduce the battery life of the device.

To ensure that the CPU continues running while your [MediaPlayer](#) is playing, call the [setWakeMode\(\)](#) method when initializing your [MediaPlayer](#). Once you do, the [MediaPlayer](#) holds the specified lock while playing and releases the lock when paused or stopped:

```
mMediaPlayer = new MediaPlayer();
// ... other initialization here ...
mMediaPlayer.setWakeMode(getApplicationContext(), PowerManager.PARTIAL_WAKE_LOCK);
```

However, the wake lock acquired in this example guarantees only that the CPU remains awake. If you are streaming media over the network and you are using Wi-Fi, you probably want to hold a [WifiLock](#) as well, which you must acquire and release manually. So, when you start preparing the [MediaPlayer](#) with the remote URL, you should create and acquire the Wi-Fi lock. For example:

```
WifiLock wifiLock = ((WifiManager) getSystemService(Context.WIFI_SERVICE))
    .createWifiLock(WifiManager.WIFI_MODE_FULL, "mylock");

wifiLock.acquire();
```

When you pause or stop your media, or when you no longer need the network, you should release the lock:

```
wifiLock.release();
```

## Running as a foreground service

Services are often used for performing background tasks, such as fetching emails, synchronizing data, downloading content, amongst other possibilities. In these cases, the user is not actively aware of the service's execution, and probably wouldn't even notice if some of these services were interrupted and later restarted.

But consider the case of a service that is playing music. Clearly this is a service that the user is actively aware of and the experience would be severely affected by any interruptions. Additionally, it's a service that the user will likely wish to interact with during its execution. In this case, the service should run as a "foreground service." A foreground service holds a higher level of importance within the system—the system will almost never kill the service, because it is of immediate importance to the user. When running in the foreground, the service also must provide a status bar notification to ensure that users are aware of the running service and allow them to open an activity that can interact with the service.

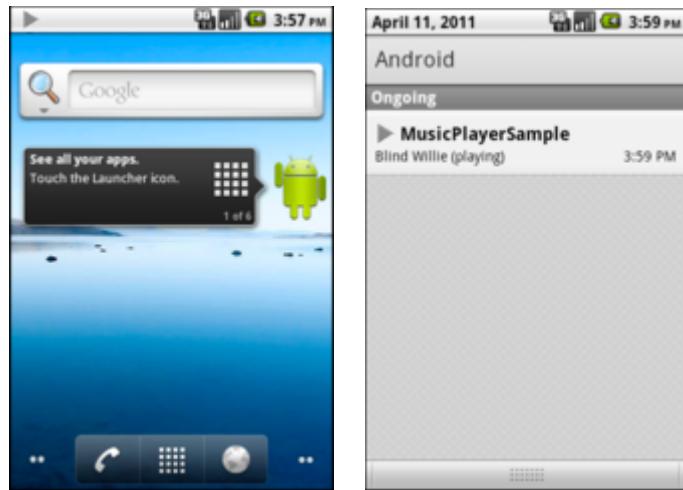
In order to turn your service into a foreground service, you must create a [Notification](#) for the status bar and call [startForeground\(\)](#) from the [Service](#). For example:

```
String songName;
// assign the song name to songName
PendingIntent pi = PendingIntent.getActivity(getApplicationContext(), 0,
    new Intent(getApplicationContext(), MainActivity.class),
    PendingIntent.FLAG_UPDATE_CURRENT);
Notification notification = new Notification();
notification.tickerText = text;
notification.icon = R.drawable.play0;
```

```
notification.flags |= Notification.FLAG_ONGOING_EVENT;
notification.setLatestEventInfo(getApplicationContext(), "MusicPlayerSample",
        "Playing: " + songName, pi);
startForeground(NOTIFICATION_ID, notification);
```

While your service is running in the foreground, the notification you configured is visible in the notification area of the device. If the user selects the notification, the system invokes the [PendingIntent](#) you supplied. In the example above, it opens an activity (`MainActivity`).

Figure 1 shows how your notification appears to the user:



**Figure 1.** Screenshots of a foreground service's notification, showing the notification icon in the status bar (left) and the expanded view (right).

You should only hold on to the "foreground service" status while your service is actually performing something the user is actively aware of. Once that is no longer true, you should release it by calling [stopForeground\(\)](#):

```
stopForeground(true);
```

For more information, see the documentation about [Services](#) and [Status Bar Notifications](#).

## Handling audio focus

Even though only one activity can run at any given time, Android is a multi-tasking environment. This poses a particular challenge to applications that use audio, because there is only one audio output and there may be several media services competing for its use. Before Android 2.2, there was no built-in mechanism to address this issue, which could in some cases lead to a bad user experience. For example, when a user is listening to music and another application needs to notify the user of something very important, the user might not hear the notification tone due to the loud music. Starting with Android 2.2, the platform offers a way for applications to negotiate their use of the device's audio output. This mechanism is called Audio Focus.

When your application needs to output audio such as music or a notification, you should always request audio focus. Once it has focus, it can use the sound output freely, but it should always listen for focus changes. If it is notified that it has lost the audio focus, it should immediately either kill the audio or lower it to a quiet level (known as "ducking"—there is a flag that indicates which one is appropriate) and only resume loud playback after it receives focus again.

Audio Focus is cooperative in nature. That is, applications are expected (and highly encouraged) to comply with the audio focus guidelines, but the rules are not enforced by the system. If an application wants to play

loud music even after losing audio focus, nothing in the system will prevent that. However, the user is more likely to have a bad experience and will be more likely to uninstall the misbehaving application.

To request audio focus, you must call [requestAudioFocus\(\)](#) from the [AudioManager](#), as the example below demonstrates:

```
AudioManager audioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
int result = audioManager.requestAudioFocus(this, AudioManager.STREAM_MUSIC,
    AudioManager.AUDIOFOCUS_GAIN);

if (result != AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {
    // could not get audio focus.
}
```

The first parameter to [requestAudioFocus\(\)](#) is an [AudioManager.OnAudioFocusChangeListener](#), whose [onAudioFocusChange\(\)](#) method is called whenever there is a change in audio focus. Therefore, you should also implement this interface on your service and activities. For example:

```
class MyService extends Service
    implements AudioManager.OnAudioFocusChangeListener {
    // ....
    public void onAudioFocusChange(int focusChange) {
        // Do something based on focus change...
    }
}
```

The `focusChange` parameter tells you how the audio focus has changed, and can be one of the following values (they are all constants defined in [AudioManager](#)):

- [AUDIOFOCUS\\_GAIN](#): You have gained the audio focus.
- [AUDIOFOCUS\\_LOSS](#): You have lost the audio focus for a presumably long time. You must stop all audio playback. Because you should expect not to have focus back for a long time, this would be a good place to clean up your resources as much as possible. For example, you should release the [MediaPlayer](#).
- [AUDIOFOCUS\\_LOSS\\_TRANSIENT](#): You have temporarily lost audio focus, but should receive it back shortly. You must stop all audio playback, but you can keep your resources because you will probably get focus back shortly.
- [AUDIOFOCUS\\_LOSS\\_TRANSIENT\\_CAN\\_DUCK](#): You have temporarily lost audio focus, but you are allowed to continue to play audio quietly (at a low volume) instead of killing audio completely.

Here is an example implementation:

```
public void onAudioFocusChange(int focusChange) {
    switch (focusChange) {
        case AudioManager.AUDIOFOCUS_GAIN:
            // resume playback
            if (mMediaPlayer == null) initMediaPlayer();
            else if (!mMediaPlayer.isPlaying()) mMediaPlayer.start();
            mMediaPlayer.setVolume(1.0f, 1.0f);
            break;

        case AudioManager.AUDIOFOCUS_LOSS:
            // Lost focus for an unbounded amount of time: stop playback and re
```

```

        if (mMediaPlayer.isPlaying()) mMediaPlayer.stop();
        mMediaPlayer.release();
        mMediaPlayer = null;
        break;

    case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT:
        // Lost focus for a short time, but we have to stop
        // playback. We don't release the media player because playback
        // is likely to resume
        if (mMediaPlayer.isPlaying()) mMediaPlayer.pause();
        break;

    case AudioManager.AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK:
        // Lost focus for a short time, but it's ok to keep playing
        // at an attenuated level
        if (mMediaPlayer.isPlaying()) mMediaPlayer.setVolume(0.1f, 0.1f);
        break;
    }
}

```

Keep in mind that the audio focus APIs are available only with API level 8 (Android 2.2) and above, so if you want to support previous versions of Android, you should adopt a backward compatibility strategy that allows you to use this feature if available, and fall back seamlessly if not.

You can achieve backward compatibility either by calling the audio focus methods by reflection or by implementing all the audio focus features in a separate class (say, `AudioFocusHelper`). Here is an example of such a class:

```

public class AudioFocusHelper implements AudioManager.OnAudioFocusChangeListener {
    AudioManager mAudioManager;

    // other fields here, you'll probably hold a reference to an interface
    // that you can use to communicate the focus changes to your Service

    public AudioFocusHelper(Context ctx, /* other arguments here */) {
        mAudioManager = (AudioManager) mContext.getSystemService(Context.AUDIO_SERVICE);
        // ...
    }

    public boolean requestFocus() {
        return AudioManager.AUDIOFOCUS_REQUEST_GRANTED ==
            mAudioManager.requestAudioFocus(mContext, AudioManager.STREAM_MUSIC,
                AudioManager.AUDIOFOCUS_GAIN);
    }

    public boolean abandonFocus() {
        return AudioManager.AUDIOFOCUS_REQUEST_GRANTED ==
            mAudioManager.abandonAudioFocus(this);
    }

    @Override
    public void onAudioFocusChange(int focusChange) {
        // let your service know about the focus change
    }
}

```

```
}
```

You can create an instance of `AudioFocusHelper` class only if you detect that the system is running API level 8 or above. For example:

```
if (android.os.Build.VERSION.SDK_INT >= 8) {
    mAudioFocusHelper = new AudioFocusHelper(getApplicationContext(), this);
} else {
    mAudioFocusHelper = null;
}
```

## Performing cleanup

As mentioned earlier, a `MediaPlayer` object can consume a significant amount of system resources, so you should keep it only for as long as you need and call `release()` when you are done with it. It's important to call this cleanup method explicitly rather than rely on system garbage collection because it might take some time before the garbage collector reclaims the `MediaPlayer`, as it's only sensitive to memory needs and not to shortage of other media-related resources. So, in the case when you're using a service, you should always override the `onDestroy()` method to make sure you are releasing the `MediaPlayer`:

```
public class MyService extends Service {
    MediaPlayer mMediaPlayer;
    // ...

    @Override
    public void onDestroy() {
        if (mMediaPlayer != null) mMediaPlayer.release();
    }
}
```

You should always look for other opportunities to release your `MediaPlayer` as well, apart from releasing it when being shut down. For example, if you expect not to be able to play media for an extended period of time (after losing audio focus, for example), you should definitely release your existing `MediaPlayer` and create it again later. On the other hand, if you only expect to stop playback for a very short time, you should probably hold on to your `MediaPlayer` to avoid the overhead of creating and preparing it again.

## Handling the AUDIO\_BECOMING\_NOISY Intent

Many well-written applications that play audio automatically stop playback when an event occurs that causes the audio to become noisy (output through external speakers). For instance, this might happen when a user is listening to music through headphones and accidentally disconnects the headphones from the device. However, this behavior does not happen automatically. If you don't implement this feature, audio plays out of the device's external speakers, which might not be what the user wants.

You can ensure your app stops playing music in these situations by handling the `AC-TION_AUDIO_BECOMING_NOISY` intent, for which you can register a receiver by adding the following to your manifest:

```
<receiver android:name=".MusicIntentReceiver">
    <intent-filter>
        <action android:name="android.media.AUDIO_BECOMING_NOISY" />
```

```
</intent-filter>
</receiver>
```

This registers the `MusicIntentReceiver` class as a broadcast receiver for that intent. You should then implement this class:

```
public class MusicIntentReceiver implements android.content.BroadcastReceiver {
    @Override
    public void onReceive(Context ctx, Intent intent) {
        if (intent.getAction().equals(
                android.media.AudioManager.ACTION_AUDIO_BECOMING_NOISY)) {
            // signal your service to stop playback
            // (via an Intent, for instance)
        }
    }
}
```

## Retrieving Media from a Content Resolver

Another feature that may be useful in a media player application is the ability to retrieve music that the user has on the device. You can do that by querying the [ContentResolver](#) for external media:

```
ContentResolver contentResolver = getContentResolver();
Uri uri = android.provider.MediaStore.Audio.Media.EXTERNAL_CONTENT_URI;
Cursor cursor = contentResolver.query(uri, null, null, null, null);
if (cursor == null) {
    // query failed, handle error.
} else if (!cursor.moveToFirst()) {
    // no media on the device
} else {
    int titleColumn = cursor.getColumnIndex(android.provider.MediaStore.Audio.Medi
    int idColumn = cursor.getColumnIndex(android.provider.MediaStore.Audio.Medi
    do {
        long thisId = cursor.getLong(idColumn);
        String thisTitle = cursor.getString(titleColumn);
        // ...process entry...
    } while (cursor.moveToNext());
}
```

To use this with the [MediaPlayer](#), you can do this:

```
long id = /* retrieve it from somewhere */;
Uri contentUri = ContentUris.withAppendedId(
    android.provider.MediaStore.Audio.Media.EXTERNAL_CONTENT_URI, id);

mMediaPlayer = new MediaPlayer();
mMediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mMediaPlayer.setDataSource(getApplicationContext(), contentUri);

// ...prepare and start...
```

# Supported Media Formats

## In this document

1. [Network Protocols](#)
2. [Core Media Formats](#)
3. [Video Encoding Recommendations](#)

## See also

1. [Multimedia and Camera](#)

## Key classes

1. [MediaPlayer](#)
2. [MediaRecorder](#)

This document describes the media codec, container, and network protocol support provided by the Android platform.

As an application developer, you are free to make use of any media codec that is available on any Android-powered device, including those provided by the Android platform and those that are device-specific. **However, it is a best practice to use media encoding profiles that are device-agnostic.**

## Network Protocols

The following network protocols are supported for audio and video playback:

- RTSP (RTP, SDP)
- HTTP/HTTPS progressive streaming
- HTTP/HTTPS live streaming [draft protocol](#):
  - MPEG-2 TS media files only
  - Protocol version 3 (Android 4.0 and above)
  - Protocol version 2 (Android 3.x)
  - Not supported before Android 3.0

**Note:** HTTPS is not supported before Android 3.1.

## Core Media Formats

The table below describes the media format support built into the Android platform. Note that any given mobile device may provide support for additional formats or file types not listed in the table.

**Note:** Media codecs that are not guaranteed to be available on all Android platform versions are accordingly noted in parentheses—for example "(Android 3.0+)".

**Table 1.** Core media format and codec support.

Type	Format / Codec	Encoder	Decoder	Details	Supported File Type(s) / Container Formats
Audio	AAC LC	•	•		• 3GPP (.3gp) • MPEG-4 (.mp4, .m4a)
	HE-AACv1 (AAC+)	• (Android 4.1+)	•	Support for mono/stereo/5.0/5.1 content with standard sampling rates from 8 to 48 kHz.	• ADTS raw AAC (.aac, decode in Android 3.1+, encode in Android 4.0+, ADIF not supported)
	HE-AACv2 (enhanced AAC+)	•	•	Support for stereo/5.0/5.1 content with standard sampling rates from 8 to 48 kHz.	• MPEG-TS (.ts, not seekable, Android 3.0+)
	AAC ELD (enhanced low delay AAC)	• (Android 4.1+)	• (Android 4.1+)	Support for mono/stereo content with standard sampling rates from 16 to 48 kHz	3GPP (.3gp)
	AMR-NB	•	•	4.75 to 12.2 kbps sampled @ 8kHz	3GPP (.3gp)
	AMR-WB	•	•	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	3GPP (.3gp)
	FLAC	• (Android 3.1+)		Mono/Stereo (no multichannel). Sample rates up to 48 kHz (but up to 44.1 kHz is recommended on devices with 44.1 kHz output, as the 48 to 44.1 kHz downampler does not include a low-pass filter). 16-bit recommended; no dither applied for 24-bit.	FLAC (.flac) only
	MP3	•		Mono/Stereo 8-320Kbps constant (CBR) or variable bit-rate (VBR)	MP3 (.mp3)
	MIDI	•		MIDI Type 0 and 1. DLS Version 1 and 2. XMF and Mobile XMF. Support for ringtone formats RTTTL/RTX, OTA, and iMelody	• Type 0 and 1 (.mid, .xmf, .mxmf) • RTTTL/RTX (.rtttl, .rtx) • OTA (.ota) • iMelody (.imy) • Ogg (.ogg) • Matroska (.mkv, Android 4.0+)
	Vorbis	•			
Image	PCM/WAVE	• (Android 4.1+)	•	8- and 16-bit linear PCM (rates up to limit of hardware). Sampling rates for raw PCM recordings at 8000, 16000 and 44100 Hz.	WAVE (.wav)
	JPEG	•	•	Base+progressive	JPEG (.jpg)
	GIF	•			GIF (.gif)
	PNG	•	•		PNG (.png)
	BMP	•			BMP (.bmp)
Video	WEBP	• (Android 4.0+)	• (Android 4.0+)		WebP (.webp)
	H.263	•	•		• 3GPP (.3gp) • MPEG-4 (.mp4)
	H.264	•			• 3GPP (.3gp)
	AVC	(Android 3.0+)		Baseline Profile (BP)	• MPEG-4 (.mp4) • MPEG-TS (.ts,

MPEG-4	•		AAC audio only, not seekable, An- droid 3.0+)
SP			3GPP (.3gp)
VP8	• (Android 4.3+)	• (Android 2.3.3+)	Streamable only in Android 4.0 and above
			• <a href="#">WebM</a> (.webm) • Matroska (.mkv, Android 4.0+)

## Video Encoding Recommendations

Table 2, below, lists examples of video encoding profiles and parameters that the Android media framework supports for playback in the H.264 Baseline Profile codec. While table 3 lists examples that the framework supports for playback in the VP8 media codec.

In addition to these encoding parameter recommendations, a device's available *video recording* profiles can be used as a proxy for media playback capabilities. These profiles can be inspected using the [CamcorderProfile](#) class, which is available since API level 8.

**Table 2.** Examples of supported video encoding parameters for the H.264 Baseline Profile codec.

	<b>SD (Low quality)</b>	<b>SD (High quality)</b>	<b>HD 720p (N/A on all devices)</b>
<b>Video resolution</b>	176 x 144 px	480 x 360 px	1280 x 720 px
<b>Video frame rate</b>	12 fps	30 fps	30 fps
<b>Video bitrate</b>	56 Kbps	500 Kbps	2 Mbps
<b>Audio codec</b>	AAC-LC	AAC-LC	AAC-LC
<b>Audio channels</b>	1 (mono)	2 (stereo)	2 (stereo)
<b>Audio bitrate</b>	24 Kbps	128 Kbps	192 Kbps

**Table 3.** Examples of supported video encoding parameters for the VP8 codec.

	<b>SD (Low quality)</b>	<b>SD (High quality)</b>	<b>HD 720p (N/A on all devices)</b>	<b>HD 1080p (N/A on all devices)</b>
<b>Video resolution</b>	320 x 180 px	640 x 360 px	1280 x 720 px	1920 x 1080 px
<b>Video frame rate</b>	30 fps	30 fps	30 fps	30 fps
<b>Video bitrate</b>	800 Kbps	2 Mbps	4 Mbps	10 Mbps

For video content that is streamed over HTTP or RTSP, there are additional requirements:

- For 3GPP and MPEG-4 containers, the `moov` atom must precede any `mdat` atoms, but must succeed the `ftyp` atom.
- For 3GPP, MPEG-4, and WebM containers, audio and video samples corresponding to the same time offset may be no more than 500 KB apart. To minimize this audio/video drift, consider interleaving audio and video in smaller chunk sizes.

# Audio Capture

## In this document

1. [Performing Audio Capture](#)
  1. [Code Example](#)

## Key classes

1. [MediaRecorder](#)

## See also

1. [Android Supported Media Formats](#)
2. [Data Storage](#)
3. [MediaPlayer](#)

The Android multimedia framework includes support for capturing and encoding a variety of common audio formats, so that you can easily integrate audio into your applications. You can record audio using the [MediaRecorder](#) APIs if supported by the device hardware.

This document shows you how to write an application that captures audio from a device microphone, save the audio and play it back.

**Note:** The Android Emulator does not have the ability to capture audio, but actual devices are likely to provide these capabilities.

## Performing Audio Capture

Audio capture from the device is a bit more complicated than audio and video playback, but still fairly simple:

1. Create a new instance of [android.media.MediaRecorder](#).
2. Set the audio source using [MediaRecorder.set AudioSource\(\)](#). You will probably want to use `MediaRecorder.AudioSource.MIC`.
3. Set output file format using [MediaRecorder.setOutputFormat\(\)](#).
4. Set output file name using [MediaRecorder.setOutputFile\(\)](#).
5. Set the audio encoder using [MediaRecorder.setAudioEncoder\(\)](#).
6. Call [MediaRecorder.prepare\(\)](#) on the MediaRecorder instance.
7. To start audio capture, call [MediaRecorder.start\(\)](#).
8. To stop audio capture, call [MediaRecorder.stop\(\)](#).
9. When you are done with the MediaRecorder instance, call [MediaRecorder.release\(\)](#) on it. Calling [MediaRecorder.release\(\)](#) is always recommended to free the resource immediately.

## Example: Record audio and play the recorded audio

The example class below illustrates how to set up, start and stop audio capture, and to play the recorded audio file.

```
/*
```

```
* The application needs to have the permission to write to external storage
```

```
* if the output file is written to the external storage, and also the
* permission to record audio. These permissions must be set in the
* application's AndroidManifest.xml file, with something like:
*
* <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
* <uses-permission android:name="android.permission.RECORD_AUDIO" />
*/
package com.android.audiorecordtest;

import android.app.Activity;
import android.widget.LinearLayout;
import android.os.Bundle;
import android.os.Environment;
import android.view.ViewGroup;
import android.widget.Button;
import android.view.View;
import android.view.View.OnClickListener;
import android.content.Context;
import android.util.Log;
import android.media.MediaRecorder;
import android.media.MediaPlayer;

import java.io.IOException;

public class AudioRecordTest extends Activity
{
    private static final String LOG_TAG = "AudioRecordTest";
    private static String mFileName = null;

    private RecordButton mRecordButton = null;
    private MediaRecorder mRecorder = null;

    private PlayButton mPlayButton = null;
    private MediaPlayer mPlayer = null;

    private void onRecord(boolean start) {
        if (start) {
            startRecording();
        } else {
            stopRecording();
        }
    }

    private void onPlay(boolean start) {
        if (start) {
            startPlaying();
        } else {
            stopPlaying();
        }
    }

    private void startPlaying() {
```

```
mPlayer = new MediaPlayer();
try {
    mPlayer.setDataSource(mFileName);
    mPlayer.prepare();
    mPlayer.start();
} catch (IOException e) {
    Log.e(LOG_TAG, "prepare() failed");
}
}

private void stopPlaying() {
    mPlayer.release();
    mPlayer = null;
}

private void startRecording() {
    mRecorder = new MediaRecorder();
    mRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    mRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    mRecorder.setOutputFile(mFileName);
    mRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);

    try {
        mRecorder.prepare();
    } catch (IOException e) {
        Log.e(LOG_TAG, "prepare() failed");
    }

    mRecorder.start();
}

private void stopRecording() {
    mRecorder.stop();
    mRecorder.release();
    mRecorder = null;
}

class RecordButton extends Button {
    boolean mStartRecording = true;

    OnClickListener clicker = new OnClickListener() {
        public void onClick(View v) {
            onRecord(mStartRecording);
            if (mStartRecording) {
                setText("Stop recording");
            } else {
                setText("Start recording");
            }
            mStartRecording = !mStartRecording;
        }
    };

    public RecordButton(Context ctx) {
        super(ctx);
```

```
        setText("Start recording");
        setOnClickListener(clicker);
    }
}

class PlayButton extends Button {
    boolean mStartPlaying = true;

    OnClickListener clicker = new OnClickListener() {
        public void onClick(View v) {
            onPlay(mStartPlaying);
            if (mStartPlaying) {
                setText("Stop playing");
            } else {
                setText("Start playing");
            }
            mStartPlaying = !mStartPlaying;
        }
    };

    public PlayButton(Context ctx) {
        super(ctx);
        setText("Start playing");
        setOnClickListener(clicker);
    }
}

public AudioRecordTest() {
    mFileName = Environment.getExternalStorageDirectory().getAbsolutePath();
    mFileName += "/audiorecordtest.3gp";
}

@Override
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    LinearLayout ll = new LinearLayout(this);
    mRecordButton = new RecordButton(this);
    ll.addView(mRecordButton,
               new LinearLayout.LayoutParams(
                   ViewGroup.LayoutParams.WRAP_CONTENT,
                   ViewGroup.LayoutParams.WRAP_CONTENT,
                   0));
    mPlayButton = new PlayButton(this);
    ll.addView(mPlayButton,
               new LinearLayout.LayoutParams(
                   ViewGroup.LayoutParams.WRAP_CONTENT,
                   ViewGroup.LayoutParams.WRAP_CONTENT,
                   0));
    setContentView(ll);
}

@Override
public void onPause() {
```

```
super.onPause();
if (mRecorder != null) {
    mRecorder.release();
    mRecorder = null;
}

if (mPlayer != null) {
    mPlayer.release();
    mPlayer = null;
}
}
```

# JetPlayer

## In this document

1. [Playing JET content](#)

## Key classes

1. [JetPlayer](#)

## Related Samples

1. [JetBoy](#)

## See also

1. [JetCreator User Manual](#)
2. [Android Supported Media Formats](#)
3. [Data Storage](#)
4. [MediaPlayer](#)

The Android platform includes a JET engine that lets you add interactive playback of JET audio content in your applications. You can create JET content for interactive playback using the JetCreator authoring application that ships with the SDK. To play and manage JET content from your application, use the [JetPlayer](#) class.

## Playing JET content

This section shows you how to write, set up and play JET content. For a description of JET concepts and instructions on how to use the JetCreator authoring tool, see the [JetCreator User Manual](#). The tool is available on Windows, OS X, and Linux platforms (Linux does not support auditioning of imported assets like with the Windows and OS X versions).

Here's an example of how to set up JET playback from a .jet file stored on the SD card:

```
JetPlayer jetPlayer = JetPlayer.getJetPlayer();
jetPlayer.loadJetFile("/sdcard/level1.jet");
byte segmentId = 0;

// queue segment 5, repeat once, use General MIDI, transpose by -1 octave
jetPlayer.queueJetSegment(5, -1, 1, -1, 0, segmentId++);
// queue segment 2
jetPlayer.queueJetSegment(2, -1, 0, 0, 0, segmentId++);

jetPlayer.play();
```

The SDK includes an example application — JetBoy — that shows how to use [JetPlayer](#) to create an interactive music soundtrack in your game. It also illustrates how to use JET events to synchronize music and game logic. The application is located at [JetBoy](#).

# Camera

## In this document

1. [Considerations](#)
2. [The Basics](#)
3. [Manifest Declarations](#)
4. [Using Existing Camera Apps](#)
  1. [Image capture intent](#)
  2. [Video capture intent](#)
  3. [Receiving camera intent result](#)
5. [Building a Camera App](#)
  1. [Detecting camera hardware](#)
  2. [Accessing cameras](#)
  3. [Checking camera features](#)
  4. [Creating a preview class](#)
  5. [Placing preview in a layout](#)
  6. [Capturing pictures](#)
  7. [Capturing videos](#)
  8. [Releasing the camera](#)
6. [Saving Media Files](#)
7. [Camera Features](#)
  1. [Checking feature availability](#)
  2. [Using camera features](#)
  3. [Metering and focus areas](#)
  4. [Face detection](#)
  5. [Time lapse video](#)

## Key Classes

1. [Camera](#)
2. [SurfaceView](#)
3. [MediaRecorder](#)
4. [Intent](#)

## See also

1. [Media Playback](#)
2. [Data Storage](#)

The Android framework includes support for various cameras and camera features available on devices, allowing you to capture pictures and videos in your applications. This document discusses a quick, simple approach to image and video capture and outlines an advanced approach for creating custom camera experiences for your users.

## Considerations

Before enabling your application to use cameras on Android devices, you should consider a few questions about how your app intends to use this hardware feature.

- **Camera Requirement** - Is the use of a camera so important to your application that you do not want your application installed on a device that does not have a camera? If so, you should declare the [camera requirement in your manifest](#).
- **Quick Picture or Customized Camera** - How will your application use the camera? Are you just interested in snapping a quick picture or video clip, or will your application provide a new way to use cameras? For a getting a quick snap or clip, consider [Using Existing Camera Apps](#). For developing a customized camera feature, check out the [Building a Camera App](#) section.
- **Storage** - Are the images or videos your application generates intended to be only visible to your application or shared so that other applications such as Gallery or other media and social apps can use them? Do you want the pictures and videos to be available even if your application is uninstalled? Check out the [Saving Media Files](#) section to see how to implement these options.

## The Basics

The Android framework supports capturing images and video through the [Camera API](#) or camera [Intent](#). Here are the relevant classes:

### [Camera](#)

This class is the primary API for controlling device cameras. This class is used to take pictures or videos when you are building a camera application.

### [SurfaceView](#)

This class is used to present a live camera preview to the user.

### [MediaRecorder](#)

This class is used to record video from the camera.

### [Intent](#)

An intent action type of [MediaStore.ACTION\\_IMAGE\\_CAPTURE](#) or [MediaStore.ACTION\\_VIDEO\\_CAPTURE](#) can be used to capture images or videos without directly using the [Camera](#) object.

## Manifest Declarations

Before starting development on your application with the Camera API, you should make sure your manifest has the appropriate declarations to allow use of camera hardware and other related features.

- **Camera Permission** - Your application must request permission to use a device camera.

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

**Note:** If you are using the camera [via an intent](#), your application does not need to request this permission.

- **Camera Features** - Your application must also declare use of camera features, for example:

```
<uses-feature android:name="android.hardware.camera" />
```

For a list of camera features, see the manifest [Features Reference](#).

Adding camera features to your manifest causes Google Play to prevent your application from being installed to devices that do not include a camera or do not support the camera features you specify. For

more information about using feature-based filtering with Google Play, see [Google Play and Feature-Based Filtering](#).

If your application *can use* a camera or camera feature for proper operation, but does not *require* it, you should specify this in the manifest by including the `android:required` attribute, and setting it to `false`:

```
<uses-feature android:name="android.hardware.camera" android:required="fa
```

- **Storage Permission** - If your application saves images or videos to the device's external storage (SD Card), you must also specify this in the manifest.

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

- **Audio Recording Permission** - For recording audio with video capture, your application must request the audio capture permission.

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

- **Location Permission** - If your application tags images with GPS location information, you must request location permission:

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

For more information about getting user location, see [Location Strategies](#).

## Using Existing Camera Apps

A quick way to enable taking pictures or videos in your application without a lot of extra code is to use an [Intent](#) to invoke an existing Android camera application. A camera intent makes a request to capture a picture or video clip through an existing camera app and then returns control back to your application. This section shows you how to capture an image or video using this technique.

The procedure for invoking a camera intent follows these general steps:

1. **Compose a Camera Intent** - Create an [Intent](#) that requests an image or video, using one of these intent types:
  - [MediaStore.ACTION\\_IMAGE\\_CAPTURE](#) - Intent action type for requesting an image from an existing camera application.
  - [MediaStore.ACTION\\_VIDEO\\_CAPTURE](#) - Intent action type for requesting a video from an existing camera application.
2. **Start the Camera Intent** - Use the [startActivityForResult\(\)](#) method to execute the camera intent. After you start the intent, the Camera application user interface appears on the device screen and the user can take a picture or video.
3. **Receive the Intent Result** - Set up an [onActivityResult\(\)](#) method in your application to receive the callback and data from the camera intent. When the user finishes taking a picture or video (or cancels the operation), the system calls this method.

### Image capture intent

Capturing images using a camera intent is quick way to enable your application to take pictures with minimal coding. An image capture intent can include the following extra information:

- [MediaStore.EXTRA\\_OUTPUT](#) - This setting requires a [Uri](#) object specifying a path and file name where you'd like to save the picture. This setting is optional but strongly recommended. If you do not specify this value, the camera application saves the requested picture in the default location with a default name, specified in the returned intent's [Intent.getData\(\)](#) field.

The following example demonstrates how to construct a image capture intent and execute it. The `getOutputMediaFileUri()` method in this example refers to the sample code shown in [Saving Media Files](#).

```
private static final int CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE = 100;
private Uri fileUri;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    // create Intent to take a picture and return control to the calling application
    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

    fileUri = getOutputMediaFileUri(MEDIA_TYPE_IMAGE); // create a file to save the image
    intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri); // set the image file name

    // start the image capture Intent
    startActivityForResult(intent, CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE);
}
```

When the `startActivityForResult()` method is executed, users see a camera application interface. After the user finishes taking a picture (or cancels the operation), the user interface returns to your application, and you must intercept the `onActivityResult()` method to receive the result of the intent and continue your application execution. For information on how to receive the completed intent, see [Receiving camera intent result](#).

## Video capture intent

Capturing video using a camera intent is a quick way to enable your application to take videos with minimal coding. A video capture intent can include the following extra information:

- [MediaStore.EXTRA\\_OUTPUT](#) - This setting requires a [Uri](#) specifying a path and file name where you'd like to save the video. This setting is optional but strongly recommended. If you do not specify this value, the Camera application saves the requested video in the default location with a default name, specified in the returned intent's [Intent.getData\(\)](#) field.
- [MediaStore.EXTRA\\_VIDEO\\_QUALITY](#) - This value can be 0 for lowest quality and smallest file size or 1 for highest quality and larger file size.
- [MediaStore.EXTRA\\_DURATION\\_LIMIT](#) - Set this value to limit the length, in seconds, of the video being captured.
- [MediaStore.EXTRA\\_SIZE\\_LIMIT](#) - Set this value to limit the file size, in bytes, of the video being captured.

The following example demonstrates how to construct a video capture intent and execute it. The `getOutputMediaFileUri()` method in this example refers to the sample code shown in [Saving Media Files](#).

```
private static final int CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE = 200;
private Uri fileUri;
```

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    //create new Intent
    Intent intent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);

    fileUri = getOutputMediaFileUri(MEDIA_TYPE_VIDEO); // create a file to save
    intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri); // set the image file m

    intent.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1); // set the video image

    // start the Video Capture Intent
    startActivityForResult(intent, CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE);
}

```

When the [startActivityForResult\(\)](#) method is executed, users see a modified camera application interface. After the user finishes taking a video (or cancels the operation), the user interface returns to your application, and you must intercept the [onActivityResult\(\)](#) method to receive the result of the intent and continue your application execution. For information on how to receive the completed intent, see the next section.

## Receiving camera intent result

Once you have constructed and executed an image or video camera intent, your application must be configured to receive the result of the intent. This section shows you how to intercept the callback from a camera intent so your application can do further processing of the captured image or video.

In order to receive the result of an intent, you must override the [onActivityResult\(\)](#) in the activity that started the intent. The following example demonstrates how to override [onActivityResult\(\)](#) to capture the result of the [image camera intent](#) or [video camera intent](#) examples shown in the previous sections.

```

private static final int CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE = 100;
private static final int CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE = 200;

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            // Image captured and saved to fileUri specified in the Intent
            Toast.makeText(this, "Image saved to:\n" +
                    data.getData(), Toast.LENGTH_LONG).show();
        } else if (resultCode == RESULT_CANCELED) {
            // User cancelled the image capture
        } else {
            // Image capture failed, advise user
        }
    }

    if (requestCode == CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            // Video captured and saved to fileUri specified in the Intent
            Toast.makeText(this, "Video saved to:\n" +
                    data.getData(), Toast.LENGTH_LONG).show();
        }
    }
}

```

```

        } else if (resultCode == RESULT_CANCELED) {
            // User cancelled the video capture
        } else {
            // Video capture failed, advise user
        }
    }
}

```

Once your activity receives a successful result, the captured image or video is available in the specified location for your application to access.

## Building a Camera App

Some developers may require a camera user interface that is customized to the look of their application or provides special features. Creating a customized camera activity requires more code than [using an intent](#), but it can provide a more compelling experience for your users.

The general steps for creating a custom camera interface for your application are as follows:

- **Detect and Access Camera** - Create code to check for the existence of cameras and request access.
- **Create a Preview Class** - Create a camera preview class that extends [SurfaceView](#) and implements the [SurfaceHolder](#) interface. This class previews the live images from the camera.
- **Build a Preview Layout** - Once you have the camera preview class, create a view layout that incorporates the preview and the user interface controls you want.
- **Setup Listeners for Capture** - Connect listeners for your interface controls to start image or video capture in response to user actions, such as pressing a button.
- **Capture and Save Files** - Setup the code for capturing pictures or videos and saving the output.
- **Release the Camera** - After using the camera, your application must properly release it for use by other applications.

Camera hardware is a shared resource that must be carefully managed so your application does not collide with other applications that may also want to use it. The following sections discusses how to detect camera hardware, how to request access to a camera, how to capture pictures or video and how to release the camera when your application is done using it.

**Caution:** Remember to release the [Camera](#) object by calling the [Camera.release\(\)](#) when your application is done using it! If your application does not properly release the camera, all subsequent attempts to access the camera, including those by your own application, will fail and may cause your or other applications to be shut down.

## Detecting camera hardware

If your application does not specifically require a camera using a manifest declaration, you should check to see if a camera is available at runtime. To perform this check, use the [PackageManager.hasSystemFeature\(\)](#) method, as shown in the example code below:

```

/** Check if this device has a camera */
private boolean checkCameraHardware(Context context) {
    if (context.getPackageManager().hasSystemFeature(PackageManager.FEATURE_CAMERA))
        // this device has a camera
        return true;
    } else {
        // no camera on this device
}

```

```
        return false;
    }
}
```

Android devices can have multiple cameras, for example a back-facing camera for photography and a front-facing camera for video calls. Android 2.3 (API Level 9) and later allows you to check the number of cameras available on a device using the [Camera.getNumberOfCameras\(\)](#) method.

## Accessing cameras

If you have determined that the device on which your application is running has a camera, you must request to access it by getting an instance of [Camera](#) (unless you are using an [intent to access the camera](#)).

To access the primary camera, use the [Camera.open\(\)](#) method and be sure to catch any exceptions, as shown in the code below:

```
/** A safe way to get an instance of the Camera object. */
public static Camera getCameraInstance() {
    Camera c = null;
    try {
        c = Camera.open(); // attempt to get a Camera instance
    }
    catch (Exception e) {
        // Camera is not available (in use or does not exist)
    }
    return c; // returns null if camera is unavailable
}
```

**Caution:** Always check for exceptions when using [Camera.open\(\)](#). Failing to check for exceptions if the camera is in use or does not exist will cause your application to be shut down by the system.

On devices running Android 2.3 (API Level 9) or higher, you can access specific cameras using [Camera.open\(int\)](#). The example code above will access the first, back-facing camera on a device with more than one camera.

## Checking camera features

Once you obtain access to a camera, you can get further information about its capabilities using the [Camera.getParameters\(\)](#) method and checking the returned [Camera.Parameters](#) object for supported capabilities. When using API Level 9 or higher, use the [Camera.getCameraInfo\(\)](#) to determine if a camera is on the front or back of the device, and the orientation of the image.

## Creating a preview class

For users to effectively take pictures or video, they must be able to see what the device camera sees. A camera preview class is a [SurfaceView](#) that can display the live image data coming from a camera, so users can frame and capture a picture or video.

The following example code demonstrates how to create a basic camera preview class that can be included in a [View](#) layout. This class implements [SurfaceHolder.Callback](#) in order to capture the callback events for creating and destroying the view, which are needed for assigning the camera preview input.

```
/** A basic Camera preview class */
public class CameraPreview extends SurfaceView implements SurfaceHolder.Callback
```

```
private SurfaceHolder mHolder;
private Camera mCamera;

public CameraPreview(Context context, Camera camera) {
    super(context);
    mCamera = camera;

    // Install a SurfaceHolder.Callback so we get notified when the
    // underlying surface is created and destroyed.
    mHolder = getHolder();
    mHolder.addCallback(this);
    // deprecated setting, but required on Android versions prior to 3.0
    mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
}

public void surfaceCreated(SurfaceHolder holder) {
    // The Surface has been created, now tell the camera where to draw the
    try {
        mCamera.setPreviewDisplay(holder);
        mCamera.startPreview();
    } catch (IOException e) {
        Log.d(TAG, "Error setting camera preview: " + e.getMessage());
    }
}

public void surfaceDestroyed(SurfaceHolder holder) {
    // empty. Take care of releasing the Camera preview in your activity.
}

public void surfaceChanged(SurfaceHolder holder, int format, int w, int h)
    // If your preview can change or rotate, take care of those events here
    // Make sure to stop the preview before resizing or reformatting it.

    if (mHolder.getSurface() == null){
        // preview surface does not exist
        return;
    }

    // stop preview before making changes
    try {
        mCamera.stopPreview();
    } catch (Exception e){
        // ignore: tried to stop a non-existent preview
    }

    // set preview size and make any resize, rotate or
    // reformatting changes here

    // start preview with new settings
    try {
        mCamera.setPreviewDisplay(mHolder);
        mCamera.startPreview();

    } catch (Exception e){
```

```

        Log.d(TAG, "Error starting camera preview: " + e.getMessage());
    }
}

```

If you want to set a specific size for your camera preview, set this in the `surfaceChanged()` method as noted in the comments above. When setting preview size, you *must use* values from `getSupportedPreviewSizes()`. *Do not* set arbitrary values in the `setPreviewSize()` method.

## Placing preview in a layout

A camera preview class, such as the example shown in the previous section, must be placed in the layout of an activity along with other user interface controls for taking a picture or video. This section shows you how to build a basic layout and activity for the preview.

The following layout code provides a very basic view that can be used to display a camera preview. In this example, the `FrameLayout` element is meant to be the container for the camera preview class. This layout type is used so that additional picture information or controls can be overlayed on the live camera preview images.

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <FrameLayout
        android:id="@+id/camera_preview"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_weight="1"
        />
    <Button
        android:id="@+id/button_capture"
        android:text="Capture"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        />
</LinearLayout>

```

On most devices, the default orientation of the camera preview is landscape. This example layout specifies a horizontal (landscape) layout and the code below fixes the orientation of the application to landscape. For simplicity in rendering a camera preview, you should change your application's preview activity orientation to landscape by adding the following to your manifest.

```

<activity android:name=".CameraActivity"
    android:label="@string/app_name"

    android:screenOrientation="landscape">
    <!-- configure this activity to use landscape orientation -->

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

```

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

**Note:** A camera preview does not have to be in landscape mode. Starting in Android 2.2 (API Level 8), you can use the [setDisplayOrientation\(\)](#) method to set the rotation of the preview image. In order to change preview orientation as the user re-orientates the phone, within the [surfaceChanged\(\)](#) method of your preview class, first stop the preview with [Camera.stopPreview\(\)](#), change the orientation and then start the preview again with [Camera.startPreview\(\)](#).

In the activity for your camera view, add your preview class to the [FrameLayout](#) element shown in the example above. Your camera activity must also ensure that it releases the camera when it is paused or shut down. The following example shows how to modify a camera activity to attach the preview class shown in [Creating a preview class](#).

```
public class CameraActivity extends Activity {

    private Camera mCamera;
    private CameraPreview mPreview;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Create an instance of Camera
        mCamera = getCameraInstance();

        // Create our Preview view and set it as the content of our activity.
        mPreview = new CameraPreview(this, mCamera);
        FrameLayout preview = (FrameLayout) findViewById(R.id.camera_preview);
        preview.addView(mPreview);
    }
}
```

**Note:** The [getCameraInstance\(\)](#) method in the example above refers to the example method shown in [Accessing cameras](#).

## Capturing pictures

Once you have built a preview class and a view layout in which to display it, you are ready to start capturing images with your application. In your application code, you must set up listeners for your user interface controls to respond to a user action by taking a picture.

In order to retrieve a picture, use the [Camera.takePicture\(\)](#) method. This method takes three parameters which receive data from the camera. In order to receive data in a JPEG format, you must implement an [Camera.PictureCallback](#) interface to receive the image data and write it to a file. The following code shows a basic implementation of the [Camera.PictureCallback](#) interface to save an image received from the camera.

```
private PictureCallback mPicture = new PictureCallback() {

    @Override
    public void onPictureTaken(byte[] data, Camera camera) {
```

```

        File pictureFile = getOutputMediaFile(MEDIA_TYPE_IMAGE);
        if (pictureFile == null) {
            Log.d(TAG, "Error creating media file, check storage permissions: " +
                    e.getMessage());
            return;
        }

        try {
            FileOutputStream fos = new FileOutputStream(pictureFile);
            fos.write(data);
            fos.close();
        } catch (FileNotFoundException e) {
            Log.d(TAG, "File not found: " + e.getMessage());
        } catch (IOException e) {
            Log.d(TAG, "Error accessing file: " + e.getMessage());
        }
    }
};

}

```

Trigger capturing an image by calling the [Camera.takePicture\(\)](#) method. The following example code shows how to call this method from a button [View.OnClickListener](#).

```

// Add a listener to the Capture button
Button captureButton = (Button) findViewById(id.button_capture);
captureButton.setOnClickListener(
    new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // get an image from the camera
            mCamera.takePicture(null, null, mPicture);
        }
    }
);

```

**Note:** The `mPicture` member in the following example refers to the example code above.

**Caution:** Remember to release the `Camera` object by calling the [Camera.release\(\)](#) when your application is done using it! For information about how to release the camera, see [Releasing the camera](#).

## Capturing videos

Video capture using the Android framework requires careful management of the `Camera` object and coordination with the `MediaRecorder` class. When recording video with `Camera`, you must manage the `Camera.lock()` and `Camera.unlock()` calls to allow `MediaRecorder` access to the camera hardware, in addition to the `Camera.open()` and `Camera.release()` calls.

**Note:** Starting with Android 4.0 (API level 14), the `Camera.lock()` and `Camera.unlock()` calls are managed for you automatically.

Unlike taking pictures with a device camera, capturing video requires a very particular call order. You must follow a specific order of execution to successfully prepare for and capture video with your application, as detailed below.

1. **Open Camera** - Use the [Camera.open\(\)](#) to get an instance of the camera object.
2. **Connect Preview** - Prepare a live camera image preview by connecting a [SurfaceView](#) to the camera using [Camera.setPreviewDisplay\(\)](#).
3. **Start Preview** - Call [Camera.startPreview\(\)](#) to begin displaying the live camera images.
4. **Start Recording Video** - The following steps must be completed *in order* to successfully record video:
  1. **Unlock the Camera** - Unlock the camera for use by [MediaRecorder](#) by calling [Camera.unlock\(\)](#).
  2. **Configure MediaRecorder** - Call in the following [MediaRecorder](#) methods *in this order*. For more information, see the [MediaRecorder](#) reference documentation.
    1. [setCamera\(\)](#) - Set the camera to be used for video capture, use your application's current instance of [Camera](#).
    2. [set AudioSource\(\)](#) - Set the audio source, use [MediaRecorder.AudioSource.CAMCORDER](#).
    3. [set Video Source\(\)](#) - Set the video source, use [MediaRecorder.VideoSource.CAMERA](#).
    4. Set the video output format and encoding. For Android 2.2 (API Level 8) and higher, use the [MediaRecorder.setProfile](#) method, and get a profile instance using [CamcorderProfile.get\(\)](#). For versions of Android prior to 2.2, you must set the video output format and encoding parameters:
      1. [set Output Format\(\)](#) - Set the output format, specify the default setting or [MediaRecorder.OutputFormat.MPEG\\_4](#).
      2. [set Audio Encoder\(\)](#) - Set the sound encoding type, specify the default setting or [MediaRecorder.AudioEncoder.AMR\\_NB](#).
      3. [set Video Encoder\(\)](#) - Set the video encoding type, specify the default setting or [MediaRecorder.VideoEncoder.MPEG\\_4\\_SP](#).
    5. [set Output File\(\)](#) - Set the output file, use `getOutputMedia(MEDIA_TYPE_VIDEO).toString()` from the example method in the [Saving Media Files](#) section.
    6. [set Preview Display\(\)](#) - Specify the [SurfaceView](#) preview layout element for your application. Use the same object you specified for **Connect Preview**.

**Caution:** You must call these [MediaRecorder](#) configuration methods *in this order*, otherwise your application will encounter errors and the recording will fail.

3. **Prepare MediaRecorder** - Prepare the [MediaRecorder](#) with provided configuration settings by calling [MediaRecorder.prepare\(\)](#).
4. **Start MediaRecorder** - Start recording video by calling [MediaRecorder.start\(\)](#).
5. **Stop Recording Video** - Call the following methods *in order*, to successfully complete a video recording:
  1. **Stop MediaRecorder** - Stop recording video by calling [MediaRecorder.stop\(\)](#).
  2. **Reset MediaRecorder** - Optionally, remove the configuration settings from the recorder by calling [MediaRecorder.reset\(\)](#).
  3. **Release MediaRecorder** - Release the [MediaRecorder](#) by calling [MediaRecorder.release\(\)](#).
  4. **Lock the Camera** - Lock the camera so that future [MediaRecorder](#) sessions can use it by calling [Camera.lock\(\)](#). Starting with Android 4.0 (API level 14), this call is not required unless the [MediaRecorder.prepare\(\)](#) call fails.
6. **Stop the Preview** - When your activity has finished using the camera, stop the preview using [Camera.stopPreview\(\)](#).
7. **Release Camera** - Release the camera so that other applications can use it by calling [Camera.release\(\)](#).

**Note:** It is possible to use [MediaRecorder](#) without creating a camera preview first and skip the first few steps of this process. However, since users typically prefer to see a preview before starting a recording, that process is not discussed here.

**Tip:** If your application is typically used for recording video, set [setRecordingHint\(boolean\)](#) to true prior to starting your preview. This setting can help reduce the time it takes to start recording.

## Configuring MediaRecorder

When using the [MediaRecorder](#) class to record video, you must perform configuration steps in a *specific order* and then call the [MediaRecorder.prepare\(\)](#) method to check and implement the configuration. The following example code demonstrates how to properly configure and prepare the [MediaRecorder](#) class for video recording.

```
private boolean prepareVideoRecorder() {  
  
    mCamera = getCameraInstance();  
    mMediaRecorder = new MediaRecorder();  
  
    // Step 1: Unlock and set camera to MediaRecorder  
    mCamera.unlock();  
    mMediaRecorder.setCamera(mCamera);  
  
    // Step 2: Set sources  
    mMediaRecorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);  
    mMediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);  
  
    // Step 3: Set a CamcorderProfile (requires API Level 8 or higher)  
    mMediaRecorder.setProfile(CamcorderProfile.get(CamcorderProfile.QUALITY_HIGH));  
  
    // Step 4: Set output file  
    mMediaRecorder.setOutputFile(getOutputMediaFile(MEDIA_TYPE_VIDEO).toString());  
  
    // Step 5: Set the preview output  
    mMediaRecorder.setPreviewDisplay(mPreview.getHolder().getSurface());  
  
    // Step 6: Prepare configured MediaRecorder  
    try {  
        mMediaRecorder.prepare();  
    } catch (IllegalStateException e) {  
        Log.d(TAG, "IllegalStateException preparing MediaRecorder: " + e.getMessage());  
        releaseMediaRecorder();  
        return false;  
    } catch (IOException e) {  
        Log.d(TAG, "IOException preparing MediaRecorder: " + e.getMessage());  
        releaseMediaRecorder();  
        return false;  
    }  
    return true;  
}
```

Prior to Android 2.2 (API Level 8), you must set the output format and encoding formats parameters directly, instead of using [CamcorderProfile](#). This approach is demonstrated in the following code:

```
// Step 3: Set output format and encoding (for versions prior to API Level  
mMediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);  
mMediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.DEFAULT);  
mMediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.DEFAULT);
```

The following video recording parameters for [MediaRecorder](#) are given default settings, however, you may want to adjust these settings for your application:

- [setVideoEncodingBitRate\(\)](#)
- [setVideoSize\(\)](#)
- [setVideoFrameRate\(\)](#)
- [setAudioEncodingBitRate\(\)](#)
- [setAudioChannels\(\)](#)
- [setAudioSamplingRate\(\)](#)

## Starting and stopping MediaRecorder

When starting and stopping video recording using the [MediaRecorder](#) class, you must follow a specific order, as listed below.

1. Unlock the camera with [Camera.unlock\(\)](#)
2. Configure [MediaRecorder](#) as shown in the code example above
3. Start recording using [MediaRecorder.start\(\)](#)
4. Record the video
5. Stop recording using [MediaRecorder.stop\(\)](#)
6. Release the media recorder with [MediaRecorder.release\(\)](#)
7. Lock the camera using [Camera.lock\(\)](#)

The following example code demonstrates how to wire up a button to properly start and stop video recording using the camera and the [MediaRecorder](#) class.

**Note:** When completing a video recording, do not release the camera or else your preview will be stopped.

```
private boolean isRecording = false;  
  
// Add a listener to the Capture button  
Button captureButton = (Button) findViewById(id.button_capture);  
captureButton.setOnClickListener(  
    new View.OnClickListener() {  
        @Override  
        public void onClick(View v) {  
            if (isRecording) {  
                // stop recording and release camera  
                mMediaRecorder.stop(); // stop the recording  
                releaseMediaRecorder(); // release the MediaRecorder object  
                mCamera.lock(); // take camera access back from MediaRe  
  
                // inform the user that recording has stopped  
                setCaptureButtonText("Capture");  
                isRecording = false;  
            } else {  
                // initialize video camera  
                if (prepareVideoRecorder()) {
```

```

        // Camera is available and unlocked, MediaRecorder is prepared
        // now you can start recording
        mMediaRecorder.start();

        // inform the user that recording has started
        setCaptureButtonText("Stop");
        isRecording = true;
    } else {
        // prepare didn't work, release the camera
        releaseMediaRecorder();
        // inform user
    }
}
}

};


```

**Note:** In the above example, the `prepareVideoRecorder()` method refers to the example code shown in [Configuring MediaRecorder](#). This method takes care of locking the camera, configuring and preparing the [MediaRecorder](#) instance.

## Releasing the camera

Cameras are a resource that is shared by applications on a device. Your application can make use of the camera after getting an instance of [Camera](#), and you must be particularly careful to release the camera object when your application stops using it, and as soon as your application is paused ([Activity.onPause\(\)](#)). If your application does not properly release the camera, all subsequent attempts to access the camera, including those by your own application, will fail and may cause your or other applications to be shut down.

To release an instance of the [Camera](#) object, use the [Camera.release\(\)](#) method, as shown in the example code below.

```

public class CameraActivity extends Activity {
    private Camera mCamera;
    private SurfaceView mPreview;
    private MediaRecorder mMediaRecorder;

    ...

    @Override
    protected void onPause() {
        super.onPause();
        releaseMediaRecorder();           // if you are using MediaRecorder, release
        releaseCamera();                 // release the camera immediately on pause
    }

    private void releaseMediaRecorder() {
        if (mMediaRecorder != null) {
            mMediaRecorder.reset();     // clear recorder configuration
            mMediaRecorder.release();  // release the recorder object
            mMediaRecorder = null;
            mCamera.lock();           // lock camera for later use
        }
    }
}

```

```

private void releaseCamera() {
    if (mCamera != null) {
        mCamera.release(); // release the camera for other applications
        mCamera = null;
    }
}

```

**Caution:** If your application does not properly release the camera, all subsequent attempts to access the camera, including those by your own application, will fail and may cause your or other applications to be shut down.

## Saving Media Files

Media files created by users such as pictures and videos should be saved to a device's external storage directory (SD Card) to conserve system space and to allow users to access these files without their device. There are many possible directory locations to save media files on a device, however there are only two standard locations you should consider as a developer:

- [Environment.getExternalStoragePublicDirectory\(Environment.DIRECTORY\\_PICTURES\)](#) - This method returns the standard, shared and recommended location for saving pictures and videos. This directory is shared (public), so other applications can easily discover, read, change and delete files saved in this location. If your application is uninstalled by the user, media files saved to this location will not be removed. To avoid interfering with users existing pictures and videos, you should create a sub-directory for your application's media files within this directory, as shown in the code sample below. This method is available in Android 2.2 (API Level 8), for equivalent calls in earlier API versions, see [Saving Shared Files](#).
- [Context.getExternalFilesDir\(Environment.DIRECTORY\\_PICTURES\)](#) - This method returns a standard location for saving pictures and videos which are associated with your application. If your application is uninstalled, any files saved in this location are removed. Security is not enforced for files in this location and other applications may read, change and delete them.

The following example code demonstrates how to create a [File](#) or [Uri](#) location for a media file that can be used when invoking a device's camera with an [Intent](#) or as part of a [Building a Camera App](#).

```

public static final int MEDIA_TYPE_IMAGE = 1;
public static final int MEDIA_TYPE_VIDEO = 2;

/** Create a file Uri for saving an image or video */
private static Uri getOutputMediaFileUri(int type) {
    return Uri.fromFile(getOutputMediaFile(type));
}

/** Create a File for saving an image or video */
private static File getOutputMediaFile(int type) {
    // To be safe, you should check that the SDCard is mounted
    // using Environment.getExternalStorageState() before doing this.

    File mediaStorageDir = new File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES), "MyCameraApp");
    // This location works best if you want the created images to be shared
    // between applications and persist after your app has been uninstalled.
}

```

```

// Create the storage directory if it does not exist
if (! mediaStorageDir.exists()){
    if (! mediaStorageDir.mkdirs()){
        Log.d("MyCameraApp", "failed to create directory");
        return null;
    }
}

// Create a media file name
String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date())
File mediaFile;
if (type == MEDIA_TYPE_IMAGE){
    mediaFile = new File(mediaStorageDir.getPath() + File.separator +
    "IMG_"+ timeStamp + ".jpg");
} else if(type == MEDIA_TYPE_VIDEO) {
    mediaFile = new File(mediaStorageDir.getPath() + File.separator +
    "VID_"+ timeStamp + ".mp4");
} else {
    return null;
}

return mediaFile;
}

```

**Note:** [Environment.getExternalStoragePublicDirectory\(\)](#) is available in Android 2.2 (API Level 8) or higher. If you are targeting devices with earlier versions of Android, use [Environment.getExternalStorageDirectory\(\)](#) instead. For more information, see [Saving Shared Files](#).

For more information about saving files on an Android device, see [Data Storage](#).

## Camera Features

Android supports a wide array of camera features you can control with your camera application, such as picture format, flash mode, focus settings, and many more. This section lists the common camera features, and briefly discusses how to use them. Most camera features can be accessed and set using the through [Camera.Parameters](#) object. However, there are several important features that require more than simple settings in [Camera.Parameters](#). These features are covered in the following sections:

- [Metering and focus areas](#)
- [Face detection](#)
- [Time lapse video](#)

For general information about how to use features that are controlled through [Camera.Parameters](#), review the [Using camera features](#) section. For more detailed information about how to use features controlled through the camera parameters object, follow the links in the feature list below to the API reference documentation.

**Table 1.** Common camera features sorted by the Android API Level in which they were introduced.

Feature	API Level	Description
<a href="#">Face Detection</a>	14	Identify human faces within a picture and use them for focus, metering and white balance
<a href="#">Metering Areas</a>	14	Specify one or more areas within an image for calculating white balance

<a href="#">Focus Areas</a>	14	Set one or more areas within an image to use for focus
<a href="#">White Balance Lock</a>	14	Stop or start automatic white balance adjustments
<a href="#">Exposure Lock</a>	14	Stop or start automatic exposure adjustments
<a href="#">Video Snapshot</a>	14	Take a picture while shooting video (frame grab)
<a href="#">Time Lapse Video</a>	11	Record frames with set delays to record a time lapse video
<a href="#">Multiple Cameras</a>	9	Support for more than one camera on a device, including front-facing and back-facing cameras
<a href="#">Focus Distance</a>	9	Reports distances between the camera and objects that appear to be in focus
<a href="#">Zoom</a>	8	Set image magnification
<a href="#">Exposure Compensation</a>	8	Increase or decrease the light exposure level
<a href="#">GPS Data</a>	5	Include or omit geographic location data with the image
<a href="#">White Balance</a>	5	Set the white balance mode, which affects color values in the captured image
<a href="#">Focus Mode</a>	5	Set how the camera focuses on a subject such as automatic, fixed, macro or infinity
<a href="#">Scene Mode</a>	5	Apply a preset mode for specific types of photography situations such as night, beach, snow or candlelight scenes
<a href="#">JPEG Quality</a>	5	Set the compression level for a JPEG image, which increases or decreases image output file quality and size
<a href="#">Flash Mode</a>	5	Turn flash on, off, or use automatic setting
<a href="#">Color Effects</a>	5	Apply a color effect to the captured image such as black and white, sepia tone or negative.
<a href="#">Anti-Banding</a>	5	Reduces the effect of banding in color gradients due to JPEG compression
<a href="#">Picture Format</a>	1	Specify the file format for the picture
<a href="#">Picture Size</a>	1	Specify the pixel dimensions of the saved picture

**Note:** These features are not supported on all devices due to hardware differences and software implementation. For information on checking the availability of features on the device where your application is running, see [Checking feature availability](#).

## Checking feature availability

The first thing to understand when setting out to use camera features on Android devices is that not all camera features are supported on all devices. In addition, devices that support a particular feature may support them to different levels or with different options. Therefore, part of your decision process as you develop a camera application is to decide what camera features you want to support and to what level. After making that decision, you should plan on including code in your camera application that checks to see if device hardware supports those features and fails gracefully if a feature is not available.

You can check the availability of camera features by getting an instance of a camera's parameters object, and checking the relevant methods. The following code sample shows you how to obtain a [Camera.Parameters](#) object and check if the camera supports the autofocus feature:

```
// get Camera parameters
Camera.Parameters params = mCamera.getParameters();

List<String> focusModes = params.getSupportedFocusModes();
if (focusModes.contains(Camera.Parameters.FOCUS_MODE_AUTO)) {
    // Autofocus mode is supported
}
```

You can use the technique shown above for most camera features. The [Camera.Parameters](#) object provides a `getSupported...()`, `is...Supported()` or `getMax...()` method to determine if (and to what extent) a feature is supported.

If your application requires certain camera features in order to function properly, you can require them through additions to your application manifest. When you declare the use of specific camera features, such as flash and auto-focus, Google Play restricts your application from being installed on devices which do not support these features. For a list of camera features that can be declared in your app manifest, see the manifest [Features Reference](#).

## Using camera features

Most camera features are activated and controlled using a [Camera.Parameters](#) object. You obtain this object by first getting an instance of the [Camera](#) object, calling the `getParameters()` method, changing the returned parameter object and then setting it back into the camera object, as demonstrated in the following example code:

```
// get Camera parameters
Camera.Parameters params = mCamera.getParameters();
// set the focus mode
params.setFocusMode(Camera.Parameters.FOCUS_MODE_AUTO);
// set Camera parameters
mCamera.setParameters(params);
```

This technique works for nearly all camera features, and most parameters can be changed at any time after you have obtained an instance of the [Camera](#) object. Changes to parameters are typically visible to the user immediately in the application's camera preview. On the software side, parameter changes may take several frames to actually take effect as the camera hardware processes the new instructions and then sends updated image data.

**Important:** Some camera features cannot be changed at will. In particular, changing the size or orientation of the camera preview requires that you first stop the preview, change the preview size, and then restart the preview. Starting with Android 4.0 (API Level 14) preview orientation can be changed without restarting the preview.

Other camera features require more code in order to implement, including:

- Metering and focus areas
- Face detection
- Time lapse video

A quick outline of how to implement these features is provided in the following sections.

## Metering and focus areas

In some photographic scenarios, automatic focusing and light metering may not produce the desired results. Starting with Android 4.0 (API Level 14), your camera application can provide additional controls to allow your app or users to specify areas in an image to use for determining focus or light level settings and pass these values to the camera hardware for use in capturing images or video.

Areas for metering and focus work very similarly to other camera features, in that you control them through methods in the [Camera.Parameters](#) object. The following code demonstrates setting two light metering areas for an instance of [Camera](#):

```

// Create an instance of Camera
mCamera = getCameraInstance();

// set Camera parameters
Camera.Parameters params = mCamera.getParameters();

if (params.getMaxNumMeteringAreas() > 0){ // check that metering areas are supported
    List<Camera.Area> meteringAreas = new ArrayList<Camera.Area>();

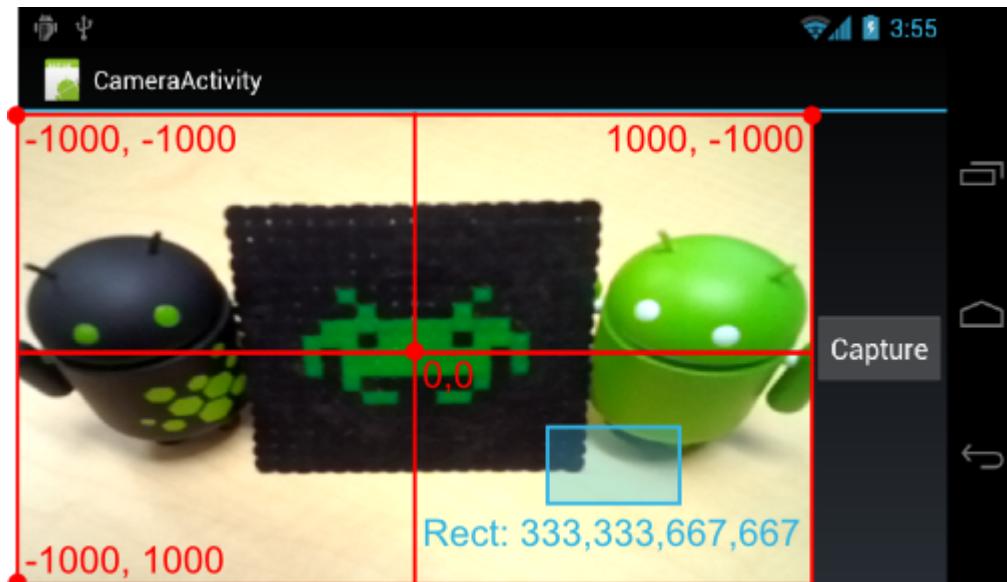
    Rect areaRect1 = new Rect(-100, -100, 100, 100); // specify an area in camera's field of view
    meteringAreas.add(new Camera.Area(areaRect1, 600)); // set weight to 60%
    Rect areaRect2 = new Rect(800, -1000, 1000, -800); // specify an area in camera's field of view
    meteringAreas.add(new Camera.Area(areaRect2, 400)); // set weight to 40%
    params.setMeteringAreas(meteringAreas);
}

mCamera.setParameters(params);

```

The [Camera.Area](#) object contains two data parameters: A [Rect](#) object for specifying an area within the camera's field of view and a weight value, which tells the camera what level of importance this area should be given in light metering or focus calculations.

The [Rect](#) field in a [Camera.Area](#) object describes a rectangular shape mapped on a 2000 x 2000 unit grid. The coordinates -1000, -1000 represent the top, left corner of the camera image, and coordinates 1000, 1000 represent the bottom, right corner of the camera image, as shown in the illustration below.



**Figure 1.** The red lines illustrate the coordinate system for specifying a [Camera.Area](#) within a camera preview. The blue box shows the location and shape of an camera area with the [Rect](#) values 333,333,667,667.

The bounds of this coordinate system always correspond to the outer edge of the image visible in the camera preview and do not shrink or expand with the zoom level. Similarly, rotation of the image preview using [Camera.setDisplayOrientation\(\)](#) does not remap the coordinate system.

## Face detection

For pictures that include people, faces are usually the most important part of the picture, and should be used for determining both focus and white balance when capturing an image. The Android 4.0 (API Level 14) framework provides APIs for identifying faces and calculating picture settings using face recognition technology.

**Note:** While the face detection feature is running, [setWhiteBalance \(String\)](#), [setFocusAreas \(List\)](#) and [setMeteringAreas \(List\)](#) have no effect.

Using the face detection feature in your camera application requires a few general steps:

- Check that face detection is supported on the device
- Create a face detection listener
- Add the face detection listener to your camera object
- Start face detection after preview (and after *every* preview restart)

The face detection feature is not supported on all devices. You can check that this feature is supported by calling [getMaxNumDetectedFaces \(\)](#). An example of this check is shown in the [startFaceDetection \(\)](#) sample method below.

In order to be notified and respond to the detection of a face, your camera application must set a listener for face detection events. In order to do this, you must create a listener class that implements the [Camera.FaceDetectionListener](#) interface as shown in the example code below.

```
class MyFaceDetectionListener implements Camera.FaceDetectionListener {  
  
    @Override  
    public void onFaceDetection(Face[] faces, Camera camera) {  
        if (faces.length > 0){  
            Log.d("FaceDetection", "face detected: "+ faces.length +  
                  " Face 1 Location X: " + faces[0].rect.centerX() +  
                  "Y: " + faces[0].rect.centerY());  
        }  
    }  
}
```

After creating this class, you then set it into your application's [Camera](#) object, as shown in the example code below:

```
mCamera.setFaceDetectionListener(new MyFaceDetectionListener());
```

Your application must start the face detection function each time you start (or restart) the camera preview. Create a method for starting face detection so you can call it as needed, as shown in the example code below.

```
public void startFaceDetection(){  
    // Try starting Face Detection  
    Camera.Parameters params = mCamera.getParameters();  
  
    // start face detection only *after* preview has started  
    if (params.getMaxNumDetectedFaces() > 0){  
        // camera supports face detection, so can start it:  
        mCamera.startFaceDetection();  
    }  
}
```

You must start face detection *each time* you start (or restart) the camera preview. If you use the preview class shown in [Creating a preview class](#), add your `startFaceDetection()` method to both the `surfaceCreated()` and `surfaceChanged()` methods in your preview class, as shown in the sample code below.

```
public void surfaceCreated(SurfaceHolder holder) {
    try {
        mCamera.setPreviewDisplay(holder);
        mCamera.startPreview();

        startFaceDetection(); // start face detection feature
    } catch (IOException e) {
        Log.d(TAG, "Error setting camera preview: " + e.getMessage());
    }
}

public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {

    if (mHolder.getSurface() == null) {
        // preview surface does not exist
        Log.d(TAG, "mHolder.getSurface() == null");
        return;
    }

    try {
        mCamera.stopPreview();

    } catch (Exception e) {
        // ignore: tried to stop a non-existent preview
        Log.d(TAG, "Error stopping camera preview: " + e.getMessage());
    }

    try {
        mCamera.setPreviewDisplay(mHolder);
        mCamera.startPreview();

        startFaceDetection(); // re-start face detection feature
    } catch (Exception e) {
        // ignore: tried to stop a non-existent preview
        Log.d(TAG, "Error starting camera preview: " + e.getMessage());
    }
}
```

**Note:** Remember to call this method *after* calling `startPreview()`. Do not attempt to start face detection in the `onCreate()` method of your camera app's main activity, as the preview is not available by this point in your application's the execution.

## Time lapse video

Time lapse video allows users to create video clips that combine pictures taken a few seconds or minutes apart. This feature uses [MediaRecorder](#) to record the images for a time lapse sequence.

To record a time lapse video with [MediaRecorder](#), you must configure the recorder object as if you are recording a normal video, setting the captured frames per second to a low number and using one of the time lapse quality settings, as shown in the code example below.

```
// Step 3: Set a CamcorderProfile (requires API Level 8 or higher)
mMediaRecorder.setProfile(CamcorderProfile.get(CamcorderProfile.QUALITY_TIME_LAPSE));
...
// Step 5.5: Set the video capture rate to a low number
mMediaRecorder.setCaptureRate(0.1); // capture a frame every 10 seconds
```

These settings must be done as part of a larger configuration procedure for [MediaRecorder](#). For a full configuration code example, see [Configuring MediaRecorder](#). Once the configuration is complete, you start the video recording as if you were recording a normal video clip. For more information about configuring and running [MediaRecorder](#), see [Capturing videos](#).

# Location and Sensors APIs

Use sensors on the device to add rich location and motion capabilities to your app, from GPS or network location to accelerometer, gyroscope, temperature, barometer, and more.

## Blog Articles

### [One Screen Turn Deserves Another](#)

However, there's a new wrinkle: recently, a few devices have shipped (see here and here) that run Android on screens that are naturally landscape in their orientation. That is, when held in the default position, the screens are wider than they are tall. This introduces a few fairly subtle issues that we've noticed causing problems in some apps.

### [A Deep Dive Into Location](#)

I've written an open-source reference app that incorporates all of the tips, tricks, and cheats I know to reduce the time between opening an app and seeing an up-to-date list of nearby venues - as well as providing a reasonable level of offline support

## Training

### [Making Your App Location Aware](#)

This class teaches you how to incorporate location based services in your Android application. You'll learn a number of methods to receive location updates and related best practices.

# Location and Maps

## In this document

1. [Location Services](#)
2. [Google Maps Android API](#)

**Note:** This is a guide to the *Android framework* location APIs in the package `android.location`. The Google Location Services API, part of Google Play Services, provides a more powerful, high-level framework that automates tasks such as location provider choice and power management. Location Services also provides new features such as activity detection that aren't available in the framework API. Developers who are using the framework API, as well as developers who are just now adding location-awareness to their apps, should strongly consider using the Location Services API.

To learn more about the Location Services API, see [Google Location Services for Android](#).

Location and maps-based apps offer a compelling experience on mobile devices. You can build these capabilities into your app using the classes of the `android.location` package and the Google Maps Android API. The sections below provide an introduction to how you can add the features.

## Location Services

Android gives your applications access to the location services supported by the device through classes in the `android.location` package. The central component of the location framework is the [LocationManager](#) system service, which provides APIs to determine location and bearing of the underlying device (if available).

As with other system services, you do not instantiate a [LocationManager](#) directly. Rather, you request an instance from the system by calling `getSystemService(Context.LOCATION_SERVICE)`. The method returns a handle to a new [LocationManager](#) instance.

Once your application has a [LocationManager](#), your application is able to do three things:

- Query for the list of all [LocationProviders](#) for the last known user location.
- Register/unregister for periodic updates of the user's current location from a location provider (specified either by criteria or name).
- Register/unregister for a given [Intent](#) to be fired if the device comes within a given proximity (specified by radius in meters) of a given lat/long.

For more information about acquiring the user location, read the [Location Strategies](#) guide.

## Google Maps Android API

With the [Google Maps Android API](#), you can add maps to your app that are based on Google Maps data. The API automatically handles access to Google Maps servers, data downloading, map display, and touch gestures on the map. You can also use API calls to add markers, polygons and overlays, and to change the user's view of a particular map area.

The key class in the Google Maps Android API is [MapView](#). A [MapView](#) displays a map with data obtained from the Google Maps service. When the [MapView](#) has focus, it will capture keypresses and touch gestures to pan and zoom the map automatically, including handling network requests for additional maps tiles. It also pro-

vides all of the UI elements necessary for users to control the map. Your application can also use [MapView](#) class methods to control the map programmatically and draw a number of overlays on top of the map.

The Google Maps Android APIs are not included in the Android platform, but are available on any device with the Google Play Store running Android 2.2 or higher, through [Google Play services](#).

To integrate Google Maps into your app, you need to install the Google Play services libraries for your Android SDK. For more details, read about [Google Play services](#).

# Location Strategies

## In this document

1. [Challenges in Determining User Location](#)
2. [Requesting Location Updates](#)
  1. [Requesting User Permissions](#)
3. [Defining a Model for the Best Performance](#)
  1. [Flow for obtaining user location](#)
  2. [Deciding when to start listening for updates](#)
  3. [Getting a fast fix with the last known location](#)
  4. [Deciding when to stop listening for updates](#)
  5. [Maintaining a current best estimate](#)
  6. [Adjusting the model to save battery and data exchange](#)
4. [Providing Mock Location Data](#)

## Key classes

1. [LocationManager](#)
2. [LocationListener](#)

**Note:** The strategies described in this guide apply to the platform location API in [android.location](#). The Google Location Services API, part of Google Play Services, provides a more powerful, high-level framework that automatically handles location providers, user movement, and location accuracy. It also handles location update scheduling based on power consumption parameters you provide. In most cases, you'll get better battery performance, as well as more appropriate accuracy, by using the Location Services API.

To learn more about the Location Services API, see [Google Location Services for Android](#).

Knowing where the user is allows your application to be smarter and deliver better information to the user. When developing a location-aware application for Android, you can utilize GPS and Android's Network Location Provider to acquire the user location. Although GPS is most accurate, it only works outdoors, it quickly consumes battery power, and doesn't return the location as quickly as users want. Android's Network Location Provider determines user location using cell tower and Wi-Fi signals, providing location information in a way that works indoors and outdoors, responds faster, and uses less battery power. To obtain the user location in your application, you can use both GPS and the Network Location Provider, or just one.

## Challenges in Determining User Location

Obtaining user location from a mobile device can be complicated. There are several reasons why a location reading (regardless of the source) can contain errors and be inaccurate. Some sources of error in the user location include:

- **Multitude of location sources**

GPS, Cell-ID, and Wi-Fi can each provide a clue to users location. Determining which to use and trust is a matter of trade-offs in accuracy, speed, and battery-efficiency.

- **User movement**

Because the user location changes, you must account for movement by re-estimating user location every so often.

- **Varying accuracy**

Location estimates coming from each location source are not consistent in their accuracy. A location obtained 10 seconds ago from one source might be more accurate than the newest location from another or same source.

These problems can make it difficult to obtain a reliable user location reading. This document provides information to help you meet these challenges to obtain a reliable location reading. It also provides ideas that you can use in your application to provide the user with an accurate and responsive geo-location experience.

## Requesting Location Updates

Before addressing some of the location errors described above, here is an introduction to how you can obtain user location on Android.

Getting user location in Android works by means of callback. You indicate that you'd like to receive location updates from the [LocationManager](#) ("Location Manager") by calling [requestLocationUpdates\(\)](#), passing it a [LocationListener](#). Your [LocationListener](#) must implement several callback methods that the Location Manager calls when the user location changes or when the status of the service changes.

For example, the following code shows how to define a [LocationListener](#) and request location updates:

```
// Acquire a reference to the system Location Manager
LocationManager locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);

// Define a listener that responds to location updates
LocationListener locationListener = new LocationListener() {
    public void onLocationChanged(Location location) {
        // Called when a new location is found by the network location provider.
        makeUseOfNewLocation(location);
    }

    public void onStatusChanged(String provider, int status, Bundle extras) {}

    public void onProviderEnabled(String provider) {}

    public void onProviderDisabled(String provider) {}
};

// Register the listener with the Location Manager to receive location updates
locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0, 0,
```

The first parameter in [requestLocationUpdates\(\)](#) is the type of location provider to use (in this case, the Network Location Provider for cell tower and Wi-Fi based location). You can control the frequency at which your listener receives updates with the second and third parameter—the second is the minimum time interval between notifications and the third is the minimum change in distance between notifications—setting both to zero requests location notifications as frequently as possible. The last parameter is your [LocationListener](#), which receives callbacks for location updates.

To request location updates from the GPS provider, substitute `GPS_PROVIDER` for `NETWORK_PROVIDER`. You can also request location updates from both the GPS and the Network Location Provider by calling [requestLocationUpdates\(\)](#) twice—once for `NETWORK_PROVIDER` and once for `GPS_PROVIDER`.

## Requesting User Permissions

In order to receive location updates from `NETWORK_PROVIDER` or `GPS_PROVIDER`, you must request user permission by declaring either the `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permission, respectively, in your Android manifest file. For example:

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

Without these permissions, your application will fail at runtime when requesting location updates.

**Note:** If you are using both `NETWORK_PROVIDER` and `GPS_PROVIDER`, then you need to request only the `ACCESS_FINE_LOCATION` permission, because it includes permission for both providers. (Permission for `ACCESS_COARSE_LOCATION` includes permission only for `NETWORK_PROVIDER`.)

## Defining a Model for the Best Performance

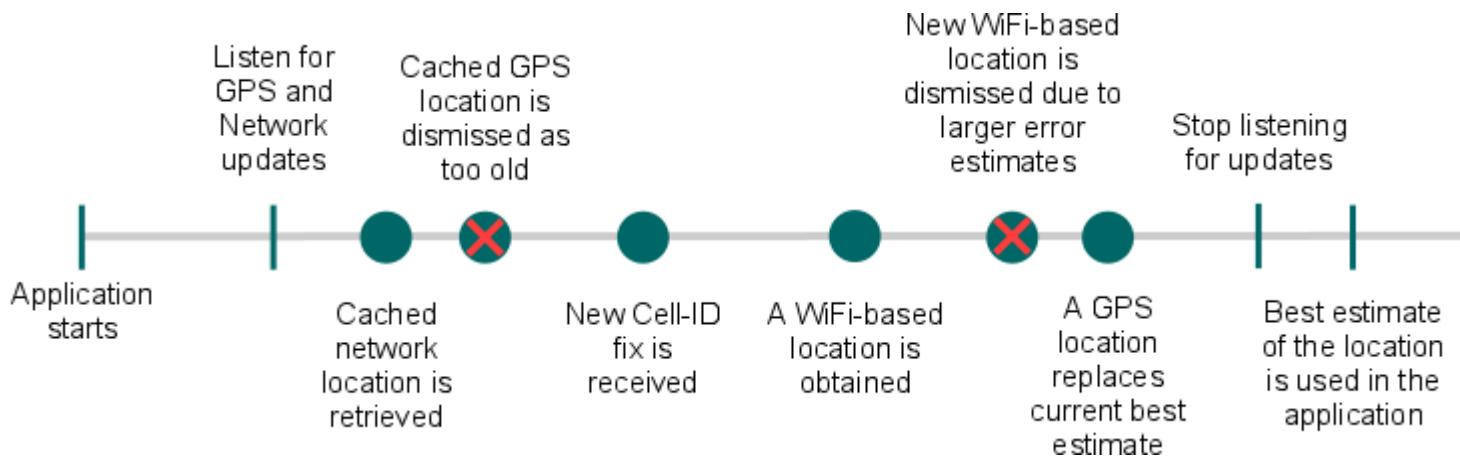
Location-based applications are now commonplace, but due to the less than optimal accuracy, user movement, the multitude of methods to obtain the location, and the desire to conserve battery, getting user location is complicated. To overcome the obstacles of obtaining a good user location while preserving battery power, you must define a consistent model that specifies how your application obtains the user location. This model includes when you start and stop listening for updates and when to use cached location data.

### Flow for obtaining user location

Here's the typical flow of procedures for obtaining the user location:

1. Start application.
2. Sometime later, start listening for updates from desired location providers.
3. Maintain a "current best estimate" of location by filtering out new, but less accurate fixes.
4. Stop listening for location updates.
5. Take advantage of the last best location estimate.

Figure 1 demonstrates this model in a timeline that visualizes the period in which an application is listening for location updates and the events that occur during that time.



**Figure 1.** A timeline representing the window in which an application listens for location updates.

This model of a window—during which location updates are received—frames many of the decisions you need to make when adding location-based services to your application.

## Deciding when to start listening for updates

You might want to start listening for location updates as soon as your application starts, or only after users activate a certain feature. Be aware that long windows of listening for location fixes can consume a lot of battery power, but short periods might not allow for sufficient accuracy.

As demonstrated above, you can begin listening for updates by calling [requestLocationUpdates\(\)](#):

```
String locationProvider = LocationManager.NETWORK_PROVIDER;
// Or, use GPS location data:
// String locationProvider = LocationManager.GPS_PROVIDER;

locationManager.requestLocationUpdates(locationProvider, 0, 0, locationListener);
```

## Getting a fast fix with the last known location

The time it takes for your location listener to receive the first location fix is often too long for users wait. Until a more accurate location is provided to your location listener, you should utilize a cached location by calling [getLastKnownLocation\(String\)](#):

```
String locationProvider = LocationManager.NETWORK_PROVIDER;
// Or use LocationManager.GPS_PROVIDER

Location lastKnownLocation = locationManager.getLastKnownLocation(locationProvider);
```

## Deciding when to stop listening for updates

The logic of deciding when new fixes are no longer necessary might range from very simple to very complex depending on your application. A short gap between when the location is acquired and when the location is used, improves the accuracy of the estimate. Always beware that listening for a long time consumes a lot of battery power, so as soon as you have the information you need, you should stop listening for updates by calling [removeUpdates\(PendingIntent\)](#):

```
// Remove the listener you previously added
locationManager.removeUpdates(locationListener);
```

## Maintaining a current best estimate

You might expect that the most recent location fix is the most accurate. However, because the accuracy of a location fix varies, the most recent fix is not always the best. You should include logic for choosing location fixes based on several criteria. The criteria also varies depending on the use-cases of the application and field testing.

Here are a few steps you can take to validate the accuracy of a location fix:

- Check if the location retrieved is significantly newer than the previous estimate.
- Check if the accuracy claimed by the location is better or worse than the previous estimate.
- Check which provider the new location is from and determine if you trust it more.

An elaborate example of this logic can look something like this:

```
private static final int TWO_MINUTES = 1000 * 60 * 2;

/** Determines whether one Location reading is better than the current Location
 * @param location The new Location that you want to evaluate
 * @param currentBestLocation The current Location fix, to which you want to
 */
protected boolean isBetterLocation(Location location, Location currentBestLocation) {
    if (currentBestLocation == null) {
        // A new location is always better than no location
        return true;
    }

    // Check whether the new location fix is newer or older
    long timeDelta = location.getTime() - currentBestLocation.getTime();
    boolean isSignificantlyNewer = timeDelta > TWO_MINUTES;
    boolean isSignificantlyOlder = timeDelta < -TWO_MINUTES;
    boolean isNewer = timeDelta > 0;

    // If it's been more than two minutes since the current location, use the new
    // location because the user has likely moved
    if (isSignificantlyNewer) {
        return true;
    } else if (isSignificantlyOlder) {
        return false;
    }

    // Check whether the new location fix is more or less accurate
    int accuracyDelta = (int) (location.getAccuracy() - currentBestLocation.getAccuracy());
    boolean isLessAccurate = accuracyDelta > 0;
    boolean isMoreAccurate = accuracyDelta < 0;
    boolean isSignificantlyLessAccurate = accuracyDelta > 200;

    // Check if the old and new location are from the same provider
    boolean isFromSameProvider = isSameProvider(location.getProvider(),
                                                 currentBestLocation.getProvider());

    // Determine location quality using a combination of timeliness and accuracy
    if (isMoreAccurate) {
        return true;
    } else if (isSignificantlyNewer) {
        return true;
    } else if (isFromSameProvider) {
        return true;
    }
}
```

```

        } else if (isNewer && !isLessAccurate) {
            return true;
        } else if (isNewer && !isSignificantlyLessAccurate && isFromSameProvider) {
            return true;
        }
        return false;
    }

/** Checks whether two providers are the same */
private boolean isSameProvider(String provider1, String provider2) {
    if (provider1 == null) {
        return provider2 == null;
    }
    return provider1.equals(provider2);
}

```

## Adjusting the model to save battery and data exchange

As you test your application, you might find that your model for providing good location and good performance needs some adjustment. Here are some things you might change to find a good balance between the two.

### Reduce the size of the window

A smaller window in which you listen for location updates means less interaction with GPS and network location services, thus, preserving battery life. But it also allows for fewer locations from which to choose a best estimate.

### Set the location providers to return updates less frequently

Reducing the rate at which new updates appear during the window can also improve battery efficiency, but at the cost of accuracy. The value of the trade-off depends on how your application is used. You can reduce the rate of updates by increasing the parameters in [requestLocationUpdates\(\)](#) that specify the interval time and minimum distance change.

### Restrict a set of providers

Depending on the environment where your application is used or the desired level of accuracy, you might choose to use only the Network Location Provider or only GPS, instead of both. Interacting with only one of the services reduces battery usage at a potential cost of accuracy.

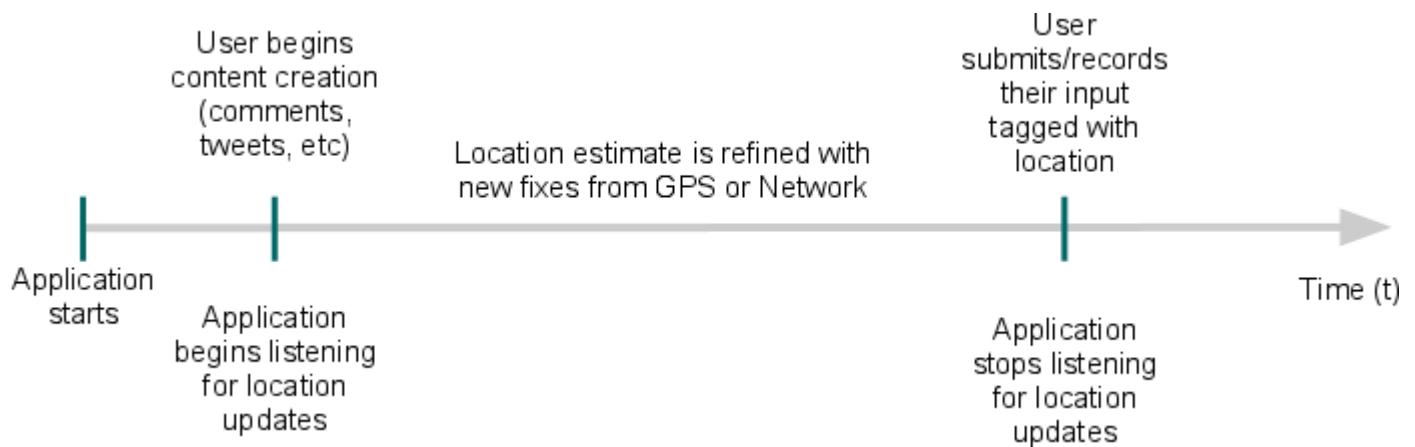
## Common application cases

There are many reasons you might want to obtain the user location in your application. Below are a couple scenarios in which you can use the user location to enrich your application. Each scenario also describes good practices for when you should start and stop listening for the location, in order to get a good reading and help preserve battery life.

### Tagging user-created content with a location

You might be creating an application where user-created content is tagged with a location. Think of users sharing their local experiences, posting a review for a restaurant, or recording some content that can be augmented

with their current location. A model of how this interaction might happen, with respect to the location services, is visualized in figure 2.



**Figure 2.** A timeline representing the window in which the user location is obtained and listening stops when the user consumes the current location.

This lines up with the previous model of how user location is obtained in code (figure 1). For best location accuracy, you might choose to start listening for location updates when users begin creating the content or even when the application starts, then stop listening for updates when content is ready to be posted or recorded. You might need to consider how long a typical task of creating the content takes and judge if this duration allows for efficient collection of a location estimate.

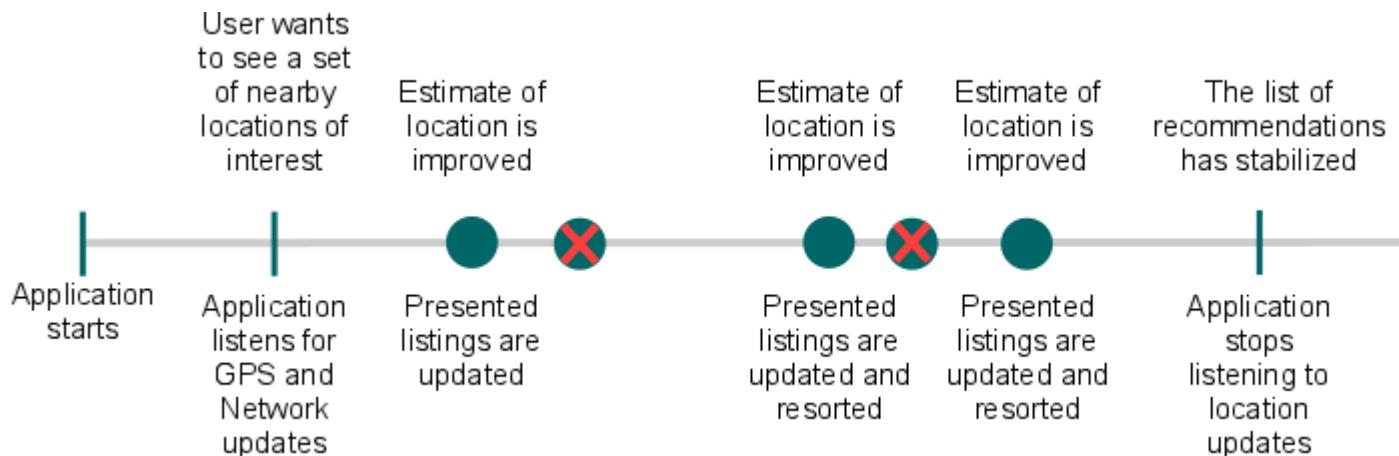
## Helping the user decide on where to go

You might be creating an application that attempts to provide users with a set of options about where to go. For example, you're looking to provide a list of nearby restaurants, stores, and entertainment and the order of recommendations changes depending on the user location.

To accommodate such a flow, you might choose to:

- Rearrange recommendations when a new best estimate is obtained
- Stop listening for updates if the order of recommendations has stabilized

This kind of model is visualized in figure 3.



**Figure 3.** A timeline representing the window in which a dynamic set of data is updated each time the user location updates.

# Providing Mock Location Data

As you develop your application, you'll certainly need to test how well your model for obtaining user location works. This is most easily done using a real Android-powered device. If, however, you don't have a device, you can still test your location-based features by mocking location data in the Android emulator. There are three different ways to send your application mock location data: using Eclipse, DDMS, or the "geo" command in the emulator console.

**Note:** Providing mock location data is injected as GPS location data, so you must request location updates from `GPS_PROVIDER` in order for mock location data to work.

## Using Eclipse

Select **Window > Show View > Other > Emulator Control**.

In the Emulator Control panel, enter GPS coordinates under Location Controls as individual lat/long coordinates, with a GPX file for route playback, or a KML file for multiple place marks. (Be sure that you have a device selected in the Devices panel—available from **Window > Show View > Other > Devices**.)

## Using DDMS

With the DDMS tool, you can simulate location data a few different ways:

- Manually send individual longitude/latitude coordinates to the device.
- Use a GPX file describing a route for playback to the device.
- Use a KML file describing individual place marks for sequenced playback to the device.

For more information on using DDMS to spoof location data, see [Using DDMS](#).

## Using the "geo" command in the emulator console

To send mock location data from the command line:

1. Launch your application in the Android emulator and open a terminal/console in your SDK's `/tools` directory.
2. Connect to the emulator console:

```
telnet localhost <console-port>
```

3. Send the location data:

- `geo fix` to send a fixed geo-location.

This command accepts a longitude and latitude in decimal degrees, and an optional altitude in meters. For example:

```
geo fix -121.45356 46.51119 4392
```

- `geo nmea` to send an NMEA 0183 sentence.

This command accepts a single NMEA sentence of type '\$GPGGA' (fix data) or '\$GPRMC' (transit data). For example:

```
geo nmea $GPRMC,081836,A,3751.65,S,14507.36,E,000.0,360.0,130998,011
```

For information about how to connect to the emulator console, see [Using the Emulator Console](#).

# Sensors Overview

## Quickview

- Learn about the sensors that Android supports and the Android sensor framework.
- Find out how to list sensors, determine sensor capabilities, and monitor sensor data.
- Learn about best practices for accessing and using sensors.

## In this document

1. [Introduction to Sensors](#)
2. [Identifying Sensors and Sensor Capabilities](#)
3. [Monitoring Sensor Events](#)
4. [Handling Different Sensor Configurations](#)
5. [Sensor Coordinate System](#)
6. [Best Practices for Accessing and Using Sensors](#)

## Key classes and interfaces

1. [Sensor](#)
2. [SensorEvent](#)
3. [SensorManager](#)
4. [SensorEventListener](#)

## Related samples

1. [Accelerometer Play](#)
2. [API Demos \(OS - RotationVectorDemo\)](#)
3. [API Demos \(OS - Sensors\)](#)

## See also

1. [Sensors](#)
2. [Motion Sensors](#)
3. [Position Sensors](#)
4. [Environment Sensors](#)

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device. For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dewpoint, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

The Android platform supports three broad categories of sensors:

- Motion sensors

These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.

- Environmental sensors

These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.

- Position sensors

These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

You can access sensors available on the device and acquire raw sensor data by using the Android sensor framework. The sensor framework provides several classes and interfaces that help you perform a wide variety of sensor-related tasks. For example, you can use the sensor framework to do the following:

- Determine which sensors are available on a device.
- Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.
- Acquire raw sensor data and define the minimum rate at which you acquire sensor data.
- Register and unregister sensor event listeners that monitor sensor changes.

This topic provides an overview of the sensors that are available on the Android platform. It also provides an introduction to the sensor framework.

## Introduction to Sensors

The Android sensor framework lets you access many types of sensors. Some of these sensors are hardware-based and some are software-based. Hardware-based sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change. Software-based sensors are not physical devices, although they mimic hardware-based sensors. Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called virtual sensors or synthetic sensors. The linear acceleration sensor and the gravity sensor are examples of software-based sensors. Table 1 summarizes the sensors that are supported by the Android platform.

Few Android-powered devices have every type of sensor. For example, most handset devices and tablets have an accelerometer and a magnetometer, but fewer devices have barometers or thermometers. Also, a device can have more than one sensor of a given type. For example, a device can have two gravity sensors, each one having a different range.

**Table 1.** Sensor types supported by the Android platform.

Sensor	Type	Description	Common Uses
<a href="#">TYPE_ACCELEROMETER</a>	Hardware	Measures the acceleration force in $\text{m/s}^2$ that is applied to a device on all three physical axes (x, y, and z), including the force of gravity.	Motion detection (shake, tilt, etc.).
<a href="#">TYPE_AMBIENT_TEMPERATURE</a>	Hardware	Measures the ambient room temperature in degrees Celsius ( $^\circ\text{C}$ ). See note below.	Monitoring air temperatures.

<a href="#">TYPE_GRAVITY</a>	Software or Hardware	Measures the force of gravity in $\text{m/s}^2$ that is applied to a device on all three physical axes (x, y, z). (shake, tilt, etc.).	Motion detection
<a href="#">TYPE_GYROSCOPE</a>	Hardware	Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z).	Rotation detection (spin, turn, etc.).
<a href="#">TYPE_LIGHT</a>	Hardware	Measures the ambient light level (illumination) in lx.	Controlling screen brightness.
<a href="#">TYPE_LINEAR_ACCELERATION</a>	Software or Hardware	Measures the acceleration force in $\text{m/s}^2$ that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity.	Monitoring acceleration along a single axis.
<a href="#">TYPE_MAGNETIC_FIELD</a>	Hardware	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in $\mu\text{T}$ .	Creating a compass.
<a href="#">TYPE_ORIENTATION</a>	Software	Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity sensor and the geomagnetic field sensor in conjunction with the <a href="#">getRotationMatrix()</a> method.	Determining device position.
<a href="#">TYPE_PRESSURE</a>	Hardware	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
<a href="#">TYPE_PROXIMITY</a>	Hardware	Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	Phone position during a call.
<a href="#">TYPE_RELATIVE_HUMIDITY</a>	Hardware	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative humidity.
<a href="#">TYPE_ROTATION_VECTOR</a>	Software or Hardware	Measures the orientation of a device by providing the three elements of the device's rotation vector.	Motion detection and rotation detection.
<a href="#">TYPE_TEMPERATURE</a>	Hardware	Measures the temperature of the device in degrees Celsius ( $^\circ\text{C}$ ). This sensor implementation varies across devices and this sensor was replaced with the <a href="#">TYPE_AMBIENT_TEMPERATURE</a> sensor in API Level 14	Monitoring temperatures.

## Sensor Framework

You can access these sensors and acquire raw sensor data by using the Android sensor framework. The sensor framework is part of the [android.hardware](#) package and includes the following classes and interfaces:

## SensorManager

You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

## Sensor

You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

## SensorEvent

The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

## SensorEventListener

You can use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

In a typical application you use these sensor-related APIs to perform two basic tasks:

- **Identifying sensors and sensor capabilities**

Identifying sensors and sensor capabilities at runtime is useful if your application has features that rely on specific sensor types or capabilities. For example, you may want to identify all of the sensors that are present on a device and disable any application features that rely on sensors that are not present. Likewise, you may want to identify all of the sensors of a given type so you can choose the sensor implementation that has the optimum performance for your application.

- **Monitor sensor events**

Monitoring sensor events is how you acquire raw sensor data. A sensor event occurs every time a sensor detects a change in the parameters it is measuring. A sensor event provides you with four pieces of information: the name of the sensor that triggered the event, the timestamp for the event, the accuracy of the event, and the raw sensor data that triggered the event.

## Sensor Availability

While sensor availability varies from device to device, it can also vary between Android versions. This is because the Android sensors have been introduced over the course of several platform releases. For example, many sensors were introduced in Android 1.5 (API Level 3), but some were not implemented and were not available for use until Android 2.3 (API Level 9). Likewise, several sensors were introduced in Android 2.3 (API Level 9) and Android 4.0 (API Level 14). Two sensors have been deprecated and replaced by newer, better sensors.

Table 2 summarizes the availability of each sensor on a platform-by-platform basis. Only four platforms are listed because those are the platforms that involved sensor changes. Sensors that are listed as deprecated are still available on subsequent platforms (provided the sensor is present on a device), which is in line with Android's forward compatibility policy.

**Table 2.** Sensor availability by platform.

Sensor	Android 4.0 (API Level 14)	Android 2.3 (API Level 9)	Android 2.2 (API Level 8)	Android 1.5 (API Level 3)
--------	-------------------------------	------------------------------	------------------------------	------------------------------

<a href="#">TYPE_ACCELEROMETER</a>	Yes	Yes	Yes	Yes
<a href="#">TYPE_AMBIENT_TEMPERATURE</a>	Yes	n/a	n/a	n/a
<a href="#">TYPE_GRAVITY</a>	Yes	Yes	n/a	n/a
<a href="#">TYPE_GYROSCOPE</a>	Yes	Yes	n/a <sup>1</sup>	n/a <sup>1</sup>
<a href="#">TYPE_LIGHT</a>	Yes	Yes	Yes	Yes
<a href="#">TYPE_LINEAR_ACCELERATION</a>	Yes	Yes	n/a	n/a
<a href="#">TYPE_MAGNETIC_FIELD</a>	Yes	Yes	Yes	Yes
<a href="#">TYPE_ORIENTATION</a>	Yes <sup>2</sup>	Yes <sup>2</sup>	Yes <sup>2</sup>	Yes
<a href="#">TYPE_PRESSURE</a>	Yes	Yes	n/a <sup>1</sup>	n/a <sup>1</sup>
<a href="#">TYPE_PROXIMITY</a>	Yes	Yes	Yes	Yes
<a href="#">TYPE_RELATIVE_HUMIDITY</a>	Yes	n/a	n/a	n/a
<a href="#">TYPE_ROTATION_VECTOR</a>	Yes	Yes	n/a	n/a
<a href="#">TYPE_TEMPERATURE</a>	Yes <sup>2</sup>	Yes	Yes	Yes

<sup>1</sup> This sensor type was added in Android 1.5 (API Level 3), but it was not available for use until Android 2.3 (API Level 9).

<sup>2</sup> This sensor is available, but it has been deprecated.

## Identifying Sensors and Sensor Capabilities

The Android sensor framework provides several methods that make it easy for you to determine at runtime which sensors are on a device. The API also provides methods that let you determine the capabilities of each sensor, such as its maximum range, its resolution, and its power requirements.

To identify the sensors that are on a device you first need to get a reference to the sensor service. To do this, you create an instance of the [SensorManager](#) class by calling the [getSystemService\(\)](#) method and passing in the [SENSOR\\_SERVICE](#) argument. For example:

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

Next, you can get a listing of every sensor on a device by calling the [getSensorList\(\)](#) method and using the [TYPE\\_ALL](#) constant. For example:

```
List<Sensor> deviceSensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);
```

If you want to list all of the sensors of a given type, you could use another constant instead of [TYPE\\_ALL](#) such as [TYPE\\_GYROSCOPE](#), [TYPE\\_LINEAR\\_ACCELERATION](#), or [TYPE\\_GRAVITY](#).

You can also determine whether a specific type of sensor exists on a device by using the [getDefaultSensor\(\)](#) method and passing in the type constant for a specific sensor. If a device has more than one sensor of a given type, one of the sensors must be designated as the default sensor. If a default sensor does not exist for a given type of sensor, the method call returns null, which means the device does not have that type of sensor. For example, the following code checks whether there's a magnetometer on a device:

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

```

if (mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null) {
    // Success! There's a magnetometer.
}
else {
    // Failure! No magnetometer.
}

```

**Note:** Android does not require device manufacturers to build any particular types of sensors into their Android-powered devices, so devices can have a wide range of sensor configurations.

In addition to listing the sensors that are on a device, you can use the public methods of the [Sensor](#) class to determine the capabilities and attributes of individual sensors. This is useful if you want your application to behave differently based on which sensors or sensor capabilities are available on a device. For example, you can use the [getResolution\(\)](#) and [getMaximumRange\(\)](#) methods to obtain a sensor's resolution and maximum range of measurement. You can also use the [getPower\(\)](#) method to obtain a sensor's power requirements.

Two of the public methods are particularly useful if you want to optimize your application for different manufacturer's sensors or different versions of a sensor. For example, if your application needs to monitor user gestures such as tilt and shake, you could create one set of data filtering rules and optimizations for newer devices that have a specific vendor's gravity sensor, and another set of data filtering rules and optimizations for devices that do not have a gravity sensor and have only an accelerometer. The following code sample shows you how you can use the [getVendor\(\)](#) and [getVersion\(\)](#) methods to do this. In this sample, we're looking for a gravity sensor that lists Google Inc. as the vendor and has a version number of 3. If that particular sensor is not present on the device, we try to use the accelerometer.

```

private SensorManager mSensorManager;
private Sensor mSensor;

...

mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

if (mSensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY) != null) {
    List<Sensor> gravSensors = mSensorManager.getSensorList(Sensor.TYPE_GRAVITY);
    for(int i=0; i<gravSensors.size(); i++) {
        if ((gravSensors.get(i).getVendor().contains("Google Inc.")) &&
            (gravSensors.get(i).getVersion() == 3)) {
            // Use the version 3 gravity sensor.
            mSensor = gravSensors.get(i);
        }
    }
} else{
    // Use the accelerometer.
    if (mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER) != null) {
        mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    }
    else{
        // Sorry, there are no accelerometers on your device.
        // You can't play this game.
    }
}

```

Another useful method is the [getMinDelay\(\)](#) method, which returns the minimum time interval (in microseconds) a sensor can use to sense data. Any sensor that returns a non-zero value for the [getMinDelay\(\)](#) method is a streaming sensor. Streaming sensors sense data at regular intervals and were introduced in Android 2.3 (API Level 9). If a sensor returns zero when you call the [getMinDelay\(\)](#) method, it means the sensor is not a streaming sensor because it reports data only when there is a change in the parameters it is sensing.

The [getMinDelay\(\)](#) method is useful because it lets you determine the maximum rate at which a sensor can acquire data. If certain features in your application require high data acquisition rates or a streaming sensor, you can use this method to determine whether a sensor meets those requirements and then enable or disable the relevant features in your application accordingly.

**Caution:** A sensor's maximum data acquisition rate is not necessarily the rate at which the sensor framework delivers sensor data to your application. The sensor framework reports data through sensor events, and several factors influence the rate at which your application receives sensor events. For more information, see [Monitoring Sensor Events](#).

## Monitoring Sensor Events

To monitor raw sensor data you need to implement two callback methods that are exposed through the [SensorEventListener](#) interface: [onAccuracyChanged\(\)](#) and [onSensorChanged\(\)](#). The Android system calls these methods whenever the following occurs:

- **A sensor's accuracy changes.**

In this case the system invokes the [onAccuracyChanged\(\)](#) method, providing you with a reference to the [Sensor](#) object that changed and the new accuracy of the sensor. Accuracy is represented by one of four status constants: [SENSOR\\_STATUS\\_ACCURACY\\_LOW](#), [SENSOR\\_STATUS\\_ACCURACY\\_MEDIUM](#), [SENSOR\\_STATUS\\_ACCURACY\\_HIGH](#), or [SENSOR\\_STATUS\\_UNRELIABLE](#).

- **A sensor reports a new value.**

In this case the system invokes the [onSensorChanged\(\)](#) method, providing you with a [SensorEvent](#) object. A [SensorEvent](#) object contains information about the new sensor data, including: the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, and the new data that the sensor recorded.

The following code shows how to use the [onSensorChanged\(\)](#) method to monitor data from the light sensor. This example displays the raw sensor data in a [TextView](#) that is defined in the main.xml file as `sensor_data`.

```
public class SensorActivity extends Activity implements SensorEventListener {
    private SensorManager mSensorManager;
    private Sensor mLight;

    @Override
    public final void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
    }
}
```

```

@Override
public final void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Do something here if sensor accuracy changes.
}

@Override
public final void onSensorChanged(SensorEvent event) {
    // The light sensor returns a single value.
    // Many sensors return 3 values, one for each axis.
    float lux = event.values[0];
    // Do something with this sensor value.
}

@Override
protected void onResume() {
    super.onResume();
    mSensorManager.registerListener(this, mLight, SensorManager.SENSOR_DELAY_NORMAL);
}

@Override
protected void onPause() {
    super.onPause();
    mSensorManager.unregisterListener(this);
}
}

```

In this example, the default data delay (`SENSOR_DELAY_NORMAL`) is specified when the [registerListener\(\)](#) method is invoked. The data delay (or sampling rate) controls the interval at which sensor events are sent to your application via the [onSensorChanged\(\)](#) callback method. The default data delay is suitable for monitoring typical screen orientation changes and uses a delay of 200,000 microseconds. You can specify other data delays, such as `SENSOR_DELAY_GAME` (20,000 microsecond delay), `SENSOR_DELAY_UI` (60,000 microsecond delay), or `SENSOR_DELAY_FASTEST` (0 microsecond delay). As of Android 3.0 (API Level 11) you can also specify the delay as an absolute value (in microseconds).

The delay that you specify is only a suggested delay. The Android system and other applications can alter this delay. As a best practice, you should specify the largest delay that you can because the system typically uses a smaller delay than the one you specify (that is, you should choose the slowest sampling rate that still meets the needs of your application). Using a larger delay imposes a lower load on the processor and therefore uses less power.

There is no public method for determining the rate at which the sensor framework is sending sensor events to your application; however, you can use the timestamps that are associated with each sensor event to calculate the sampling rate over several events. You should not have to change the sampling rate (delay) once you set it. If for some reason you do need to change the delay, you will have to unregister and reregister the sensor listener.

It's also important to note that this example uses the [onResume\(\)](#) and [onPause\(\)](#) callback methods to register and unregister the sensor event listener. As a best practice you should always disable sensors you don't need, especially when your activity is paused. Failing to do so can drain the battery in just a few hours because some sensors have substantial power requirements and can use up battery power quickly. The system will not disable sensors automatically when the screen turns off.

# Handling Different Sensor Configurations

Android does not specify a standard sensor configuration for devices, which means device manufacturers can incorporate any sensor configuration that they want into their Android-powered devices. As a result, devices can include a variety of sensors in a wide range of configurations. For example, the Motorola Xoom has a pressure sensor, but the Samsung Nexus S does not. Likewise, the Xoom and Nexus S have gyroscopes, but the HTC Nexus One does not. If your application relies on a specific type of sensor, you have to ensure that the sensor is present on a device so your app can run successfully. You have two options for ensuring that a given sensor is present on a device:

- Detect sensors at runtime and enable or disable application features as appropriate.
- Use Google Play filters to target devices with specific sensor configurations.

Each option is discussed in the following sections.

## Detecting sensors at runtime

If your application uses a specific type of sensor, but doesn't rely on it, you can use the sensor framework to detect the sensor at runtime and then disable or enable application features as appropriate. For example, a navigation application might use the temperature sensor, pressure sensor, GPS sensor, and geomagnetic field sensor to display the temperature, barometric pressure, location, and compass bearing. If a device doesn't have a pressure sensor, you can use the sensor framework to detect the absence of the pressure sensor at runtime and then disable the portion of your application's UI that displays pressure. For example, the following code checks whether there's a pressure sensor on a device:

```
private SensorManager mSensorManager;  
...  
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);  
if (mSensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE) != null) {  
    // Success! There's a pressure sensor.  
}  
else {  
    // Failure! No pressure sensor.  
}
```

## Using Google Play filters to target specific sensor configurations

If you are publishing your application on Google Play you can use the `<uses-feature>` element in your manifest file to filter your application from devices that do not have the appropriate sensor configuration for your application. The `<uses-feature>` element has several hardware descriptors that let you filter applications based on the presence of specific sensors. The sensors you can list include: accelerometer, barometer, compass (geomagnetic field), gyroscope, light, and proximity. The following is an example manifest entry that filters apps that do not have an accelerometer:

```
<uses-feature android:name="android.hardware.sensor.accelerometer"  
            android:required="true" />
```

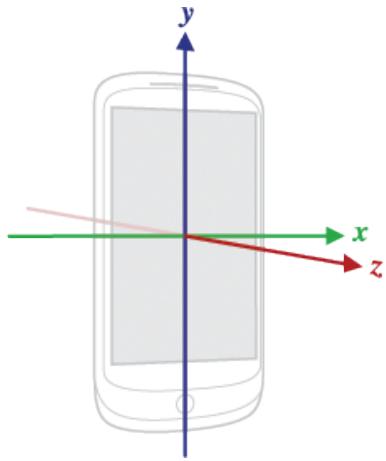
If you add this element and descriptor to your application's manifest, users will see your application on Google Play only if their device has an accelerometer.

You should set the descriptor to `android:required="true"` only if your application relies entirely on a specific sensor. If your application uses a sensor for some functionality, but still runs without the sensor, you should list the sensor in the `<uses-feature>` element, but set the descriptor to an-

`droid:required="false"`. This helps ensure that devices can install your app even if they do not have that particular sensor. This is also a project management best practice that helps you keep track of the features your application uses. Keep in mind, if your application uses a particular sensor, but still runs without the sensor, then you should detect the sensor at runtime and disable or enable application features as appropriate.

## Sensor Coordinate System

In general, the sensor framework uses a standard 3-axis coordinate system to express data values. For most sensors, the coordinate system is defined relative to the device's screen when the device is held in its default orientation (see figure 1). When a device is held in its default orientation, the X axis is horizontal and points to the right, the Y axis is vertical and points up, and the Z axis points toward the outside of the screen face. In this system, coordinates behind the screen have negative Z values. This coordinate system is used by the following sensors:



**Figure 1.** Coordinate system (relative to a device) that's used by the Sensor API.

- [Acceleration sensor](#)
- [Gravity sensor](#)
- [Gyroscope](#)
- [Linear acceleration sensor](#)
- [Geomagnetic field sensor](#)

The most important point to understand about this coordinate system is that the axes are not swapped when the device's screen orientation changes—that is, the sensor's coordinate system never changes as the device moves. This behavior is the same as the behavior of the OpenGL coordinate system.

Another point to understand is that your application must not assume that a device's natural (default) orientation is portrait. The natural orientation for many tablet devices is landscape. And the sensor coordinate system is always based on the natural orientation of a device.

Finally, if your application matches sensor data to the on-screen display, you need to use the [`getRotation\(\)`](#) method to determine screen rotation, and then use the [`remapCoordinateSystem\(\)`](#) method to map sensor coordinates to screen coordinates. You need to do this even if your manifest specifies portrait-only display.

For more information about the sensor coordinate system, including information about how to handle screen rotations, see [One Screen Turn Deserves Another](#).

**Note:** Some sensors and methods use a coordinate system that is relative to the world's frame of reference (as opposed to the device's frame of reference). These sensors and methods return data that represent device motion

or device position relative to the earth. For more information, see the [getOrientation\(\)](#) method, the [getRotationMatrix\(\)](#) method, [Orientation Sensor](#), and [Rotation Vector Sensor](#).

## Best Practices for Accessing and Using Sensors

As you design your sensor implementation, be sure to follow the guidelines that are discussed in this section. These guidelines are recommended best practices for anyone who is using the sensor framework to access sensors and acquire sensor data.

### Unregister sensor listeners

Be sure to unregister a sensor's listener when you are done using the sensor or when the sensor activity pauses. If a sensor listener is registered and its activity is paused, the sensor will continue to acquire data and use battery resources unless you unregister the sensor. The following code shows how to use the [onPause\(\)](#) method to unregister a listener:

```
private SensorManager mSensorManager;  
...  
@Override  
protected void onPause() {  
    super.onPause();  
    mSensorManager.unregisterListener(this);  
}
```

For more information, see [unregisterListener\(SensorEventListener\)](#).

### Don't test your code on the emulator

You currently can't test sensor code on the emulator because the emulator cannot emulate sensors. You must test your sensor code on a physical device. There are, however, sensor simulators that you can use to simulate sensor output.

### Don't block the onSensorChanged() method

Sensor data can change at a high rate, which means the system may call the [onSensorChanged\(SensorEvent\)](#) method quite often. As a best practice, you should do as little as possible within the [onSensorChanged\(SensorEvent\)](#) method so you don't block it. If your application requires you to do any data filtering or reduction of sensor data, you should perform that work outside of the [onSensorChanged\(SensorEvent\)](#) method.

### Avoid using deprecated methods or sensor types

Several methods and constants have been deprecated. In particular, the [TYPE\\_ORIENTATION](#) sensor type has been deprecated. To get orientation data you should use the [getOrientation\(\)](#) method instead. Likewise, the [TYPE\\_TEMPERATURE](#) sensor type has been deprecated. You should use the [TYPE\\_AMBIENT\\_TEMPERATURE](#) sensor type instead on devices that are running Android 4.0.

### Verify sensors before you use them

Always verify that a sensor exists on a device before you attempt to acquire data from it. Don't assume that a sensor exists simply because it's a frequently-used sensor. Device manufacturers are not required to provide any particular sensors in their devices.

## **Choose sensor delays carefully**

When you register a sensor with the [`registerListener\(\)`](#) method, be sure you choose a delivery rate that is suitable for your application or use-case. Sensors can provide data at very high rates. Allowing the system to send extra data that you don't need wastes system resources and uses battery power.

# Motion Sensors

## In this document

1. [Using the Accelerometer](#)
2. [Using the Gravity Sensor](#)
3. [Using the Gyroscope](#)
4. [Using the Linear Accelerometer](#)
5. [Using the Rotation Vector Sensor](#)

## Key classes and interfaces

1. [Sensor](#)
2. [SensorEvent](#)
3. [SensorManager](#)
4. [SensorEventListener](#)

## Related samples

1. [Accelerometer Play](#)
2. [API Demos \(OS - RotationVectorDemo\)](#)
3. [API Demos \(OS - Sensors\)](#)

## See also

1. [Sensors](#)
2. [Sensors Overview](#)
3. [Position Sensors](#)
4. [Environment Sensors](#)

The Android platform provides several sensors that let you monitor the motion of a device. Two of these sensors are always hardware-based (the accelerometer and gyroscope), and three of these sensors can be either hardware-based or software-based (the gravity, linear acceleration, and rotation vector sensors). For example, on some devices the software-based sensors derive their data from the accelerometer and magnetometer, but on other devices they may also use the gyroscope to derive their data. Most Android-powered devices have an accelerometer, and many now include a gyroscope. The availability of the software-based sensors is more variable because they often rely on one or more hardware sensors to derive their data.

Motion sensors are useful for monitoring device movement, such as tilt, shake, rotation, or swing. The movement is usually a reflection of direct user input (for example, a user steering a car in a game or a user controlling a ball in a game), but it can also be a reflection of the physical environment in which the device is sitting (for example, moving with you while you drive your car). In the first case, you are monitoring motion relative to the device's frame of reference or your application's frame of reference; in the second case you are monitoring motion relative to the world's frame of reference. Motion sensors by themselves are not typically used to monitor device position, but they can be used with other sensors, such as the geomagnetic field sensor, to determine a device's position relative to the world's frame of reference (see [Position Sensors](#) for more information).

All of the motion sensors return multi-dimensional arrays of sensor values for each [SensorEvent](#). For example, during a single sensor event the accelerometer returns acceleration force data for the three coordinate axes, and the gyroscope returns rate of rotation data for the three coordinate axes. These data values are returned in a

float array (`values`) along with other `SensorEvent` parameters. Table 1 summarizes the motion sensors that are available on the Android platform.

**Table 1.** Motion sensors that are supported on the Android platform.

Sensor	Sensor event data	Description	Units of measure
<a href="#">TYPE_ACCELEROMETER</a>	<code>SensorEvent.values[0]</code>	Acceleration force along the x axis (including gravity).	$\text{m/s}^2$
	<code>SensorEvent.values[1]</code>	Acceleration force along the y axis (including gravity).	
	<code>SensorEvent.values[2]</code>	Acceleration force along the z axis (including gravity).	
<a href="#">TYPE_GRAVITY</a>	<code>SensorEvent.values[0]</code>	Force of gravity along the x axis.	$\text{m/s}^2$
	<code>SensorEvent.values[1]</code>	Force of gravity along the y axis.	
	<code>SensorEvent.values[2]</code>	Force of gravity along the z axis.	
<a href="#">TYPE_GYROSCOPE</a>	<code>SensorEvent.values[0]</code>	Rate of rotation around the x axis.	$\text{rad/s}$
	<code>SensorEvent.values[1]</code>	Rate of rotation around the y axis.	
	<code>SensorEvent.values[2]</code>	Rate of rotation around the z axis.	
<a href="#">TYPE_LINEAR_ACCELERATION</a>	<code>SensorEvent.values[0]</code>	Acceleration force along the x axis (excluding gravity).	$\text{m/s}^2$
	<code>SensorEvent.values[1]</code>	Acceleration force along the y axis (excluding gravity).	
	<code>SensorEvent.values[2]</code>	Acceleration force along the z axis (excluding gravity).	
<a href="#">TYPE_ROTATION_VECTOR</a>	<code>SensorEvent.values[0]</code>	Rotation vector component along the x axis ( $x * \sin(\theta/2)$ ).	Unitless
	<code>SensorEvent.values[1]</code>	Rotation vector component along the y axis ( $y * \sin(\theta/2)$ ).	
	<code>SensorEvent.values[2]</code>	Rotation vector component along the z axis ( $z * \sin(\theta/2)$ ).	
	<code>SensorEvent.values[3]</code>	Scalar component of the rotation vector ( $\cos(\theta/2)$ ). <sup>1</sup>	

<sup>1</sup> The scalar component is an optional value.

The rotation vector sensor and the gravity sensor are the most frequently used sensors for motion detection and monitoring. The rotational vector sensor is particularly versatile and can be used for a wide range of motion-related tasks, such as detecting gestures, monitoring angular change, and monitoring relative orientation changes. For example, the rotational vector sensor is ideal if you are developing a game, an augmented reality application, a 2-dimensional or 3-dimensional compass, or a camera stabilization app. In most cases, using these sensors is a better choice than using the accelerometer and geomagnetic field sensor or the orientation sensor.

## Android Open Source Project Sensors

The Android Open Source Project (AOSP) provides three software-based motion sensors: a gravity sensor, a linear acceleration sensor, and a rotation vector sensor. These sensors were updated in Android 4.0 and now use a device's gyroscope (in addition to other sensors) to improve stability and performance. If you want to try these sensors, you can identify them by using the [getVendor\(\)](#) method and the [getVersion\(\)](#) method (the vendor is Google Inc.; the version number is 3). Identifying these sensors by vendor and version number is necessary because the Android system considers these three sensors to be secondary sensors. For example, if a device manufacturer provides their own gravity sensor, then the AOSP gravity sensor shows up as a secondary gravity sensor. All three of these sensors rely on a gyroscope: if a device does not have a gyroscope, these sensors do not show up and are not available for use.

## Using the Accelerometer

An acceleration sensor measures the acceleration applied to the device, including the force of gravity. The following code shows you how to get an instance of the default acceleration sensor:

```
private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
```

Conceptually, an acceleration sensor determines the acceleration that is applied to a device ( $A_d$ ) by measuring the forces that are applied to the sensor itself ( $F_s$ ) using the following relationship:

$$A_d = - \sum F_s / \text{mass}$$

However, the force of gravity is always influencing the measured acceleration according to the following relationship:

$$A_d = -g - \sum F / \text{mass}$$

For this reason, when the device is sitting on a table (and not accelerating), the accelerometer reads a magnitude of  $g = 9.81 \text{ m/s}^2$ . Similarly, when the device is in free fall and therefore rapidly accelerating toward the ground at  $9.81 \text{ m/s}^2$ , its accelerometer reads a magnitude of  $g = 0 \text{ m/s}^2$ . Therefore, to measure the real acceleration of the device, the contribution of the force of gravity must be removed from the accelerometer data. This can be achieved by applying a high-pass filter. Conversely, a low-pass filter can be used to isolate the force of gravity. The following example shows how you can do this:

```
public void onSensorChanged(SensorEvent event) {
    // In this example, alpha is calculated as t / (t + dT),
    // where t is the low-pass filter's time-constant and
    // dT is the event delivery rate.

    final float alpha = 0.8;

    // Isolate the force of gravity with the low-pass filter.
    gravity[0] = alpha * gravity[0] + (1 - alpha) * event.values[0];
    gravity[1] = alpha * gravity[1] + (1 - alpha) * event.values[1];
    gravity[2] = alpha * gravity[2] + (1 - alpha) * event.values[2];

    // Remove the gravity contribution with the high-pass filter.
```

```

linear_acceleration[0] = event.values[0] - gravity[0];
linear_acceleration[1] = event.values[1] - gravity[1];
linear_acceleration[2] = event.values[2] - gravity[2];
}

```

**Note:** You can use many different techniques to filter sensor data. The code sample above uses a simple filter constant (alpha) to create a low-pass filter. This filter constant is derived from a time constant ( $t$ ), which is a rough representation of the latency that the filter adds to the sensor events, and the sensor's event delivery rate ( $dt$ ). The code sample uses an alpha value of 0.8 for demonstration purposes. If you use this filtering method you may need to choose a different alpha value.

Accelerometers use the standard sensor [coordinate system](#). In practice, this means that the following conditions apply when a device is laying flat on a table in its natural orientation:

- If you push the device on the left side (so it moves to the right), the x acceleration value is positive.
- If you push the device on the bottom (so it moves away from you), the y acceleration value is positive.
- If you push the device toward the sky with an acceleration of  $A \text{ m/s}^2$ , the z acceleration value is equal to  $A + 9.81$ , which corresponds to the acceleration of the device ( $+A \text{ m/s}^2$ ) minus the force of gravity ( $-9.81 \text{ m/s}^2$ ).
- The stationary device will have an acceleration value of  $+9.81$ , which corresponds to the acceleration of the device ( $0 \text{ m/s}^2$  minus the force of gravity, which is  $-9.81 \text{ m/s}^2$ ).

In general, the accelerometer is a good sensor to use if you are monitoring device motion. Almost every Android-powered handset and tablet has an accelerometer, and it uses about 10 times less power than the other motion sensors. One drawback is that you might have to implement low-pass and high-pass filters to eliminate gravitational forces and reduce noise.

The Android SDK provides a sample application that shows how to use the acceleration sensor ([Accelerometer Play](#)).

## Using the Gravity Sensor

The gravity sensor provides a three dimensional vector indicating the direction and magnitude of gravity. The following code shows you how to get an instance of the default gravity sensor:

```

private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY);

```

The units are the same as those used by the acceleration sensor ( $\text{m/s}^2$ ), and the coordinate system is the same as the one used by the acceleration sensor.

**Note:** When a device is at rest, the output of the gravity sensor should be identical to that of the accelerometer.

## Using the Gyroscope

The gyroscope measures the rate or rotation in rad/s around a device's x, y, and z axis. The following code shows you how to get an instance of the default gyroscope:

```

private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);

```

The sensor's [coordinate system](#) is the same as the one used for the acceleration sensor. Rotation is positive in the counter-clockwise direction; that is, an observer looking from some positive location on the x, y or z axis at a device positioned on the origin would report positive rotation if the device appeared to be rotating counter clockwise. This is the standard mathematical definition of positive rotation and is not the same as the definition for roll that is used by the orientation sensor.

Usually, the output of the gyroscope is integrated over time to calculate a rotation describing the change of angles over the timestep. For example:

```

// Create a constant to convert nanoseconds to seconds.
private static final float NS2S = 1.0f / 1000000000.0f;
private final float[] deltaRotationVector = new float[4]();
private float timestamp;

public void onSensorChanged(SensorEvent event) {
    // This timestep's delta rotation to be multiplied by the current rotation
    // after computing it from the gyro sample data.
    if (timestamp != 0) {
        final float dT = (event.timestamp - timestamp) * NS2S;
        // Axis of the rotation sample, not normalized yet.
        float axisX = event.values[0];
        float axisY = event.values[1];
        float axisZ = event.values[2];

        // Calculate the angular speed of the sample
        float omegaMagnitude = sqrt(axisX*axisX + axisY*axisY + axisZ*axisZ);

        // Normalize the rotation vector if it's big enough to get the axis
        // (that is, EPSILON should represent your maximum allowable margin of error)
        if (omegaMagnitude > EPSILON) {
            axisX /= omegaMagnitude;
            axisY /= omegaMagnitude;
            axisZ /= omegaMagnitude;
        }

        // Integrate around this axis with the angular speed by the timestep
        // in order to get a delta rotation from this sample over the timestep
        // We will convert this axis-angle representation of the delta rotation
        // into a quaternion before turning it into the rotation matrix.
        float thetaOverTwo = omegaMagnitude * dT / 2.0f;
        float sinThetaOverTwo = sin(thetaOverTwo);
        float cosThetaOverTwo = cos(thetaOverTwo);
        deltaRotationVector[0] = sinThetaOverTwo * axisX;
        deltaRotationVector[1] = sinThetaOverTwo * axisY;
        deltaRotationVector[2] = sinThetaOverTwo * axisZ;
        deltaRotationVector[3] = cosThetaOverTwo;
    }
    timestamp = event.timestamp;
}

```

```

float[] deltaRotationMatrix = new float[9];
SensorManager.getRotationMatrixFromVector(deltaRotationMatrix, deltaRotationMatrix);
    // User code should concatenate the delta rotation we computed with the current
    // in order to get the updated rotation.
    // rotationCurrent = rotationCurrent * deltaRotationMatrix;
}
}

```

Standard gyroscopes provide raw rotational data without any filtering or correction for noise and drift (bias). In practice, gyroscope noise and drift will introduce errors that need to be compensated for. You usually determine the drift (bias) and noise by monitoring other sensors, such as the gravity sensor or accelerometer.

## Using the Linear Accelerometer

The linear acceleration sensor provides you with a three-dimensional vector representing acceleration along each device axis, excluding gravity. The following code shows you how to get an instance of the default linear acceleration sensor:

```

private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION);

```

Conceptually, this sensor provides you with acceleration data according to the following relationship:

$$\text{linear acceleration} = \text{acceleration} - \text{acceleration due to gravity}$$

You typically use this sensor when you want to obtain acceleration data without the influence of gravity. For example, you could use this sensor to see how fast your car is going. The linear acceleration sensor always has an offset, which you need to remove. The simplest way to do this is to build a calibration step into your application. During calibration you can ask the user to set the device on a table, and then read the offsets for all three axes. You can then subtract that offset from the acceleration sensor's direct readings to get the actual linear acceleration.

The sensor [coordinate system](#) is the same as the one used by the acceleration sensor, as are the units of measure ( $\text{m/s}^2$ ).

## Using the Rotation Vector Sensor

The rotation vector represents the orientation of the device as a combination of an angle and an axis, in which the device has rotated through an angle  $\theta$  around an axis (x, y, or z). The following code shows you how to get an instance of the default rotation vector sensor:

```

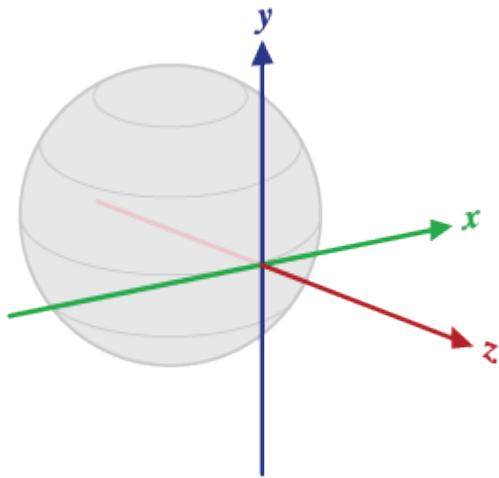
private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR);

```

The three elements of the rotation vector are expressed as follows:

```
x*sin(θ/2)  
y*sin(θ/2)  
z*sin(θ/2)
```

Where the magnitude of the rotation vector is equal to  $\sin(\theta/2)$ , and the direction of the rotation vector is equal to the direction of the axis of rotation.



**Figure 1.** Coordinate system used by the rotation vector sensor.

The three elements of the rotation vector are equal to the last three components of a unit quaternion ( $\cos(\theta/2)$ ,  $x*\sin(\theta/2)$ ,  $y*\sin(\theta/2)$ ,  $z*\sin(\theta/2)$ ). Elements of the rotation vector are unitless. The x, y, and z axes are defined in the same way as the acceleration sensor. The reference coordinate system is defined as a direct orthonormal basis (see figure 1). This coordinate system has the following characteristics:

- X is defined as the vector product Y x Z. It is tangential to the ground at the device's current location and points approximately East.
- Y is tangential to the ground at the device's current location and points toward the geomagnetic North Pole.
- Z points toward the sky and is perpendicular to the ground plane.

The Android SDK provides a sample application that shows how to use the rotation vector sensor. The sample application is located in the API Demos code ([OS - RotationVectorDemo](#)).

# Position Sensors

## In this document

1. [Using the Orientation Sensor](#)
2. [Using the Geomagnetic Field Sensor](#)
3. [Using the Proximity Sensor](#)

## Key classes and interfaces

1. [Sensor](#)
2. [SensorEvent](#)
3. [SensorManager](#)
4. [SensorEventListener](#)

## Related samples

1. [Accelerometer Play](#)
2. [API Demos \(OS - RotationVectorDemo\)](#)
3. [API Demos \(OS - Sensors\)](#)

## See also

1. [Sensors](#)
2. [Sensors Overview](#)
3. [Motion Sensors](#)
4. [Environment Sensors](#)

The Android platform provides two sensors that let you determine the position of a device: the geomagnetic field sensor and the orientation sensor. The Android platform also provides a sensor that lets you determine how close the face of a device is to an object (known as the proximity sensor). The geomagnetic field sensor and the proximity sensor are hardware-based. Most handset and tablet manufacturers include a geomagnetic field sensor. Likewise, handset manufacturers usually include a proximity sensor to determine when a handset is being held close to a user's face (for example, during a phone call). The orientation sensor is software-based and derives its data from the accelerometer and the geomagnetic field sensor.

**Note:** The orientation sensor was deprecated in Android 2.2 (API Level 8).

Position sensors are useful for determining a device's physical position in the world's frame of reference. For example, you can use the geomagnetic field sensor in combination with the accelerometer to determine a device's position relative to the magnetic North Pole. You can also use the orientation sensor (or similar sensor-based orientation methods) to determine a device's position in your application's frame of reference. Position sensors are not typically used to monitor device movement or motion, such as shake, tilt, or thrust (for more information, see [Motion Sensors](#)).

The geomagnetic field sensor and orientation sensor return multi-dimensional arrays of sensor values for each [SensorEvent](#). For example, the orientation sensor provides geomagnetic field strength values for each of the three coordinate axes during a single sensor event. Likewise, the orientation sensor provides azimuth (yaw), pitch, and roll values during a single sensor event. For more information about the coordinate systems that are used by sensors, see [Sensor Coordinate Systems](#). The proximity sensor provides a single value for each sensor event. Table 1 summarizes the position sensors that are supported on the Android platform.

**Table 1.** Position sensors that are supported on the Android platform.

Sensor	Sensor event data	Description	Units of measure
<u>TYPE_MAGNETIC_FIELD</u>	SensorEvent.values[0]	Geomagnetic field strength along the x axis.	$\mu\text{T}$
	SensorEvent.values[1]	Geomagnetic field strength along the y axis.	
	SensorEvent.values[2]	Geomagnetic field strength along the z axis.	
<u>TYPE_ORIENTATION</u> <sup>1</sup>	SensorEvent.values[0]	Azimuth (angle around the z-axis).	Degrees
	SensorEvent.values[1]	Pitch (angle around the x-axis).	
	SensorEvent.values[2]	Roll (angle around the y-axis).	
<u>TYPE_PROXIMITY</u>	SensorEvent.values[0]	Distance from object. <sup>2</sup>	cm

<sup>1</sup> This sensor was deprecated in Android 2.2 (API Level 8). The sensor framework provides alternate methods for acquiring device orientation, which are discussed in [Using the Orientation Sensor](#).

<sup>2</sup> Some proximity sensors provide only binary values representing near and far.

## Using the Orientation Sensor

The orientation sensor lets you monitor the position of a device relative to the earth's frame of reference (specifically, magnetic north). The following code shows you how to get an instance of the default orientation sensor :

```
private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);
```

The orientation sensor derives its data by using a device's geomagnetic field sensor in combination with a device's accelerometer. Using these two hardware sensors, an orientation sensor provides data for the following three dimensions:

- Azimuth (degrees of rotation around the z axis). This is the angle between magnetic north and the device's y axis. For example, if the device's y axis is aligned with magnetic north this value is 0, and if the device's y axis is pointing south this value is 180. Likewise, when the y axis is pointing east this value is 90 and when it is pointing west this value is 270.
- Pitch (degrees of rotation around the x axis). This value is positive when the positive z axis rotates toward the positive y axis, and it is negative when the positive z axis rotates toward the negative y axis. The range of values is 180 degrees to -180 degrees.
- Roll (degrees of rotation around the y axis). This value is positive when the positive z axis rotates toward the positive x axis, and it is negative when the positive z axis rotates toward the negative x axis. The range of values is 90 degrees to -90 degrees.

This definition is different from yaw, pitch, and roll used in aviation, where the X axis is along the long side of the plane (tail to nose). Also, for historical reasons the roll angle is positive in the clockwise direction (mathematically speaking, it should be positive in the counter-clockwise direction).

The orientation sensor derives its data by processing the raw sensor data from the accelerometer and the geo-magnetic field sensor. Because of the heavy processing that is involved, the accuracy and precision of the ori-

tation sensor is diminished (specifically, this sensor is only reliable when the roll component is 0). As a result, the orientation sensor was deprecated in Android 2.2 (API level 8). Instead of using raw data from the orientation sensor, we recommend that you use the [getRotationMatrix\(\)](#) method in conjunction with the [getOrientation\(\)](#) method to compute orientation values. You can also use the [remapCoordinateSystem\(\)](#) method to translate the orientation values to your application's frame of reference.

The following code sample shows how to acquire orientation data directly from the orientation sensor. We recommend that you do this only if a device has negligible roll.

```
public class SensorActivity extends Activity implements SensorEventListener {

    private SensorManager mSensorManager;
    private Sensor mOrientation;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        mOrientation = mSensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // Do something here if sensor accuracy changes.
        // You must implement this callback in your code.
    }

    @Override
    protected void onResume() {
        super.onResume();
        mSensorManager.registerListener(this, mOrientation, SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    protected void onPause() {
        super.onPause();
        mSensorManager.unregisterListener(this);
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
        float azimuth_angle = event.values[0];
        float pitch_angle = event.values[1];
        float roll_angle = event.values[2];
        // Do something with these orientation angles.
    }
}
```

You do not usually need to perform any data processing or filtering of the raw data that you obtain from an orientation sensor, other than translating the sensor's coordinate system to your application's frame of reference. The [Accelerometer Play](#) sample shows you how to translate acceleration sensor data into another frame of reference; the technique is similar to the one you might use with the orientation sensor.

# Using the Geomagnetic Field Sensor

The geomagnetic field sensor lets you monitor changes in the earth's magnetic field. The following code shows you how to get an instance of the default geomagnetic field sensor:

```
private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
```

This sensor provides raw field strength data (in  $\mu\text{T}$ ) for each of the three coordinate axes. Usually, you do not need to use this sensor directly. Instead, you can use the rotation vector sensor to determine raw rotational movement or you can use the accelerometer and geomagnetic field sensor in conjunction with the [getRotationMatrix\(\)](#) method to obtain the rotation matrix and the inclination matrix. You can then use these matrices with the [getOrientation\(\)](#) and [getInclination\(\)](#) methods to obtain azimuth and geomagnetic inclination data.

# Using the Proximity Sensor

The proximity sensor lets you determine how far away an object is from a device. The following code shows you how to get an instance of the default proximity sensor:

```
private SensorManager mSensorManager;
private Sensor mSensor;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
```

The proximity sensor is usually used to determine how far away a person's head is from the face of a handset device (for example, when a user is making or receiving a phone call). Most proximity sensors return the absolute distance, in cm, but some return only near and far values. The following code shows you how to use the proximity sensor:

```
public class SensorActivity extends Activity implements SensorEventListener {
    private SensorManager mSensorManager;
    private Sensor mProximity;

    @Override
    public final void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Get an instance of the sensor service, and use that to get an instance of
        // a particular sensor.
        mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        mProximity = mSensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
    }

    @Override
    public final void onAccuracyChanged(Sensor sensor, int accuracy) {
        // Do something here if sensor accuracy changes.
    }
}
```

```
@Override
public final void onSensorChanged(SensorEvent event) {
    float distance = event.values[0];
    // Do something with this sensor data.
}

@Override
protected void onResume() {
    // Register a listener for the sensor.
    super.onResume();
    mSensorManager.registerListener(this, mProximity, SensorManager.SENSOR_DELAY_NORMAL);
}

@Override
protected void onPause() {
    // Be sure to unregister the sensor when the activity pauses.
    super.onPause();
    mSensorManager.unregisterListener(this);
}
}
```

**Note:** Some proximity sensors return binary values that represent "near" or "far." In this case, the sensor usually reports its maximum range value in the far state and a lesser value in the near state. Typically, the far value is a value > 5 cm, but this can vary from sensor to sensor. You can determine a sensor's maximum range by using the [getMaximumRange\(\)](#) method.

# Environment Sensors

## In this document

1. [Using the Light, Pressure, and Temperature Sensors](#)
2. [Using the Humidity Sensor](#)

## Related samples

1. [Accelerometer Play](#)
2. [API Demos \(OS - RotationVectorDemo\)](#)
3. [API Demos \(OS - Sensors\)](#)

## See also

1. [Sensors](#)
2. [Sensors Overview](#)
3. [Position Sensors](#)
4. [Motion Sensors](#)

The Android platform provides four sensors that let you monitor various environmental properties. You can use these sensors to monitor relative ambient humidity, illuminance, ambient pressure, and ambient temperature near an Android-powered device. All four environment sensors are hardware-based and are available only if a device manufacturer has built them into a device. With the exception of the light sensor, which most device manufacturers use to control screen brightness, environment sensors are not always available on devices. Because of this, it's particularly important that you verify at runtime whether an environment sensor exists before you attempt to acquire data from it.

Unlike most motion sensors and position sensors, which return a multi-dimensional array of sensor values for each [SensorEvent](#), environment sensors return a single sensor value for each data event. For example, the temperature in °C or the pressure in hPa. Also, unlike motion sensors and position sensors, which often require high-pass or low-pass filtering, environment sensors do not typically require any data filtering or data processing. Table 1 provides a summary of the environment sensors that are supported on the Android platform.

**Table 1.** Environment sensors that are supported on the Android platform.

Sensor	Sensor event data	Units of measure	Data description
<a href="#">TYPE_AMBIENT_TEMPERATURE</a>	event.values[0]	°C	Ambient air temperature.
<a href="#">TYPE_LIGHT</a>	event.values[0]	lx	Illuminance.
<a href="#">TYPE_PRESSURE</a>	event.values[0]	hPa or mbar	Ambient air pressure.
<a href="#">TYPE_RELATIVE_HUMIDITY</a>	event.values[0]	%	Ambient relative humidity.
<a href="#">TYPE_TEMPERATURE</a>	event.values[0]	°C	Device temperature. <sup>1</sup>

<sup>1</sup> Implementations vary from device to device. This sensor was deprecated in Android 4.0 (API Level 14).

## Using the Light, Pressure, and Temperature Sensors

The raw data you acquire from the light, pressure, and temperature sensors usually requires no calibration, filtering, or modification, which makes them some of the easiest sensors to use. To acquire data from these sen-

sors you first create an instance of the [SensorManager](#) class, which you can use to get an instance of a physical sensor. Then you register a sensor listener in the [onResume\(\)](#) method, and start handling incoming sensor data in the [onSensorChanged\(\)](#) callback method. The following code shows you how to do this:

```
public class SensorActivity extends Activity implements SensorEventListener {  
    private SensorManager mSensorManager;  
    private Sensor mPressure;  
  
    @Override  
    public final void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
  
        // Get an instance of the sensor service, and use that to get an instance  
        // a particular sensor.  
        mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);  
        mPressure = mSensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE);  
    }  
  
    @Override  
    public final void onAccuracyChanged(Sensor sensor, int accuracy) {  
        // Do something here if sensor accuracy changes.  
    }  
  
    @Override  
    public final void onSensorChanged(SensorEvent event) {  
        float millibars_of_pressure = event.values[0];  
        // Do something with this sensor data.  
    }  
  
    @Override  
    protected void onResume() {  
        // Register a listener for the sensor.  
        super.onResume();  
        mSensorManager.registerListener(this, mPressure, SensorManager.SENSOR_DELAY_NORMAL);  
    }  
  
    @Override  
    protected void onPause() {  
        // Be sure to unregister the sensor when the activity pauses.  
        super.onPause();  
        mSensorManager.unregisterListener(this);  
    }  
}
```

You must always include implementations of both the [onAccuracyChanged\(\)](#) and [onSensorChanged\(\)](#) callback methods. Also, be sure that you always unregister a sensor when an activity pauses. This prevents a sensor from continually sensing data and draining the battery.

## Using the Humidity Sensor

You can acquire raw relative humidity data by using the humidity sensor the same way that you use the light, pressure, and temperature sensors. However, if a device has both a humidity sensor

([TYPE\\_RELATIVE\\_HUMIDITY](#)) and a temperature sensor ([TYPE\\_AMBIENT\\_TEMPERATURE](#)) you can use these two data streams to calculate the dew point and the absolute humidity.

## Dew point

The dew point is the temperature at which a given volume of air must be cooled, at constant barometric pressure, for water vapor to condense into water. The following equation shows how you can calculate the dew point:

$$t_d(t, RH) = T_n + \frac{\ln(RH/100\%) + m \cdot t / (T_n + t)}{m - [\ln(RH/100\%) + m \cdot t / (T_n + t)]}$$

Where,

- $t_d$  = dew point temperature in degrees C
- $t$  = actual temperature in degrees C
- RH = actual relative humidity in percent (%)
- $m = 17.62$
- $T_n = 243.12$

## Absolute humidity

The absolute humidity is the mass of water vapor in a given volume of dry air. Absolute humidity is measured in grams/meter<sup>3</sup>. The following equation shows how you can calculate the absolute humidity:

$$d_v(t, RH) = 216.7 \cdot \frac{(RH/100\%) \cdot A \cdot \exp(m \cdot t / (T_n + t))}{273.15 + t}$$

Where,

- $d_v$  = absolute humidity in grams/meter<sup>3</sup>
- $t$  = actual temperature in degrees C
- RH = actual relative humidity in percent (%)
- $m = 17.62$
- $T_n = 243.12$  degrees C
- $A = 6.112$  hPa



# Connectivity

Android provides rich APIs to let your app connect and interact with other devices over Bluetooth, NFC, Wi-Fi Direct, USB, and SIP, in addition to standard network connections.

## Blog Articles

### [Android's HTTP Clients](#)

Most network-connected Android apps will use HTTP to send and receive data. Android includes two HTTP clients: `HttpURLConnection` and Apache HTTP Client. Both support HTTPS, streaming uploads and downloads, configurable timeouts, IPv6 and connection pooling.

## Training

### [Transferring Data Without Draining the Battery](#)

This class demonstrates the best practices for scheduling and executing downloads using techniques such as caching, polling, and prefetching. You will learn how the power-use profile of the wireless radio can affect your choices on when, what, and how to transfer data in order to minimize impact on battery life.

### [Syncing to the Cloud](#)

This class covers different strategies for cloud enabled applications. It covers syncing data with the cloud using your own back-end web application, and backing up data using the cloud so that users can restore their data when installing your application on a new device.

# Bluetooth

## In this document

1. [The Basics](#)
2. [Bluetooth Permissions](#)
3. [Setting Up Bluetooth](#)
4. [Finding Devices](#)
  1. [Querying paired devices](#)
  2. [Discovering devices](#)
5. [Connecting Devices](#)
  1. [Connecting as a server](#)
  2. [Connecting as a client](#)
6. [Managing a Connection](#)
7. [Working with Profiles](#)
  1. [Vendor-specific AT commands](#)
  2. [Health Device Profile](#)

## Key classes

1. [BluetoothAdapter](#)
2. [BluetoothDevice](#)
3. [BluetoothSocket](#)
4. [BluetoothServerSocket](#)

## Related samples

1. [Bluetooth Chat](#)
2. [Bluetooth HDP \(Health Device Profile\)](#)

The Android platform includes support for the Bluetooth network stack, which allows a device to wirelessly exchange data with other Bluetooth devices. The application framework provides access to the Bluetooth functionality through the Android Bluetooth APIs. These APIs let applications wirelessly connect to other Bluetooth devices, enabling point-to-point and multipoint wireless features.

Using the Bluetooth APIs, an Android application can perform the following:

- Scan for other Bluetooth devices
- Query the local Bluetooth adapter for paired Bluetooth devices
- Establish RFCOMM channels
- Connect to other devices through service discovery
- Transfer data to and from other devices
- Manage multiple connections

This document describes how to use *Classic Bluetooth*. Classic Bluetooth is the right choice for more battery-intensive operations such as streaming and communicating between Android devices. For Bluetooth devices with low power requirements, Android 4.3 (API Level 18) introduces API support for Bluetooth Low Energy. To learn more, see [Bluetooth Low Energy](#).

# The Basics

This document describes how to use the Android Bluetooth APIs to accomplish the four major tasks necessary to communicate using Bluetooth: setting up Bluetooth, finding devices that are either paired or available in the local area, connecting devices, and transferring data between devices.

All of the Bluetooth APIs are available in the [android.bluetooth](#) package. Here's a summary of the classes and interfaces you will need to create Bluetooth connections:

## [BluetoothAdapter](#)

Represents the local Bluetooth adapter (Bluetooth radio). The [BluetoothAdapter](#) is the entry-point for all Bluetooth interaction. Using this, you can discover other Bluetooth devices, query a list of bonded (paired) devices, instantiate a [BluetoothDevice](#) using a known MAC address, and create a [BluetoothServerSocket](#) to listen for communications from other devices.

## [BluetoothDevice](#)

Represents a remote Bluetooth device. Use this to request a connection with a remote device through a [BluetoothSocket](#) or query information about the device such as its name, address, class, and bonding state.

## [BluetoothSocket](#)

Represents the interface for a Bluetooth socket (similar to a TCP [Socket](#)). This is the connection point that allows an application to exchange data with another Bluetooth device via InputStream and OutputStream.

## [BluetoothServerSocket](#)

Represents an open server socket that listens for incoming requests (similar to a TCP [ServerSocket](#)). In order to connect two Android devices, one device must open a server socket with this class. When a remote Bluetooth device makes a connection request to the this device, the [BluetoothServerSocket](#) will return a connected [BluetoothSocket](#) when the connection is accepted.

## [BluetoothClass](#)

Describes the general characteristics and capabilities of a Bluetooth device. This is a read-only set of properties that define the device's major and minor device classes and its services. However, this does not reliably describe all Bluetooth profiles and services supported by the device, but is useful as a hint to the device type.

## [BluetoothProfile](#)

An interface that represents a Bluetooth profile. A *Bluetooth profile* is a wireless interface specification for Bluetooth-based communication between devices. An example is the Hands-Free profile. For more discussion of profiles, see [Working with Profiles](#)

## [BluetoothHeadset](#)

Provides support for Bluetooth headsets to be used with mobile phones. This includes both Bluetooth Headset and Hands-Free (v1.5) profiles.

## [BluetoothA2dp](#)

Defines how high quality audio can be streamed from one device to another over a Bluetooth connection. "A2DP" stands for Advanced Audio Distribution Profile.

## [BluetoothHealth](#)

Represents a Health Device Profile proxy that controls the Bluetooth service.

## [BluetoothHealthCallback](#)

An abstract class that you use to implement [BluetoothHealth](#) callbacks. You must extend this class and implement the callback methods to receive updates about changes in the application's registration state and Bluetooth channel state.

## [BluetoothHealthAppConfiguration](#)

Represents an application configuration that the Bluetooth Health third-party application registers to communicate with a remote Bluetooth health device.

## [BluetoothProfile.ServiceListener](#)

An interface that notifies [BluetoothProfile](#) IPC clients when they have been connected to or disconnected from the service (that is, the internal service that runs a particular profile).

# Bluetooth Permissions

In order to use Bluetooth features in your application, you must declare the Bluetooth permission [BLUETOOTH](#). You need this permission to perform any Bluetooth communication, such as requesting a connection, accepting a connection, and transferring data.

If you want your app to initiate device discovery or manipulate Bluetooth settings, you must also declare the [BLUETOOTH\\_ADMIN](#) permission. Most applications need this permission solely for the ability to discover local Bluetooth devices. The other abilities granted by this permission should not be used, unless the application is a "power manager" that will modify Bluetooth settings upon user request. **Note:** If you use [BLUETOOTH\\_ADMIN](#) permission, then you must also have the [BLUETOOTH](#) permission.

Declare the Bluetooth permission(s) in your application manifest file. For example:

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

See the [<uses-permission>](#) reference for more information about declaring application permissions.

# Setting Up Bluetooth

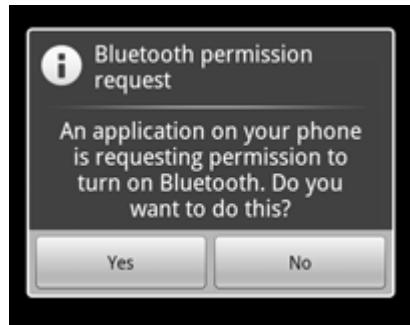


Figure 1: The enabling Bluetooth dialog.

Before your application can communicate over Bluetooth, you need to verify that Bluetooth is supported on the device, and if so, ensure that it is enabled.

If Bluetooth is not supported, then you should gracefully disable any Bluetooth features. If Bluetooth is supported, but disabled, then you can request that the user enable Bluetooth without leaving your application. This setup is accomplished in two steps, using the [BluetoothAdapter](#).

## 1. Get the [BluetoothAdapter](#)

The [BluetoothAdapter](#) is required for any and all Bluetooth activity. To get the [BluetoothAdapter](#), call the static [getDefaultAdapter\(\)](#) method. This returns a [BluetoothAdapter](#) that represents the device's own Bluetooth adapter (the Bluetooth radio). There's one Bluetooth adapter for the entire system, and your application can interact with it using this object. If [getDefaultAdapter\(\)](#) returns null, then the device does not support Bluetooth and your story ends here. For example:

```
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter()
if (mBluetoothAdapter == null) {
    // Device does not support Bluetooth
}
```

## 2. Enable Bluetooth

Next, you need to ensure that Bluetooth is enabled. Call [isEnabled\(\)](#) to check whether Bluetooth is currently enable. If this method returns false, then Bluetooth is disabled. To request that Bluetooth be enabled, call [startActivityForResult\(\)](#) with the [ACTION\\_REQUEST\\_ENABLE](#) action Intent. This will issue a request to enable Bluetooth through the system settings (without stopping your application). For example:

```
if (!mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

A dialog will appear requesting user permission to enable Bluetooth, as shown in Figure 1. If the user responds "Yes," the system will begin to enable Bluetooth and focus will return to your application once the process completes (or fails).

The [REQUEST\\_ENABLE\\_BT](#) constant passed to [startActivityForResult\(\)](#) is a locally defined integer (which must be greater than 0), that the system passes back to you in your [onActivityResult\(\)](#) implementation as the `requestCode` parameter.

If enabling Bluetooth succeeds, your activity receives the [RESULT\\_OK](#) result code in the [onActivityResult\(\)](#) callback. If Bluetooth was not enabled due to an error (or the user responded "No") then the result code is [RESULT\\_CANCELED](#).

Optionally, your application can also listen for the [ACTION\\_STATE\\_CHANGED](#) broadcast Intent, which the system will broadcast whenever the Bluetooth state has changed. This broadcast contains the extra fields [EXTRA\\_STATE](#) and [EXTRA\\_PREVIOUS\\_STATE](#), containing the new and old Bluetooth states, respectively. Possible values for these extra fields are [STATE\\_TURNING\\_ON](#), [STATE\\_ON](#), [STATE\\_TURNING\\_OFF](#), and [STATE\\_OFF](#). Listening for this broadcast can be useful to detect changes made to the Bluetooth state while your app is running.

**Tip:** Enabling discoverability will automatically enable Bluetooth. If you plan to consistently enable device discoverability before performing Bluetooth activity, you can skip step 2 above. Read about [enabling discoverability](#) below.

# Finding Devices

Using the [BluetoothAdapter](#), you can find remote Bluetooth devices either through device discovery or by querying the list of paired (bonded) devices.

Device discovery is a scanning procedure that searches the local area for Bluetooth enabled devices and then requesting some information about each one (this is sometimes referred to as "discovering," "inquiring" or "scanning"). However, a Bluetooth device within the local area will respond to a discovery request only if it is currently enabled to be discoverable. If a device is discoverable, it will respond to the discovery request by sharing some information, such as the device name, class, and its unique MAC address. Using this information, the device performing discovery can then choose to initiate a connection to the discovered device.

Once a connection is made with a remote device for the first time, a pairing request is automatically presented to the user. When a device is paired, the basic information about that device (such as the device name, class, and MAC address) is saved and can be read using the Bluetooth APIs. Using the known MAC address for a remote device, a connection can be initiated with it at any time without performing discovery (assuming the device is within range).

Remember there is a difference between being paired and being connected. To be paired means that two devices are aware of each other's existence, have a shared link-key that can be used for authentication, and are capable of establishing an encrypted connection with each other. To be connected means that the devices currently share an RFCOMM channel and are able to transmit data with each other. The current Android Bluetooth API's require devices to be paired before an RFCOMM connection can be established. (Pairing is automatically performed when you initiate an encrypted connection with the Bluetooth APIs.)

The following sections describe how to find devices that have been paired, or discover new devices using device discovery.

**Note:** Android-powered devices are not discoverable by default. A user can make the device discoverable for a limited time through the system settings, or an application can request that the user enable discoverability without leaving the application. How to [enable discoverability](#) is discussed below.

## Querying paired devices

Before performing device discovery, it's worth querying the set of paired devices to see if the desired device is already known. To do so, call [getBondedDevices \(\)](#). This will return a Set of [BluetoothDevice](#)s representing paired devices. For example, you can query all paired devices and then show the name of each device to the user, using an ArrayAdapter:

```
Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();  
// If there are paired devices  
if (pairedDevices.size() > 0) {  
    // Loop through paired devices  
    for (BluetoothDevice device : pairedDevices) {  
        // Add the name and address to an array adapter to show in a ListView  
        mArrayAdapter.add(device.getName() + "\n" + device.getAddress());  
    }  
}
```

All that's needed from the [BluetoothDevice](#) object in order to initiate a connection is the MAC address. In this example, it's saved as a part of an ArrayAdapter that's shown to the user. The MAC address can later be extracted in order to initiate the connection. You can learn more about creating a connection in the section about [Connecting Devices](#).

## Discovering devices

To start discovering devices, simply call `startDiscovery()`. The process is asynchronous and the method will immediately return with a boolean indicating whether discovery has successfully started. The discovery process usually involves an inquiry scan of about 12 seconds, followed by a page scan of each found device to retrieve its Bluetooth name.

Your application must register a BroadcastReceiver for the `ACTION_FOUND` Intent in order to receive information about each device discovered. For each device, the system will broadcast the `ACTION_FOUND` Intent. This Intent carries the extra fields `EXTRA_DEVICE` and `EXTRA_CLASS`, containing a `BluetoothDevice` and a `BluetoothClass`, respectively. For example, here's how you can register to handle the broadcast when devices are discovered:

```
// Create a BroadcastReceiver for ACTION_FOUND
private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        // When discovery finds a device
        if (BluetoothDevice.ACTION_FOUND.equals(action)) {
            // Get the BluetoothDevice object from the Intent
            BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            // Add the name and address to an array adapter to show in a ListView
            mArrayAdapter.add(device.getName() + "\n" + device.getAddress());
        }
    }
};

// Register the BroadcastReceiver
IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
registerReceiver(mReceiver, filter); // Don't forget to unregister during onDest...
```

All that's needed from the `BluetoothDevice` object in order to initiate a connection is the MAC address. In this example, it's saved as a part of an ArrayAdapter that's shown to the user. The MAC address can later be extracted in order to initiate the connection. You can learn more about creating a connection in the section about [Connecting Devices](#).

**Caution:** Performing device discovery is a heavy procedure for the Bluetooth adapter and will consume a lot of its resources. Once you have found a device to connect, be certain that you always stop discovery with `cancelDiscovery()` before attempting a connection. Also, if you already hold a connection with a device, then performing discovery can significantly reduce the bandwidth available for the connection, so you should not perform discovery while connected.

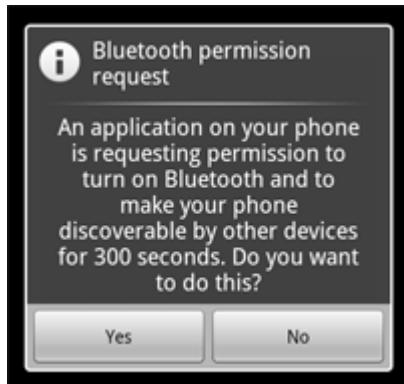
## Enabling discoverability

If you would like to make the local device discoverable to other devices, call `startActivityForResult(Intent, int)` with the `ACTION_REQUEST_DISCOVERABLE` action Intent. This will issue a request to enable discoverable mode through the system settings (without stopping your application). By default, the device will become discoverable for 120 seconds. You can define a different duration by adding the `EXTRA_DISCOVERABLE_DURATION` Intent extra. The maximum duration an app can set is 3600 seconds, and a value of 0 means the device is always discoverable. Any value below 0 or above 3600 is automatically set to 120 secs). For example, this snippet sets the duration to 300:

```

Intent discoverableIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
startActivity(discoverableIntent);

```



**Figure 2:** The enabling discoverability dialog.

A dialog will be displayed, requesting user permission to make the device discoverable, as shown in Figure 2. If the user responds "Yes," then the device will become discoverable for the specified amount of time. Your activity will then receive a call to the [onActivityResult\(\)](#) callback, with the result code equal to the duration that the device is discoverable. If the user responded "No" or if an error occurred, the result code will be [RESULT\\_CANCELED](#).

**Note:** If Bluetooth has not been enabled on the device, then enabling device discoverability will automatically enable Bluetooth.

The device will silently remain in discoverable mode for the allotted time. If you would like to be notified when the discoverable mode has changed, you can register a BroadcastReceiver for the [ACTION\\_SCAN\\_MODE\\_CHANGED](#) Intent. This will contain the extra fields [EXTRA\\_SCAN\\_MODE](#) and [EXTRA\\_PREVIOUS\\_SCAN\\_MODE](#), which tell you the new and old scan mode, respectively. Possible values for each are [SCAN\\_MODE\\_CONNECTABLE\\_DISCOVERABLE](#), [SCAN\\_MODE\\_CONNECTABLE](#), or [SCAN\\_MODE\\_NONE](#), which indicate that the device is either in discoverable mode, not in discoverable mode but still able to receive connections, or not in discoverable mode and unable to receive connections, respectively.

You do not need to enable device discoverability if you will be initiating the connection to a remote device. Enabling discoverability is only necessary when you want your application to host a server socket that will accept incoming connections, because the remote devices must be able to discover the device before it can initiate the connection.

## Connecting Devices

In order to create a connection between your application on two devices, you must implement both the server-side and client-side mechanisms, because one device must open a server socket and the other one must initiate the connection (using the server device's MAC address to initiate a connection). The server and client are considered connected to each other when they each have a connected [BluetoothSocket](#) on the same RF-COMM channel. At this point, each device can obtain input and output streams and data transfer can begin, which is discussed in the section about [Managing a Connection](#). This section describes how to initiate the connection between two devices.

The server device and the client device each obtain the required [BluetoothSocket](#) in different ways. The server will receive it when an incoming connection is accepted. The client will receive it when it opens an RF-COMM channel to the server.

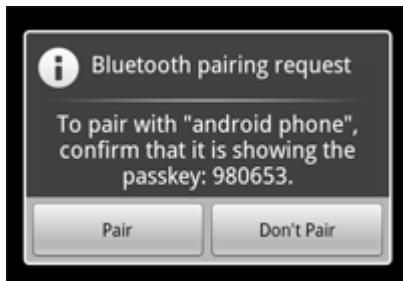


Figure 3: The Bluetooth pairing dialog.

One implementation technique is to automatically prepare each device as a server, so that each one has a server socket open and listening for connections. Then either device can initiate a connection with the other and become the client. Alternatively, one device can explicitly "host" the connection and open a server socket on demand and the other device can simply initiate the connection.

**Note:** If the two devices have not been previously paired, then the Android framework will automatically show a pairing request notification or dialog to the user during the connection procedure, as shown in Figure 3. So when attempting to connect devices, your application does not need to be concerned about whether or not the devices are paired. Your RFCOMM connection attempt will block until the user has successfully paired, or will fail if the user rejects pairing, or if pairing fails or times out.

## Connecting as a server

When you want to connect two devices, one must act as a server by holding an open [BluetoothServerSocket](#). The purpose of the server socket is to listen for incoming connection requests and when one is accepted, provide a connected [BluetoothSocket](#). When the [BluetoothSocket](#) is acquired from the [BluetoothServerSocket](#), the [BluetoothServerSocket](#) can (and should) be discarded, unless you want to accept more connections.

## About UUID

A Universally Unique Identifier (UUID) is a standardized 128-bit format for a string ID used to uniquely identify information. The point of a UUID is that it's big enough that you can select any random and it won't clash. In this case, it's used to uniquely identify your application's Bluetooth service. To get a UUID to use with your application, you can use one of the many random UUID generators on the web, then initialize a [UUID](#) with [fromString\(String\)](#).

Here's the basic procedure to set up a server socket and accept a connection:

1. Get a [BluetoothServerSocket](#) by calling the [listenUsingRfcommWithServiceRecord\(String, UUID\)](#).

The string is an identifiable name of your service, which the system will automatically write to a new Service Discovery Protocol (SDP) database entry on the device (the name is arbitrary and can simply be your application name). The UUID is also included in the SDP entry and will be the basis for the connection agreement with the client device. That is, when the client attempts to connect with this device, it will carry a UUID that uniquely identifies the service with which it wants to connect. These UUIDs must match in order for the connection to be accepted (in the next step).

2. Start listening for connection requests by calling [accept\(\)](#).

This is a blocking call. It will return when either a connection has been accepted or an exception has occurred. A connection is accepted only when a remote device has sent a connection request with a UUID matching the one registered with this listening server socket. When successful, [accept\(\)](#) will return a connected [BluetoothSocket](#).

3. Unless you want to accept additional connections, call [close\(\)](#).

This releases the server socket and all its resources, but does *not* close the connected [BluetoothSocket](#) that's been returned by [accept\(\)](#). Unlike TCP/IP, RFCOMM only allows one connected client per channel at a time, so in most cases it makes sense to call [close\(\)](#) on the [BluetoothServerSocket](#) immediately after accepting a connected socket.

The [accept\(\)](#) call should not be executed in the main activity UI thread because it is a blocking call and will prevent any other interaction with the application. It usually makes sense to do all work with a [BluetoothServerSocket](#) or [BluetoothSocket](#) in a new thread managed by your application. To abort a blocked call such as [accept\(\)](#), call [close\(\)](#) on the [BluetoothServerSocket](#) (or [BluetoothSocket](#)) from another thread and the blocked call will immediately return. Note that all methods on a [BluetoothServerSocket](#) or [BluetoothSocket](#) are thread-safe.

## Example

Here's a simplified thread for the server component that accepts incoming connections:

```
private class AcceptThread extends Thread {
    private final BluetoothServerSocket mmServerSocket;

    public AcceptThread() {
        // Use a temporary object that is later assigned to mmServerSocket,
        // because mmServerSocket is final
        BluetoothServerSocket tmp = null;
        try {
            // MY_UUID is the app's UUID string, also used by the client code
            tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_UUID);
        } catch (IOException e) { }
        mmServerSocket = tmp;
    }

    public void run() {
        BluetoothSocket socket = null;
        // Keep listening until exception occurs or a socket is returned
        while (true) {
            try {
                socket = mmServerSocket.accept();
            } catch (IOException e) {
                break;
            }
            // If a connection was accepted
            if (socket != null) {
                // Do work to manage the connection (in a separate thread)
                manageConnectedSocket(socket);
                mmServerSocket.close();
                break;
            }
        }
    }

    /** Will cancel the listening socket, and cause the thread to finish */
}
```

```

public void cancel() {
    try {
        mmServerSocket.close();
    } catch (IOException e) { }
}
}

```

In this example, only one incoming connection is desired, so as soon as a connection is accepted and the [BluetoothSocket](#) is acquired, the application sends the acquired [BluetoothSocket](#) to a separate thread, closes the [BluetoothServerSocket](#) and breaks the loop.

Note that when [accept\(\)](#) returns the [BluetoothSocket](#), the socket is already connected, so you should *not* call [connect\(\)](#) (as you do from the client-side).

[manageConnectedSocket\(\)](#) is a fictional method in the application that will initiate the thread for transferring data, which is discussed in the section about [Managing a Connection](#).

You should usually close your [BluetoothServerSocket](#) as soon as you are done listening for incoming connections. In this example, [close\(\)](#) is called as soon as the [BluetoothSocket](#) is acquired. You may also want to provide a public method in your thread that can close the private [BluetoothSocket](#) in the event that you need to stop listening on the server socket.

## Connecting as a client

In order to initiate a connection with a remote device (a device holding an open server socket), you must first obtain a [BluetoothDevice](#) object that represents the remote device. (Getting a [BluetoothDevice](#) is covered in the above section about [Finding Devices](#).) You must then use the [BluetoothDevice](#) to acquire a [BluetoothSocket](#) and initiate the connection.

Here's the basic procedure:

1. Using the [BluetoothDevice](#), get a [BluetoothSocket](#) by calling [createRfcommSocketToServiceRecord\(UUID\)](#).

This initializes a [BluetoothSocket](#) that will connect to the [BluetoothDevice](#). The UUID passed here must match the UUID used by the server device when it opened its [BluetoothServerSocket](#) (with [listenUsingRfcommWithServiceRecord\(String, UUID\)](#)). Using the same UUID is simply a matter of hard-coding the UUID string into your application and then referencing it from both the server and client code.

2. Initiate the connection by calling [connect\(\)](#).

Upon this call, the system will perform an SDP lookup on the remote device in order to match the UUID. If the lookup is successful and the remote device accepts the connection, it will share the RF-COMM channel to use during the connection and [connect\(\)](#) will return. This method is a blocking call. If, for any reason, the connection fails or the [connect\(\)](#) method times out (after about 12 seconds), then it will throw an exception.

Because [connect\(\)](#) is a blocking call, this connection procedure should always be performed in a thread separate from the main activity thread.

Note: You should always ensure that the device is not performing device discovery when you call [connect \(\)](#). If discovery is in progress, then the connection attempt will be significantly slowed and is more likely to fail.

## Example

Here is a basic example of a thread that initiates a Bluetooth connection:

```
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {
        // Use a temporary object that is later assigned to mmSocket,
        // because mmSocket is final
        BluetoothSocket tmp = null;
        mmDevice = device;

        // Get a BluetoothSocket to connect with the given BluetoothDevice
        try {
            // MY_UUID is the app's UUID string, also used by the server code
            tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
        } catch (IOException e) { }
        mmSocket = tmp;
    }

    public void run() {
        // Cancel discovery because it will slow down the connection
        mBluetoothAdapter.cancelDiscovery();

        try {
            // Connect the device through the socket. This will block
            // until it succeeds or throws an exception
            mmSocket.connect();
        } catch (IOException connectException) {
            // Unable to connect; close the socket and get out
            try {
                mmSocket.close();
            } catch (IOException closeException) { }
            return;
        }

        // Do work to manage the connection (in a separate thread)
        manageConnectedSocket(mmSocket);
    }

    /** Will cancel an in-progress connection, and close the socket */
    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) { }
    }
}
```

Notice that `cancelDiscovery()` is called before the connection is made. You should always do this before connecting and it is safe to call without actually checking whether it is running or not (but if you do want to check, call `isDiscovering()`).

`manageConnectedSocket()` is a fictional method in the application that will initiate the thread for transferring data, which is discussed in the section about [Managing a Connection](#).

When you're done with your `BluetoothSocket`, always call `close()` to clean up. Doing so will immediately close the connected socket and clean up all internal resources.

## Managing a Connection

When you have successfully connected two (or more) devices, each one will have a connected `BluetoothSocket`. This is where the fun begins because you can share data between devices. Using the `BluetoothSocket`, the general procedure to transfer arbitrary data is simple:

1. Get the `InputStream` and `OutputStream` that handle transmissions through the socket, via `getInputStream()` and `getOutputStream()`, respectively.
2. Read and write data to the streams with `read(byte[])` and `write(byte[])`.

That's it.

There are, of course, implementation details to consider. First and foremost, you should use a dedicated thread for all stream reading and writing. This is important because both `read(byte[])` and `write(byte[])` methods are blocking calls. `read(byte[])` will block until there is something to read from the stream. `write(byte[])` does not usually block, but can block for flow control if the remote device is not calling `read(byte[])` quickly enough and the intermediate buffers are full. So, your main loop in the thread should be dedicated to reading from the `InputStream`. A separate public method in the thread can be used to initiate writes to the `OutputStream`.

### Example

Here's an example of how this might look:

```
private class ConnectedThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final InputStream mmInStream;
    private final OutputStream mmOutStream;

    public ConnectedThread(BluetoothSocket socket) {
        mmSocket = socket;
        InputStream tmpIn = null;
        OutputStream tmpOut = null;

        // Get the input and output streams, using temp objects because
        // member streams are final
        try {
            tmpIn = socket.getInputStream();
            tmpOut = socket.getOutputStream();
        } catch (IOException e) { }

        mmInStream = tmpIn;
        mmOutStream = tmpOut;
    }

    public void run() {
        byte[] buffer = new byte[1024];
        int bytes;
        try {
            while (true) {
                bytes = mmInStream.read(buffer);
                if (bytes > 0) {
                    // Process the data here.
                }
            }
        } catch (IOException e) {
            Log.e("BluetoothChat", "Error during read", e);
        }
    }

    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) {
            Log.e("BluetoothChat", "Error closing socket", e);
        }
    }
}
```

```

        mmOutStream = tmpOut;
    }

    public void run() {
        byte[] buffer = new byte[1024]; // buffer store for the stream
        int bytes; // bytes returned from read()

        // Keep listening to the InputStream until an exception occurs
        while (true) {
            try {
                // Read from the InputStream
                bytes = mmInStream.read(buffer);
                // Send the obtained bytes to the UI activity
                mHandler.obtainMessage(MESSAGE_READ, bytes, -1, buffer)
                    .sendToTarget();
            } catch (IOException e) {
                break;
            }
        }
    }

    /* Call this from the main activity to send data to the remote device */
    public void write(byte[] bytes) {
        try {
            mmOutStream.write(bytes);
        } catch (IOException e) { }
    }

    /* Call this from the main activity to shutdown the connection */
    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) { }
    }
}

```

The constructor acquires the necessary streams and once executed, the thread will wait for data to come through the `InputStream`. When [read\(byte\[\]\)](#) returns with bytes from the stream, the data is sent to the main activity using a member Handler from the parent class. Then it goes back and waits for more bytes from the stream.

Sending outgoing data is as simple as calling the thread's `write()` method from the main activity and passing in the bytes to be sent. This method then simply calls [write\(byte\[\]\)](#) to send the data to the remote device.

The thread's `cancel()` method is important so that the connection can be terminated at any time by closing the [BluetoothSocket](#). This should always be called when you're done using the Bluetooth connection.

For a demonstration of using the Bluetooth APIs, see the [Bluetooth Chat sample app](#).

## Working with Profiles

Starting in Android 3.0, the Bluetooth API includes support for working with Bluetooth profiles. A *Bluetooth profile* is a wireless interface specification for Bluetooth-based communication between devices. An example is

the Hands-Free profile. For a mobile phone to connect to a wireless headset, both devices must support the Hands-Free profile.

You can implement the interface [BluetoothProfile](#) to write your own classes to support a particular Bluetooth profile. The Android Bluetooth API provides implementations for the following Bluetooth profiles:

- **Headset.** The Headset profile provides support for Bluetooth headsets to be used with mobile phones. Android provides the [BluetoothHeadset](#) class, which is a proxy for controlling the Bluetooth Headset Service via interprocess communication ([IPC](#)). This includes both Bluetooth Headset and Hands-Free (v1.5) profiles. The [BluetoothHeadset](#) class includes support for AT commands. For more discussion of this topic, see [Vendor-specific AT commands](#)
- **A2DP.** The Advanced Audio Distribution Profile (A2DP) profile defines how high quality audio can be streamed from one device to another over a Bluetooth connection. Android provides the [BluetoothA2dp](#) class, which is a proxy for controlling the Bluetooth A2DP Service via IPC.
- **Health Device.** Android 4.0 (API level 14) introduces support for the Bluetooth Health Device Profile (HDP). This lets you create applications that use Bluetooth to communicate with health devices that support Bluetooth, such as heart-rate monitors, blood meters, thermometers, scales, and so on. For a list of supported devices and their corresponding device data specialization codes, refer to **Bluetooth Assigned Numbers** at [www.bluetooth.org](http://www.bluetooth.org). Note that these values are also referenced in the ISO/IEEE 11073-20601 [7] specification as MDC\_DEV\_SPEC\_PROFILE\_\* in the Nomenclature Codes Annex. For more discussion of HDP, see [Health Device Profile](#).

Here are the basic steps for working with a profile:

1. Get the default adapter, as described in [Setting Up Bluetooth](#).
2. Use [getProfileProxy\(\)](#) to establish a connection to the profile proxy object associated with the profile. In the example below, the profile proxy object is an instance of [BluetoothHeadset](#).
3. Set up a [BluetoothProfile.ServiceListener](#). This listener notifies [BluetoothProfile](#) IPC clients when they have been connected to or disconnected from the service.
4. In [onServiceConnected\(\)](#), get a handle to the profile proxy object.
5. Once you have the profile proxy object, you can use it to monitor the state of the connection and perform other operations that are relevant to that profile.

For example, this code snippet shows how to connect to a [BluetoothHeadset](#) proxy object so that you can control the Headset profile:

```
BluetoothHeadset mBluetoothHeadset;

// Get the default adapter
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

// Establish connection to the proxy.
mBluetoothAdapter.getProfileProxy(context, mProfileListener, BluetoothProfile.H

private BluetoothProfile.ServiceListener mProfileListener = new BluetoothProfile
    public void onServiceConnected(int profile, BluetoothProfile proxy) {
        if (profile == BluetoothProfile.HEADSET) {
            mBluetoothHeadset = (BluetoothHeadset) proxy;
        }
    }
    public void onServiceDisconnected(int profile) {
        if (profile == BluetoothProfile.HEADSET) {
            mBluetoothHeadset = null;
        }
    }
}
```

```

        }
    } ;

// ... call functions on mBluetoothHeadset

// Close proxy connection after use.
mBluetoothAdapter.closeProfileProxy(mBluetoothHeadset);

```

## Vendor-specific AT commands

Starting in Android 3.0, applications can register to receive system broadcasts of pre-defined vendor-specific AT commands sent by headsets (such as a Plantronics +XEVENT command). For example, an application could receive broadcasts that indicate a connected device's battery level and could notify the user or take other action as needed. Create a broadcast receiver for the [ACTION\\_VENDOR\\_SPECIFIC\\_HEADSET\\_EVENT](#) intent to handle vendor-specific AT commands for the headset.

## Health Device Profile

Android 4.0 (API level 14) introduces support for the Bluetooth Health Device Profile (HDP). This lets you create applications that use Bluetooth to communicate with health devices that support Bluetooth, such as heart-rate monitors, blood meters, thermometers, and scales. The Bluetooth Health API includes the classes [BluetoothHealth](#), [BluetoothHealthCallback](#), and [BluetoothHealthAppConfiguration](#), which are described in [The Basics](#).

In using the Bluetooth Health API, it's helpful to understand these key HDP concepts:

Concept	Description
<b>Source</b>	A role defined in HDP. A <i>source</i> is a health device that transmits medical data (weight scale, glucose meter, thermometer, etc.) to a smart device such as an Android phone or tablet.
<b>Sink</b>	A role defined in HDP. In HDP, a <i>sink</i> is the smart device that receives the medical data. In an Android HDP application, the sink is represented by a <a href="#">BluetoothHealthAppConfiguration</a> object.
<b>Registration</b>	Refers to registering a sink for a particular health device.
<b>Connection</b>	Refers to opening a channel between a health device and a smart device such as an Android phone or tablet.

## Creating an HDP Application

Here are the basic steps involved in creating an Android HDP application:

1. Get a reference to the [BluetoothHealth](#) proxy object.

Similar to regular headset and A2DP profile devices, you must call [getProfileProxy\(\)](#) with a [BluetoothProfile.ServiceListener](#) and the [HEALTH](#) profile type to establish a connection with the profile proxy object.

2. Create a [BluetoothHealthCallback](#) and register an application configuration ([BluetoothHealthAppConfiguration](#)) that acts as a health sink.
3. Establish a connection to a health device. Some devices will initiate the connection. It is unnecessary to carry out this step for those devices.
4. When connected successfully to a health device, read/write to the health device using the file descriptor.

The received data needs to be interpreted using a health manager which implements the IEEE 11073-xxxxx specifications.

5. When done, close the health channel and unregister the application. The channel also closes when there is extended inactivity.

For a complete code sample that illustrates these steps, see [Bluetooth HDP \(Health Device Profile\)](#).

# Bluetooth Low Energy

## In this document

1. [Key Terms and Concepts](#)
  1. [Roles and Responsibilities](#)
2. [BLE Permissions](#)
3. [Setting Up BLE](#)
4. [Finding BLE Devices](#)
5. [Connecting to a GATT Server](#)
6. [Reading BLE Attributes](#)
7. [Receiving GATT Notifications](#)
8. [Closing the Client App](#)

## Key classes

1. [BluetoothGatt](#)
2. [BluetoothGattCallback](#)
3. [BluetoothGattCharacteristic](#)
4. [BluetoothGattService](#)

## Related samples

1. [Bluetooth LE sample](#)

## See Also

1. [Best Practices for Bluetooth Development](#) (video)

Android 4.3 (API Level 18) introduces built-in platform support for Bluetooth Low Energy in the *central role* and provides APIs that apps can use to discover devices, query for services, and read/write characteristics. In contrast to [Classic Bluetooth](#), Bluetooth Low Energy (BLE) is designed to provide significantly lower power consumption. This allows Android apps to communicate with BLE devices that have low power requirements, such as proximity sensors, heart rate monitors, fitness devices, and so on.

## Key Terms and Concepts

Here is a summary of key BLE terms and concepts:

- **Generic Attribute Profile (GATT)**—The GATT profile is a general specification for sending and receiving short pieces of data known as "attributes" over a BLE link. All current Low Energy application profiles are based on GATT.
  - The Bluetooth SIG defines many [profiles](#) for Low Energy devices. A profile is a specification for how a device works in a particular application. Note that a device can implement more than one profile. For example, a device could contain a heart rate monitor and a battery level detector.
- **Attribute Protocol (ATT)**—GATT is built on top of the Attribute Protocol (ATT). This is also referred to as GATT/ATT. ATT is optimized to run on BLE devices. To this end, it uses as few bytes as possible. Each attribute is uniquely identified by a Universally Unique Identifier (UUID), which is a standard-

ized 128-bit format for a string ID used to uniquely identify information. The *attributes* transported by ATT are formatted as *characteristics* and *services*.

- **Characteristic**—A characteristic contains a single value and 0-n descriptors that describe the characteristic's value. A characteristic can be thought of as a type, analogous to a class.
- **Descriptor**—Descriptors are defined attributes that describe a characteristic value. For example, a descriptor might specify a human-readable description, an acceptable range for a characteristic's value, or a unit of measure that is specific to a characteristic's value.
- **Service**—A service is a collection of characteristics. For example, you could have a service called "Heart Rate Monitor" that includes characteristics such as "heart rate measurement." You can find a list of existing GATT-based profiles and services on [bluetooth.org](https://bluetooth.org).

## Roles and Responsibilities

Here are the roles and responsibilities that apply when an Android device interacts with a BLE device:

- Central vs. peripheral. This applies to the BLE connection itself. The device in the central role scans, looking for advertisement, and the device in the peripheral role makes the advertisement.
- GATT server vs. GATT client. This determines how two devices talk to each other once they've established the connection.

To understand the distinction, imagine that you have an Android phone and an activity tracker that is a BLE device. The phone supports the central role; the activity tracker supports the peripheral role (to establish a BLE connection you need one of each—two things that only support peripheral couldn't talk to each other, nor could two things that only support central).

Once the phone and the activity tracker have established a connection, they start transferring GATT metadata to one another. Depending on the kind of data they transfer, one or the other might act as the server. For example, if the activity tracker wants to report sensor data to the phone, it might make sense for the activity tracker to act as the server. If the activity tracker wants to receive updates from the phone, then it might make sense for the phone to act as the server.

In the example used in this document, the Android app (running on an Android device) is the GATT client. The app gets data from the GATT server, which is a BLE heart rate monitor that supports the [Heart Rate Profile](#). But you could alternatively design your Android app to play the GATT server role. See [BluetoothGattServer](#) for more information.

## BLE Permissions

In order to use Bluetooth features in your application, you must declare the Bluetooth permission [BLUETOOTH](#). You need this permission to perform any Bluetooth communication, such as requesting a connection, accepting a connection, and transferring data.

If you want your app to initiate device discovery or manipulate Bluetooth settings, you must also declare the [BLUETOOTH\\_ADMIN](#) permission. **Note:** If you use the [BLUETOOTH\\_ADMIN](#) permission, then you must also have the [BLUETOOTH](#) permission.

Declare the Bluetooth permission(s) in your application manifest file. For example:

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

If you want to declare that your app is available to BLE-capable devices only, include the following in your app's manifest:

```
<uses-feature android:name="android.hardware.bluetooth_le" android:required="true"/>
```

However, if you want to make your app available to devices that don't support BLE, you should still include this element in your app's manifest, but set `required="false"`. Then at run-time you can determine BLE availability by using [PackageManager.hasSystemFeature\(\)](#):

```
// Use this check to determine whether BLE is supported on the device. Then
// you can selectively disable BLE-related features.
if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE))
    Toast.makeText(this, R.string.ble_not_supported, Toast.LENGTH_SHORT).show()
    finish();
}
```

## Setting Up BLE

Before your application can communicate over BLE, you need to verify that BLE is supported on the device, and if so, ensure that it is enabled. Note that this check is only necessary if `<uses-feature.../>` is set to false.

If BLE is not supported, then you should gracefully disable any BLE features. If BLE is supported, but disabled, then you can request that the user enable Bluetooth without leaving your application. This setup is accomplished in two steps, using the [BluetoothAdapter](#).

### 1. Get the [BluetoothAdapter](#)

The [BluetoothAdapter](#) is required for any and all Bluetooth activity. The [BluetoothAdapter](#) represents the device's own Bluetooth adapter (the Bluetooth radio). There's one Bluetooth adapter for the entire system, and your application can interact with it using this object. The snippet below shows how to get the adapter. Note that this approach uses [getSystemService\(\)](#) to return an instance of [BluetoothManager](#), which is then used to get the adapter. Android 4.3 (API Level 18) introduces [BluetoothManager](#):

```
// Initializes Bluetooth adapter.
final BluetoothManager bluetoothManager =
    (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
mBluetoothAdapter = bluetoothManager.getAdapter();
```

### 2. Enable Bluetooth

Next, you need to ensure that Bluetooth is enabled. Call [isEnabled\(\)](#) to check whether Bluetooth is currently enabled. If this method returns false, then Bluetooth is disabled. The following snippet checks whether Bluetooth is enabled. If it isn't, the snippet displays an error prompting the user to go to Settings to enable Bluetooth:

```
private BluetoothAdapter mBluetoothAdapter;
...
// Ensures Bluetooth is available on the device and it is enabled. If not
// displays a dialog requesting user permission to enable Bluetooth.
if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

# Finding BLE Devices

To find BLE devices, you use the [startLeScan\(\)](#) method. This method takes a [Blue-toothAdapter.LeScanCallback](#) as a parameter. You must implement this callback, because that is how scan results are returned. Because scanning is battery-intensive, you should observe the following guidelines:

- As soon as you find the desired device, stop scanning.
- Never scan on a loop, and set a time limit on your scan. A device that was previously available may have moved out of range, and continuing to scan drains the battery.

The following snippet shows how to start and stop a scan:

```
/**  
 * Activity for scanning and displaying available BLE devices.  
 */  
public class DeviceScanActivity extends ListActivity {  
  
    private BluetoothAdapter mBluetoothAdapter;  
    private boolean mScanning;  
    private Handler mHandler;  
  
    // Stops scanning after 10 seconds.  
    private static final long SCAN_PERIOD = 10000;  
    ...  
    private void scanLeDevice(final boolean enable) {  
        if (enable) {  
            // Stops scanning after a pre-defined scan period.  
            mHandler.postDelayed(new Runnable() {  
                @Override  
                public void run() {  
                    mScanning = false;  
                    mBluetoothAdapter.stopLeScan(mLeScanCallback);  
                }  
            }, SCAN_PERIOD);  
  
            mScanning = true;  
            mBluetoothAdapter.startLeScan(mLeScanCallback);  
        } else {  
            mScanning = false;  
            mBluetoothAdapter.stopLeScan(mLeScanCallback);  
        }  
        ...  
    }  
    ...  
}
```

If you want to scan for only specific types of peripherals, you can instead call [startLeScan\(UUID\[\], BluetoothAdapter.LeScanCallback\)](#), providing an array of [UUID](#) objects that specify the GATT services your app supports.

Here is an implementation of the [BluetoothAdapter.LeScanCallback](#), which is the interface used to deliver BLE scan results:

```

private LeDeviceListAdapter mLeDeviceListAdapter;
...
// Device scan callback.
private BluetoothAdapter.LeScanCallback mLeScanCallback =
    new BluetoothAdapter.LeScanCallback() {
        @Override
        public void onLeScan(final BluetoothDevice device, int rssi,
            byte[] scanRecord) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    mLeDeviceListAdapter.addDevice(device);
                    mLeDeviceListAdapter.notifyDataSetChanged();
                }
            });
        }
    };
}

```

**Note:** You can only scan for Bluetooth LE devices *or* scan for Classic Bluetooth devices, as described in [Bluetooth](#). You cannot scan for both Bluetooth LE and classic devices at the same time.

## Connecting to a GATT Server

The first step in interacting with a BLE device is connecting to it— more specifically, connecting to the GATT server on the device. To connect to a GATT server on a BLE device, you use the [connectGatt\(\)](#) method. This method takes three parameters: a [Context](#) object, autoConnect (boolean indicating whether to automatically connect to the BLE device as soon as it becomes available), and a reference to a [BluetoothGattCallback](#):

```
mBluetoothGatt = device.connectGatt(this, false, mGattCallback);
```

This connects to the GATT server hosted by the BLE device, and returns a [BluetoothGatt](#) instance, which you can then use to conduct GATT client operations. The caller (the Android app) is the GATT client. The [BluetoothGattCallback](#) is used to deliver results to the client, such as connection status, as well as any further GATT client operations.

In this example, the BLE app provides an activity ([DeviceControlActivity](#)) to connect, display data, and display GATT services and characteristics supported by the device. Based on user input, this activity communicates with a [Service](#) called [BluetoothLeService](#), which interacts with the BLE device via the Android BLE API:

```

// A service that interacts with the BLE device via the Android BLE API.
public class BluetoothLeService extends Service {
    private final static String TAG = BluetoothLeService.class.getSimpleName();

    private BluetoothManager mBluetoothManager;
    private BluetoothAdapter mBluetoothAdapter;
    private String mBluetoothDeviceAddress;
    private BluetoothGatt mBluetoothGatt;
    private int mConnectionState = STATE_DISCONNECTED;

    private static final int STATE_DISCONNECTED = 0;
    private static final int STATE_CONNECTING = 1;

```

```
private static final int STATE_CONNECTED = 2;

public final static String ACTION_GATT_CONNECTED =
        "com.example.bluetooth.le.ACTION_GATT_CONNECTED";
public final static String ACTION_GATT_DISCONNECTED =
        "com.example.bluetooth.le.ACTION_GATT_DISCONNECTED";
public final static String ACTION_GATT_SERVICES_DISCOVERED =
        "com.example.bluetooth.le.ACTION_GATT_SERVICES_DISCOVERED";
public final static String ACTION_DATA_AVAILABLE =
        "com.example.bluetooth.le.ACTION_DATA_AVAILABLE";
public final static String EXTRA_DATA =
        "com.example.bluetooth.le.EXTRA_DATA";

public final static UUID UUID_HEART_RATE_MEASUREMENT =
        UUID.fromString(SampleGattAttributes.HEART_RATE_MEASUREMENT);

// Various callback methods defined by the BLE API.
private final BluetoothGattCallback mGattCallback =
        new BluetoothGattCallback() {
    @Override
    public void onConnectionStateChange(BluetoothGatt gatt, int status,
            int newState) {
        String intentAction;
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            intentAction = ACTION_GATT_CONNECTED;
            mConnectionState = STATE_CONNECTED;
            broadcastUpdate(intentAction);
            Log.i(TAG, "Connected to GATT server.");
            Log.i(TAG, "Attempting to start service discovery:" +
                    mBluetoothGatt.discoverServices());
        } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
            intentAction = ACTION_GATT_DISCONNECTED;
            mConnectionState = STATE_DISCONNECTED;
            Log.i(TAG, "Disconnected from GATT server.");
            broadcastUpdate(intentAction);
        }
    }

    @Override
    // New services discovered
    public void onServicesDiscovered(BluetoothGatt gatt, int status) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(ACTION_GATT_SERVICES_DISCOVERED);
        } else {
            Log.w(TAG, "onServicesDiscovered received: " + status);
        }
    }

    @Override
    // Result of a characteristic read operation
    public void onCharacteristicRead(BluetoothGatt gatt,
            BluetoothGattCharacteristic characteristic,
            int status) {
```

```

        if (status == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
        }
    }
    ...
};

}

```

When a particular callback is triggered, it calls the appropriate `broadcastUpdate()` helper method and passes it an action. Note that the data parsing in this section is performed in accordance with the Bluetooth Heart Rate Measurement [profile specifications](#):

```

private void broadcastUpdate(final String action) {
    final Intent intent = new Intent(action);
    sendBroadcast(intent);
}

private void broadcastUpdate(final String action,
                           final BluetoothGattCharacteristic characteristic)
final Intent intent = new Intent(action);

// This is special handling for the Heart Rate Measurement profile. Data
// parsing is carried out as per profile specifications.
if (UUID_HEART_RATE_MEASUREMENT.equals(characteristic.getUuid())) {
    int flag = characteristic.getProperties();
    int format = -1;
    if ((flag & 0x01) != 0) {
        format = BluetoothGattCharacteristic.FORMAT_UINT16;
        Log.d(TAG, "Heart rate format UINT16.");
    } else {
        format = BluetoothGattCharacteristic.FORMAT_UINT8;
        Log.d(TAG, "Heart rate format UINT8.");
    }
    final int heartRate = characteristic.getIntValue(format, 1);
    Log.d(TAG, String.format("Received heart rate: %d", heartRate));
    intent.putExtra(EXTRA_DATA, String.valueOf(heartRate));
} else {
    // For all other profiles, writes the data formatted in HEX.
    final byte[] data = characteristic.getValue();
    if (data != null && data.length > 0) {
        final StringBuilder stringBuilder = new StringBuilder(data.length);
        for(byte byteChar : data)
            stringBuilder.append(String.format("%02X ", byteChar));
        intent.putExtra(EXTRA_DATA, new String(data) + "\n" +
                       stringBuilder.toString());
    }
}
sendBroadcast(intent);
}

```

Back in `DeviceControlActivity`, these events are handled by a [BroadcastReceiver](#):

```

// Handles various events fired by the Service.
// ACTION_GATT_CONNECTED: connected to a GATT server.
// ACTION_GATT_DISCONNECTED: disconnected from a GATT server.
// ACTION_GATT_SERVICES_DISCOVERED: discovered GATT services.
// ACTION_DATA_AVAILABLE: received data from the device. This can be a
// result of read or notification operations.
private final BroadcastReceiver mGattUpdateReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();
        if (BluetoothLeService.ACTION_GATT_CONNECTED.equals(action)) {
            mConnected = true;
            updateConnectionState(R.string.connected);
            invalidateOptionsMenu();
        } else if (BluetoothLeService.ACTION_GATT_DISCONNECTED.equals(action)) {
            mConnected = false;
            updateConnectionState(R.string.disconnected);
            invalidateOptionsMenu();
            clearUI();
        } else if (BluetoothLeService.
                    ACTION_GATT_SERVICES_DISCOVERED.equals(action)) {
            // Show all the supported services and characteristics on the
            // user interface.
            displayGattServices(mBluetoothLeService.getSupportedGattServices());
        } else if (BluetoothLeService.ACTION_DATA_AVAILABLE.equals(action)) {
            displayData(intent.getStringExtra(BluetoothLeService.EXTRA_DATA));
        }
    }
};

```

## Reading BLE Attributes

Once your Android app has connected to a GATT server and discovered services, it can read and write attributes, where supported. For example, this snippet iterates through the server's services and characteristics and displays them in the UI:

```

public class DeviceControlActivity extends Activity {
    ...
    // Demonstrates how to iterate through the supported GATT
    // Services/Characteristics.
    // In this sample, we populate the data structure that is bound to the
    // ExpandableListView on the UI.
    private void displayGattServices(List<BluetoothGattService> gattServices) {
        if (gattServices == null) return;
        String uuid = null;
        String unknownServiceString = getResources().
            getString(R.string.unknown_service);
        String unknownCharaString = getResources().
            getString(R.string.unknown_characteristic);
        ArrayList<HashMap<String, String>> gattServiceData =
            new ArrayList<HashMap<String, String>>();
        ArrayList<ArrayList<HashMap<String, String>>> gattCharacteristicData
            = new ArrayList<ArrayList<HashMap<String, String>>>();
        mGattCharacteristics =

```

```

        new ArrayList<ArrayList<BluetoothGattCharacteristic>>();

    // Loops through available GATT Services.
    for (BluetoothGattService gattService : gattServices) {
        HashMap<String, String> currentServiceData =
            new HashMap<String, String>();
        uuid = gattService.getUuid().toString();
        currentServiceData.put(
            LIST_NAME, SampleGattAttributes.
                lookup(uuid, unknownServiceString));
        currentServiceData.put(LIST_UUID, uuid);
        gattServiceData.add(currentServiceData);

        ArrayList<HashMap<String, String>> gattCharacteristicGroupData =
            new ArrayList<HashMap<String, String>>();
        List<BluetoothGattCharacteristic> gattCharacteristics =
            gattService.getCharacteristics();
        ArrayList<BluetoothGattCharacteristic> charas =
            new ArrayList<BluetoothGattCharacteristic>();
        // Loops through available Characteristics.
        for (BluetoothGattCharacteristic gattCharacteristic :
            gattCharacteristics) {
            charas.add(gattCharacteristic);
            HashMap<String, String> currentCharaData =
                new HashMap<String, String>();
            uuid = gattCharacteristic.getUuid().toString();
            currentCharaData.put(
                LIST_NAME, SampleGattAttributes.lookup(uuid,
                    unknownCharaString));
            currentCharaData.put(LIST_UUID, uuid);
            gattCharacteristicGroupData.add(currentCharaData);
        }
        mGattCharacteristics.add(charas);
        gattCharacteristicData.add(gattCharacteristicGroupData);
    }
}

...
}

}

```

## Receiving GATT Notifications

It's common for BLE apps to ask to be notified when a particular characteristic changes on the device. This snippet shows how to set a notification for a characteristic, using the [setCharacteristicNotification\(\)](#) method:

```

private BluetoothGatt mBluetoothGatt;
BluetoothGattCharacteristic characteristic;
boolean enabled;
...
mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);
...
BluetoothGattDescriptor descriptor = characteristic.getDescriptor(

```

```
UUID.fromString(SampleGattAttributes.CLIENT_CHARACTERISTIC_CONFIG));  
descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);  
mBluetoothGatt.writeDescriptor(descriptor);
```

Once notifications are enabled for a characteristic, an [onCharacteristicChanged\(\)](#) callback is triggered if the characteristic changes on the remote device:

```
@Override  
// Characteristic notification  
public void onCharacteristicChanged(BluetoothGatt gatt,  
        BluetoothGattCharacteristic characteristic) {  
    broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);  
}
```

## Closing the Client App

Once your app has finished using a BLE device, it should call [close\(\)](#) so the system can release resources appropriately:

```
public void close() {  
    if (mBluetoothGatt == null) {  
        return;  
    }  
    mBluetoothGatt.close();  
    mBluetoothGatt = null;  
}
```

# Near Field Communication

Near Field Communication (NFC) is a set of short-range wireless technologies, typically requiring a distance of 4cm or less to initiate a connection. NFC allows you to share small payloads of data between an NFC tag and an Android-powered device, or between two Android-powered devices.

Tags can range in complexity. Simple tags offer just read and write semantics, sometimes with one-time-programmable areas to make the card read-only. More complex tags offer math operations, and have cryptographic hardware to authenticate access to a sector. The most sophisticated tags contain operating environments, allowing complex interactions with code executing on the tag. The data stored in the tag can also be written in a variety of formats, but many of the Android framework APIs are based around a [NFC Forum](#) standard called NDEF (NFC Data Exchange Format).

## [NFC Basics](#)

This document describes how Android handles discovered NFC tags and how it notifies applications of data that is relevant to the application. It also goes over how to work with the NDEF data in your applications and gives an overview of the framework APIs that support the basic NFC feature set of Android.

## [Advanced NFC](#)

This document goes over the APIs that enable use of the various tag technologies that Android supports. When you are not working with NDEF data, or when you are working with NDEF data that Android cannot fully understand, you have to manually read or write to the tag in raw bytes using your own protocol stack. In these cases, Android provides support to detect certain tag technologies and to open communication with the tag using your own protocol stack.

# NFC Basics

## In this document

1. [The Tag Dispatch System](#)
  1. [How NFC tags are mapped to MIME types and URIs](#)
  2. [How NFC Tags are Dispatched to Applications](#)
2. [Requesting NFC Access in the Android Manifest](#)
3. [Filtering for Intents](#)
  1. [ACTION\\_NDEF\\_DISCOVERED](#)
  2. [ACTION\\_TECH\\_DISCOVERED](#)
  3. [ACTION\\_TAG\\_DISCOVERED](#)
  4. [Obtaining information from intents](#)
4. [Creating Common Types of NDEF Records](#)
  1. [TNF\\_ABSOLUTE\\_URI](#)
  2. [TNF\\_MIME\\_MEDIA](#)
  3. [TNF\\_WELL\\_KNOWN with RTD\\_TEXT](#)
  4. [TNF\\_WELL\\_KNOWN with RTD\\_URI](#)
  5. [TNF\\_EXTERNAL\\_TYPE](#)
  6. [Android Application Records](#)
5. [Beaming NDEF Messages to Other Devices](#)

This document describes the basic NFC tasks you perform in Android. It explains how to send and receive NFC data in the form of NDEF messages and describes the Android framework APIs that support these features. For more advanced topics, including a discussion of working with non-NDEF data, see [Advanced NFC](#).

There are two major uses cases when working with NDEF data and Android:

- Reading NDEF data from an NFC tag
- Beaming NDEF messages from one device to another with [Android Beam™](#)

Reading NDEF data from an NFC tag is handled with the [tag dispatch system](#), which analyzes discovered NFC tags, appropriately categorizes the data, and starts an application that is interested in the categorized data. An application that wants to handle the scanned NFC tag can [declare an intent filter](#) and request to handle the data.

The Android Beam™ feature allows a device to push an NDEF message onto another device by physically tapping the devices together. This interaction provides an easier way to send data than other wireless technologies like Bluetooth, because with NFC, no manual device discovery or pairing is required. The connection is automatically started when two devices come into range. Android Beam is available through a set of NFC APIs, so any application can transmit information between devices. For example, the Contacts, Browser, and YouTube applications use Android Beam to share contacts, web pages, and videos with other devices.

## The Tag Dispatch System

Android-powered devices are usually looking for NFC tags when the screen is unlocked, unless NFC is disabled in the device's Settings menu. When an Android-powered device discovers an NFC tag, the desired behavior is to have the most appropriate activity handle the intent without asking the user what application to use. Because devices scan NFC tags at a very short range, it is likely that making users manually select an activity would force them to move the device away from the tag and break the connection. You should develop your activity to only handle the NFC tags that your activity cares about to prevent the Activity Chooser from appearing.

To help you with this goal, Android provides a special tag dispatch system that analyzes scanned NFC tags, parses them, and tries to locate applications that are interested in the scanned data. It does this by:

1. Parsing the NFC tag and figuring out the MIME type or a URI that identifies the data payload in the tag.
2. Encapsulating the MIME type or URI and the payload into an intent. These first two steps are described in [How NFC tags are mapped to MIME types and URIs](#).
3. Starts an activity based on the intent. This is described in [How NFC Tags are Dispatched to Applications](#).

## How NFC tags are mapped to MIME types and URIs

Before you begin writing your NFC applications, it is important to understand the different types of NFC tags, how the tag dispatch system parses NFC tags, and the special work that the tag dispatch system does when it detects an NDEF message. NFC tags come in a wide array of technologies and can also have data written to them in many different ways. Android has the most support for the NDEF standard, which is defined by the [NFC Forum](#).

NDEF data is encapsulated inside a message ([NdefMessage](#)) that contains one or more records ([NdefRecord](#)). Each NDEF record must be well-formed according to the specification of the type of record that you want to create. Android also supports other types of tags that do not contain NDEF data, which you can work with by using the classes in the [android.nfc.tech](#) package. To learn more about these technologies, see the [Advanced NFC](#) topic. Working with these other types of tags involves writing your own protocol stack to communicate with the tags, so we recommend using NDEF when possible for ease of development and maximum support for Android-powered devices.

**Note:** To download complete NDEF specifications, go to the [NFC Forum Specification Download](#) site and see [Creating common types of NDEF records](#) for examples of how to construct NDEF records.

Now that you have some background in NFC tags, the following sections describe in more detail how Android handles NDEF formatted tags. When an Android-powered device scans an NFC tag containing NDEF formatted data, it parses the message and tries to figure out the data's MIME type or identifying URI. To do this, the system reads the first [NdefRecord](#) inside the [NdefMessage](#) to determine how to interpret the entire NDEF message (an NDEF message can have multiple NDEF records). In a well-formed NDEF message, the first [NdefRecord](#) contains the following fields:

### 3-bit TNF (Type Name Format)

Indicates how to interpret the variable length type field. Valid values are described in described in [Table 1](#).

### Variable length type

Describes the type of the record. If using [TNF WELL KNOWN](#), use this field to specify the Record Type Definition (RTD). Valid RTD values are described in [Table 2](#).

### Variable length ID

A unique identifier for the record. This field is not used often, but if you need to uniquely identify a tag, you can create an ID for it.

### Variable length payload

The actual data payload that you want to read or write. An NDEF message can contain multiple NDEF records, so don't assume the full payload is in the first NDEF record of the NDEF message.

The tag dispatch system uses the TNF and type fields to try to map a MIME type or URI to the NDEF message. If successful, it encapsulates that information inside of a [ACTION\\_NDEF\\_DISCOVERED](#) intent along with the actual payload. However, there are cases when the tag dispatch system cannot determine the type of data based

on the first NDEF record. This happens when the NDEF data cannot be mapped to a MIME type or URI, or when the NFC tag does not contain NDEF data to begin with. In such cases, a [Tag](#) object that has information about the tag's technologies and the payload are encapsulated inside of a [ACTION\\_TECH\\_DISCOVERED](#) intent instead.

[Table 1.](#) describes how the tag dispatch system maps TNF and type fields to MIME types or URIs. It also describes which TNFs cannot be mapped to a MIME type or URI. In these cases, the tag dispatch system falls back to [ACTION\\_TECH\\_DISCOVERED](#).

For example, if the tag dispatch system encounters a record of type [TNF\\_ABSOLUTE\\_URI](#), it maps the variable length type field of that record into a URI. The tag dispatch system encapsulates that URI in the data field of an [ACTION\\_NDEF\\_DISCOVERED](#) intent along with other information about the tag, such as the payload. On the other hand, if it encounters a record of type [TNF\\_UNKNOWN](#), it creates an intent that encapsulates the tag's technologies instead.

**Table 1.** Supported TNFs and their mappings

Type Name Format (TNF)	Mapping
<a href="#">TNF_ABSOLUTE_URI</a>	URI based on the type field.
<a href="#">TNF_EMPTY</a>	Falls back to <a href="#">ACTION_TECH_DISCOVERED</a> .
<a href="#">TNF_EXTERNAL_TYPE</a>	URI based on the URN in the type field. The URN is encoded into the NDEF type field in a shortened form: <domain_name>:<service_name>. Android maps this to a URI in the form: vnd.android.nfc://ext/<domain_name>:<service_name>.
<a href="#">TNF_MIME_MEDIA</a>	MIME type based on the type field.
<a href="#">TNF_UNCHANGED</a>	Invalid in the first record, so falls back to <a href="#">ACTION_TECH_DISCOVERED</a> .
<a href="#">TNF_UNKNOWN</a>	Falls back to <a href="#">ACTION_TECH_DISCOVERED</a> .
<a href="#">TNF_WELL_KNOWN</a>	MIME type or URI depending on the Record Type Definition (RTD), which you set in the type field. See <a href="#">Table 2.</a> for more information on available RTDs and their mappings.

**Table 2.** Supported RTDs for TNF\_WELL\_KNOWN and their mappings

Record Type Definition (RTD)	Mapping
<a href="#">RTD_ALTERNATIVE_CARRIER</a>	Falls back to <a href="#">ACTION_TECH_DISCOVERED</a> .
<a href="#">RTD_HANDOVER_CARRIER</a>	Falls back to <a href="#">ACTION_TECH_DISCOVERED</a> .
<a href="#">RTD_HANDOVER_REQUEST</a>	Falls back to <a href="#">ACTION_TECH_DISCOVERED</a> .
<a href="#">RTD_HANDOVER_SELECT</a>	Falls back to <a href="#">ACTION_TECH_DISCOVERED</a> .
<a href="#">RTD_SMART_POSTER</a>	URI based on parsing the payload.
<a href="#">RTD_TEXT</a>	MIME type of text/plain.
<a href="#">RTD_URI</a>	URI based on payload.

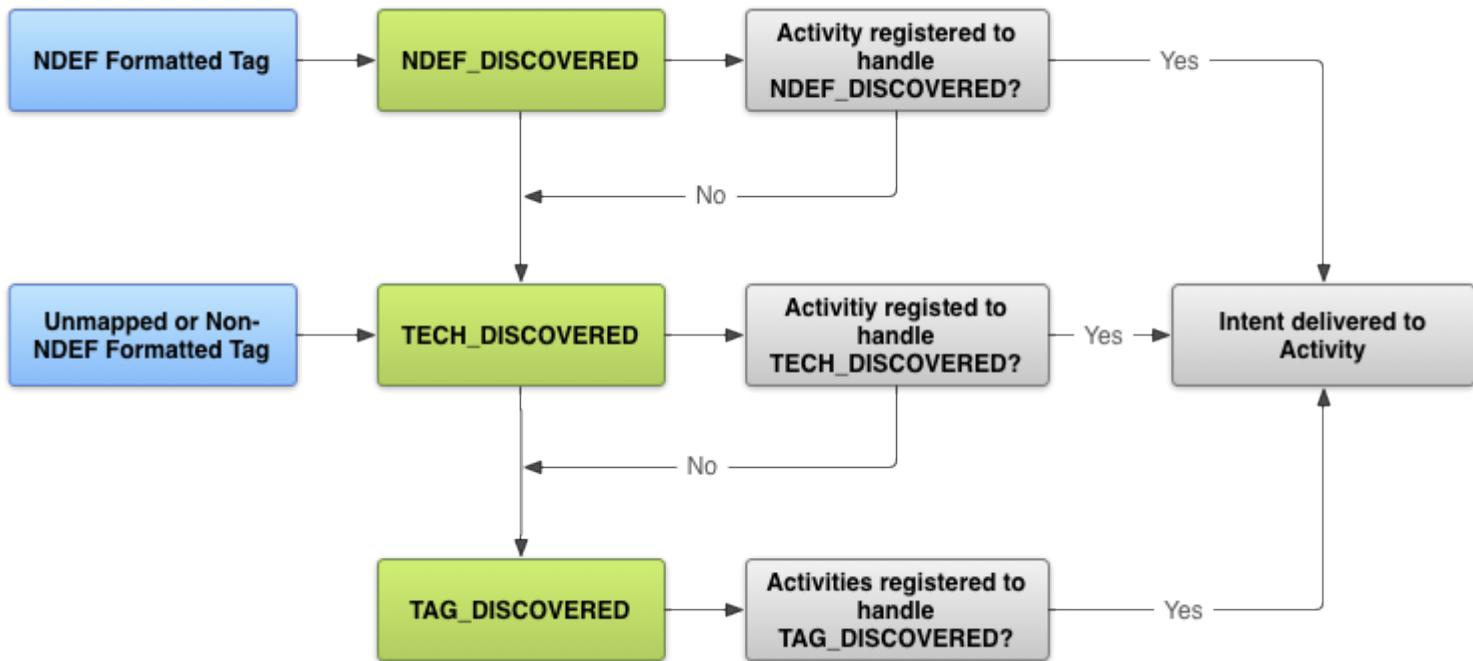
## How NFC Tags are Dispatched to Applications

When the tag dispatch system is done creating an intent that encapsulates the NFC tag and its identifying information, it sends the intent to an interested application that filters for the intent. If more than one application can handle the intent, the Activity Chooser is presented so the user can select the Activity. The tag dispatch system defines three intents, which are listed in order of highest to lowest priority:

1. [ACTION\\_NDEF\\_DISCOVERED](#): This intent is used to start an Activity when a tag that contains an NDEF payload is scanned and is of a recognized type. This is the highest priority intent, and the tag dispatch system tries to start an Activity with this intent before any other intent, whenever possible.
2. [ACTION\\_TECH\\_DISCOVERED](#): If no activities register to handle the [ACTION\\_NDEF\\_DISCOVERED](#) intent, the tag dispatch system tries to start an application with this intent. This intent is also directly started (without starting [ACTION\\_NDEF\\_DISCOVERED](#) first) if the tag that is scanned contains NDEF data that cannot be mapped to a MIME type or URI, or if the tag does not contain NDEF data but is of a known tag technology.
3. [ACTION\\_TAG\\_DISCOVERED](#): This intent is started if no activities handle the [ACTION\\_NDEF\\_DISCOVERED](#) or [ACTION\\_TECH\\_DISCOVERED](#) intents.

The basic way the tag dispatch system works is as follows:

1. Try to start an Activity with the intent that was created by the tag dispatch system when parsing the NFC tag (either [ACTION\\_NDEF\\_DISCOVERED](#) or [ACTION\\_TECH\\_DISCOVERED](#)).
2. If no activities filter for that intent, try to start an Activity with the next lowest priority intent (either [ACTION\\_TECH\\_DISCOVERED](#) or [ACTION\\_TAG\\_DISCOVERED](#)) until an application filters for the intent or until the tag dispatch system tries all possible intents.
3. If no applications filter for any of the intents, do nothing.



**Figure 1.** Tag Dispatch System

Whenever possible, work with NDEF messages and the [ACTION\\_NDEF\\_DISCOVERED](#) intent, because it is the most specific out of the three. This intent allows you to start your application at a more appropriate time than the other two intents, giving the user a better experience.

## Requesting NFC Access in the Android Manifest

Before you can access a device's NFC hardware and properly handle NFC intents, declare these items in your `AndroidManifest.xml` file:

- The NFC `<uses-permission>` element to access the NFC hardware:

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

- The minimum SDK version that your application can support. API level 9 only supports limited tag dispatch via [ACTION\\_TAG\\_DISCOVERED](#), and only gives access to NDEF messages via the [EXTRA\\_NDEF\\_MESSAGES](#) extra. No other tag properties or I/O operations are accessible. API level 10 includes comprehensive reader/writer support as well as foreground NDEF pushing, and API level 14 provides an easier way to push NDEF messages to other devices with Android Beam and extra convenience methods to create NDEF records.

```
<uses-sdk android:minSdkVersion="10"/>
```

- The `uses-feature` element so that your application shows up in Google Play only for devices that have NFC hardware:

```
<uses-feature android:name="android.hardware.nfc" android:required="true">
```

If your application uses NFC functionality, but that functionality is not crucial to your application, you can omit the `uses-feature` element and check for NFC availability at runtime by checking to see if [getOrDefaultAdapter\(\)](#) is null.

## Filtering for NFC Intents

To start your application when an NFC tag that you want to handle is scanned, your application can filter for one, two, or all three of the NFC intents in the Android manifest. However, you usually want to filter for the [ACTION\\_NDEF\\_DISCOVERED](#) intent for the most control of when your application starts. The [ACTION\\_TECH\\_DISCOVERED](#) intent is a fallback for [ACTION\\_NDEF\\_DISCOVERED](#) when no applications filter for [ACTION\\_NDEF\\_DISCOVERED](#) or for when the payload is not NDEF. Filtering for [ACTION\\_TAG\\_DISCOVERED](#) is usually too general of a category to filter on. Many applications will filter for [ACTION\\_NDEF\\_DISCOVERED](#) or [ACTION\\_TECH\\_DISCOVERED](#) before [ACTION\\_TAG\\_DISCOVERED](#), so your application has a low probability of starting. [ACTION\\_TAG\\_DISCOVERED](#) is only available as a last resort for applications to filter for in the cases where no other applications are installed to handle the [ACTION\\_NDEF\\_DISCOVERED](#) or [ACTION\\_TECH\\_DISCOVERED](#) intent.

Because NFC tag deployments vary and are many times not under your control, this is not always possible, which is why you can fallback to the other two intents when necessary. When you have control over the types of tags and data written, it is recommended that you use NDEF to format your tags. The following sections describe how to filter for each type of intent.

### **ACTION\_NDEF\_DISCOVERED**

To filter for [ACTION\\_NDEF\\_DISCOVERED](#) intents, declare the intent filter along with the type of data that you want to filter for. The following example filters for [ACTION\\_NDEF\\_DISCOVERED](#) intents with a MIME type of `text/plain`:

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain" />
</intent-filter>
```

The following example filters for a URI in the form of `http://developer.android.com/index.html`.

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
```

```
<category android:name="android.intent.category.DEFAULT"/>
<data android:scheme="http"
      android:host="developer.android.com"
      android:pathPrefix="/index.html" />
</intent-filter>
```

## ACTION\_TECH\_DISCOVERED

If your activity filters for the [ACTION\\_TECH\\_DISCOVERED](#) intent, you must create an XML resource file that specifies the technologies that your activity supports within a `tech-list` set. Your activity is considered a match if a `tech-list` set is a subset of the technologies that are supported by the tag, which you can obtain by calling [getTechList\(\)](#).

For example, if the tag that is scanned supports MifareClassic, NdefFormattable, and NfcA, your `tech-list` set must specify all three, two, or one of the technologies (and nothing else) in order for your activity to be matched.

The following sample defines all of the technologies. You can remove the ones that you do not need. Save this file (you can name it anything you wish) in the `<project-root>/res/xml` folder.

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
  <tech-list>
    <tech>android.nfc.tech.IsoDep</tech>
    <tech>android.nfc.tech.NfcA</tech>
    <tech>android.nfc.tech.NfcB</tech>
    <tech>android.nfc.tech.NfcF</tech>
    <tech>android.nfc.tech.NfcV</tech>
    <tech>android.nfc.tech.Ndef</tech>
    <tech>android.nfc.tech.NdefFormattable</tech>
    <tech>android.nfc.tech.MifareClassic</tech>
    <tech>android.nfc.tech.MifareUltralight</tech>
  </tech-list>
</resources>
```

You can also specify multiple `tech-list` sets. Each of the `tech-list` sets is considered independently, and your activity is considered a match if any single `tech-list` set is a subset of the technologies that are returned by [getTechList\(\)](#). This provides AND and OR semantics for matching technologies. The following example matches tags that can support the NfcA and Ndef technologies or can support the NfcB and Ndef technologies:

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
  <tech-list>
    <tech>android.nfc.tech.NfcA</tech>
    <tech>android.nfc.tech.Ndef</tech>
  </tech-list>
</resources>

<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
  <tech-list>
    <tech>android.nfc.tech.NfcB</tech>
    <tech>android.nfc.tech.Ndef</tech>
  </tech-list>
</resources>
```

In your `AndroidManifest.xml` file, specify the resource file that you just created in the `<meta-data>` element inside the `<activity>` element like in the following example:

```
<activity>
...
<intent-filter>
    <action android:name="android.nfc.action.TECH_DISCOVERED"/>
</intent-filter>

<meta-data android:name="android.nfc.action.TECH_DISCOVERED"
    android:resource="@xml/nfc_tech_filter" />
...
</activity>
```

For more information about working with tag technologies and the [ACTION\\_TECH\\_DISCOVERED](#) intent, see [Working with Supported Tag Technologies](#) in the Advanced NFC document.

## ACTION\_TAG\_DISCOVERED

To filter for [ACTION\\_TAG\\_DISCOVERED](#) use the following intent filter:

```
<intent-filter>
    <action android:name="android.nfc.action.TAG_DISCOVERED"/>
</intent-filter>
```

## Obtaining information from intents

If an activity starts because of an NFC intent, you can obtain information about the scanned NFC tag from the intent. Intents can contain the following extras depending on the tag that was scanned:

- [EXTRA\\_TAG](#) (required): A [Tag](#) object representing the scanned tag.
- [EXTRA\\_NDEF\\_MESSAGES](#) (optional): An array of NDEF messages parsed from the tag. This extra is mandatory on [intents](#).
- [{@link android.nfc.NfcAdapter#EXTRA\\_ID}](#) (optional): The low-level ID of the tag.

To obtain these extras, check to see if your activity was launched with one of the NFC intents to ensure that a tag was scanned, and then obtain the extras out of the intent. The following example checks for the [ACTION\\_NDEF\\_DISCOVERED](#) intent and gets the NDEF messages from an intent extra.

```
public void onResume() {
    super.onResume();
    ...
    if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction())) {
        Parcelable[] rawMsgs = intent.getParcelableArrayExtra(NfcAdapter.EXTRA_
        if (rawMsgs != null) {
            msgs = new NdefMessage[rawMsgs.length];
            for (int i = 0; i < rawMsgs.length; i++) {
                msgs[i] = (NdefMessage) rawMsgs[i];
            }
        }
    }
    //process the msgs array
}
```

Alternatively, you can obtain a [Tag](#) object from the intent, which will contain the payload and allow you to enumerate the tag's technologies:

```
Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
```

## Creating Common Types of NDEF Records

This section describes how to create common types of NDEF records to help you when writing to NFC tags or sending data with Android Beam. Starting with Android 4.0 (API level 14), the [createUri\(\)](#) method is available to help you create URI records automatically. Starting in Android 4.1 (API level 16), [createExternal\(\)](#) and [createMime\(\)](#) are available to help you create MIME and external type NDEF records. Use these helper methods whenever possible to avoid mistakes when manually creating NDEF records.

This section also describes how to create the corresponding intent filter for the record. All of these NDEF record examples should be in the first NDEF record of the NDEF message that you are writing to a tag or beaming.

### TNF\_ABSOLUTE\_URI

**Note:** We recommend that you use the [RTD\\_URI](#) type instead of [TNF\\_ABSOLUTE\\_URI](#), because it is more efficient.

You can create a [TNF\\_ABSOLUTE\\_URI](#) NDEF record in the following way:

```
NdefRecord uriRecord = new NdefRecord(  
    NdefRecord.TNF_ABSOLUTE_URI,  
    "http://developer.android.com/index.html".getBytes(Charset.forName("US-ASCII"))  
    new byte[0], new byte[0]);
```

The intent filter for the previous NDEF record would look like this:

```
<intent-filter>  
    <action android:name="android.nfc.action.NDEF_DISCOVERED" />  
    <category android:name="android.intent.category.DEFAULT" />  
    <data android:scheme="http"  
        android:host="developer.android.com"  
        android:pathPrefix="/index.html" />  
</intent-filter>
```

### TNF\_MIME\_MEDIA

You can create a [TNF\\_MIME\\_MEDIA](#) NDEF record in the following ways.

Using the [createMime\(\)](#) method:

```
NdefRecord mimeRecord = NdefRecord.createMime("application/vnd.com.example.android.beam"  
    "Beam me up, Android".getBytes(Charset.forName("US-ASCII")));
```

Creating the [NdefRecord](#) manually:

```
NdefRecord mimeRecord = new NdefRecord(  
    NdefRecord.TNF_MIME_MEDIA,
```

```
"application/vnd.com.example.android.beam".getBytes(Charset.forName("US-ASCII"))
new byte[0], "Beam me up, Android!".getBytes(Charset.forName("US-ASCII")));
```

The intent filter for the previous NDEF records would look like this:

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="application/vnd.com.example.android.beam" />
</intent-filter>
```

## TNF\_WELL\_KNOWN with RTD\_TEXT

You can create a [TNF WELL KNOWN](#) NDEF record in the following way:

```
public NdefRecord createTextRecord(String payload, Locale locale, boolean encodeInUtf8) {
    byte[] langBytes = locale.getLanguage().getBytes(Charset.forName("US-ASCII"));
    Charset utfEncoding = encodeInUtf8 ? Charset.forName("UTF-8") : Charset.forName("UTF-16");
    byte[] textBytes = payload.getBytes(utfEncoding);
    int utfBit = encodeInUtf8 ? 0 : (1 << 7);
    char status = (char) (utfBit + langBytes.length);
    byte[] data = new byte[1 + langBytes.length + textBytes.length];
    data[0] = (byte) status;
    System.arraycopy(langBytes, 0, data, 1, langBytes.length);
    System.arraycopy(textBytes, 0, data, 1 + langBytes.length, textBytes.length);
    NdefRecord record = new NdefRecord(NdefRecord.TNF_WELL_KNOWN,
                                        NdefRecord.RTD_TEXT, new byte[0], data);
    return record;
}
```

the intent filter would look like this:

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
</intent-filter>
```

## TNF\_WELL\_KNOWN with RTD\_URI

You can create a [TNF WELL KNOWN](#) NDEF record in the following ways.

Using the [createUri\(String\)](#) method:

```
NdefRecord rtdUriRecord1 = NdefRecord.createUri("http://example.com");
```

Using the [createUri\(Uri\)](#) method:

```
Uri uri = new Uri("http://example.com");
NdefRecord rtdUriRecord2 = NdefRecord.createUri(uri);
```

Creating the [NdefRecord](#) manually:

```

byte[] uriField = "example.com".getBytes(Charset.forName("US-ASCII"));
byte[] payload = new byte[uriField.length + 1]; //add 1 for the URI prefix
byte payload[0] = 0x01; //prefixes http://
System.arraycopy(uriField, 0, payload, 1, uriField.length); //appends URI to payload
NdefRecord rtdUriRecord = new NdefRecord(
    NdefRecord.TNF_WELL_KNOWN, NdefRecord.RTD_URI, new byte[0], payload);

```

The intent filter for the previous NDEF records would look like this:

```

<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:scheme="http"
        android:host="example.com"
        android:pathPrefix="" />
</intent-filter>

```

## TNF\_EXTERNAL\_TYPE

You can create a [TNF\\_EXTERNAL\\_TYPE](#) NDEF record in the following ways:

Using the [createExternal\(\)](#) method:

```

byte[] payload; //assign to your data
String domain = "com.example"; //usually your app's package name
String type = "externalType";
NdefRecord extRecord = NdefRecord.createExternal(domain, type, payload);

```

Creating the [NdefRecord](#) manually:

```

byte[] payload;
...
NdefRecord extRecord = new NdefRecord(
    NdefRecord.TNF_EXTERNAL_TYPE, "com.example:externalType", new byte[0], payload);

```

The intent filter for the previous NDEF records would look like this:

```

<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:scheme="vnd.android.nfc"
        android:host="ext"
        android:pathPrefix="/com.example:externalType"/>
</intent-filter>

```

Use TNF\_EXTERNAL\_TYPE for more generic NFC tag deployments to better support both Android-powered and non-Android-powered devices.

**Note:** URNs for [TNF\\_EXTERNAL\\_TYPE](#) have a canonical format of:

urn:nfc:ext:example.com:externalType, however the NFC Forum RTD specification declares that the urn:nfc:ext: portion of the URN must be omitted from the NDEF record. So all you need to provide is the domain (example.com in the example) and type (externalType in the example) separated by a colon. When dispatching TNF\_EXTERNAL\_TYPE, Android converts the

`urn:nfc:ext:example.com:externalType` URN to a `vnfd.android.nfc://ext/example.com:externalType` URI, which is what the intent filter in the example declares.

## Android Application Records

Introduced in Android 4.0 (API level 14), an Android Application Record (AAR) provides a stronger certainty that your application is started when an NFC tag is scanned. An AAR has the package name of an application embedded inside an NDEF record. You can add an AAR to any NDEF record of your NDEF message, because Android searches the entire NDEF message for AARs. If it finds an AAR, it starts the application based on the package name inside the AAR. If the application is not present on the device, Google Play is launched to download the application.

AARs are useful if you want to prevent other applications from filtering for the same intent and potentially handling specific tags that you have deployed. AARs are only supported at the application level, because of the package name constraint, and not at the Activity level as with intent filtering. If you want to handle an intent at the Activity level, [use intent filters](#).

If a tag contains an AAR, the tag dispatch system dispatches in the following manner:

1. Try to start an Activity using an intent filter as normal. If the Activity that matches the intent also matches the AAR, start the Activity.
2. If the Activity that filters for the intent does not match the AAR, if multiple Activities can handle the intent, or if no Activity handles the intent, start the application specified by the AAR.
3. If no application can start with the AAR, go to Google Play to download the application based on the AAR.

**Note:** You can override AARs and the intent dispatch system with the [foreground dispatch system](#), which allows a foreground activity to have priority when an NFC tag is discovered. With this method, the activity must be in the foreground to override AARs and the intent dispatch system.

If you still want to filter for scanned tags that do not contain an AAR, you can declare intent filters as normal. This is useful if your application is interested in other tags that do not contain an AAR. For example, maybe you want to guarantee that your application handles proprietary tags that you deploy as well as general tags deployed by third parties. Keep in mind that AARs are specific to Android 4.0 devices or later, so when deploying tags, you most likely want to use a combination of AARs and MIME types/URIs to support the widest range of devices. In addition, when you deploy NFC tags, think about how you want to write your NFC tags to enable support for the most devices (Android-powered and other devices). You can do this by defining a relatively unique MIME type or URI to make it easier for applications to distinguish.

Android provides a simple API to create an AAR, [createApplicationRecord\(\)](#). All you need to do is embed the AAR anywhere in your [NdefMessage](#). You do not want to use the first record of your [NdefMessage](#), unless the AAR is the only record in the [NdefMessage](#). This is because the Android system checks the first record of an [NdefMessage](#) to determine the MIME type or URI of the tag, which is used to create an intent for applications to filter. The following code shows you how to create an AAR:

```
NdefMessage msg = new NdefMessage(  
    new NdefRecord[] {  
        ...,  
        NdefRecord.createApplicationRecord("com.example.android.beam")  
    }  
)
```

# Beaming NDEF Messages to Other Devices

Android Beam allows simple peer-to-peer data exchange between two Android-powered devices. The application that wants to beam data to another device must be in the foreground and the device receiving the data must not be locked. When the beaming device comes in close enough contact with a receiving device, the beaming device displays the "Touch to Beam" UI. The user can then choose whether or not to beam the message to the receiving device.

**Note:** Foreground NDEF pushing was available at API level 10, which provides similar functionality to Android Beam. These APIs have since been deprecated, but are available to support older devices. See [enable-ForegroundNdefPush\(\)](#) for more information.

You can enable Android Beam for your application by calling one of the two methods:

- [setNdefPushMessage\(\)](#): Accepts an [NdefMessage](#) to set as the message to beam. Automatically beams the message when two devices are in close enough proximity.
- [setNdefPushMessageCallback\(\)](#): Accepts a callback that contains a [createNdefMessage\(\)](#) which is called when a device is in range to beam data to. The callback lets you create the NDEF message only when necessary.

An activity can only push one NDEF message at a time, so [setNdefPushMessageCallback\(\)](#) takes precedence over [setNdefPushMessage\(\)](#) if both are set. To use Android Beam, the following general guidelines must be met:

- The activity that is beaming the data must be in the foreground. Both devices must have their screens unlocked.
- You must encapsulate the data that you are beaming in an [NdefMessage](#) object.
- The NFC device that is receiving the beamed data must support the `com.android.npp` NDEF push protocol or NFC Forum's SNEP (Simple NDEF Exchange Protocol). The `com.android.npp` protocol is required for devices on API level 9 (Android 2.3) to API level 13 (Android 3.2). `com.android.npp` and SNEP are both required on API level 14 (Android 4.0) and later.

**Note:** If your activity enables Android Beam and is in the foreground, the standard intent dispatch system is disabled. However, if your activity also enables [foreground dispatching](#), then it can still scan tags that match the intent filters set in the foreground dispatching.

To enable Android Beam:

1. Create an [NdefMessage](#) that contains the [NdefRecords](#) that you want to push onto the other device.
2. Call [setNdefPushMessage\(\)](#) with a [NdefMessage](#) or call [setNdefPushMessageCallback\(\)](#) passing in a [NfcAdapter.CreateNdefMessageCallback](#) object in the `onCreate()` method of your activity. These methods require at least one activity that you want to enable with Android Beam, along with an optional list of other activities to activate.

In general, you normally use [setNdefPushMessage\(\)](#) if your Activity only needs to push the same NDEF message at all times, when two devices are in range to communicate. You use [setNdefPushMessageCallback](#) when your application cares about the current context of the application and wants to push an NDEF message depending on what the user is doing in your application.

The following sample shows how a simple activity calls [NfcAdapter.CreateNdefMessageCallback](#) in the `onCreate()` method of an activity (see [AndroidBeamDemo](#) for the complete sample). This example also has methods to help you create a MIME record:

```
package com.example.android.beam;

import android.app.Activity;
import android.content.Intent;
import android.nfc.NdefMessage;
import android.nfc.NdefRecord;
import android.nfc.NfcAdapter;
import android.nfc.NfcAdapter.CreateNdefMessageCallback;
import android.nfc.NfcEvent;
import android.os.Bundle;
import android.os.Parcelable;
import android.widget.TextView;
import android.widget.Toast;
import java.nio.charset.Charset;

public class Beam extends Activity implements CreateNdefMessageCallback {
    NfcAdapter mNfcAdapter;
    TextView textView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView textView = (TextView) findViewById(R.id.textView);
        // Check for available NFC Adapter
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        if (mNfcAdapter == null) {
            Toast.makeText(this, "NFC is not available", Toast.LENGTH_LONG).show();
            finish();
            return;
        }
        // Register callback
        mNfcAdapter.setNdefPushMessageCallback(this, this);
    }

    @Override
    public NdefMessage createNdefMessage(NfcEvent event) {
        String text = ("Beam me up, Android!\n\n" +
                      "Beam Time: " + System.currentTimeMillis());
        NdefMessage msg = new NdefMessage(
            new NdefRecord[] { createMime(
                "application/vnd.com.example.android.beam", text.getBytes()
            ) });
        /**
         * The Android Application Record (AAR) is commented out. When a device
         * receives a push with an AAR in it, the application specified in the
         * is guaranteed to run. The AAR overrides the tag dispatch system.
         * You can add it back in to guarantee that this
         * activity starts when receiving a beamed message. For now, this code
         * uses the tag dispatch system.
        */
        //,NdefRecord.createApplicationRecord("com.example.android.beam")
    );
    return msg;
}
```

```

}

@Override
public void onResume() {
    super.onResume();
    // Check to see that the Activity started due to an Android Beam
    if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction()))
        processIntent(getIntent());
}
}

@Override
public void onNewIntent(Intent intent) {
    // onResume gets called after this to handle the intent
    setIntent(intent);
}

/**
 * Parses the NDEF Message from the intent and prints to the TextView
 */
void processIntent(Intent intent) {
    textView = (TextView) findViewById(R.id.textView);
    Parcelable[] rawMsgs = intent.getParcelableArrayExtra(
        NfcAdapter.EXTRA_NDEF_MESSAGES);
    // only one message sent during the beam
    NdefMessage msg = (NdefMessage) rawMsgs[0];
    // record 0 contains the MIME type, record 1 is the AAR, if present
    textView.setText(new String(msg.getRecords()[0].getPayload()));
}
}

```

Note that this code comments out an AAR, which you can remove. If you enable the AAR, the application specified in the AAR always receives the Android Beam message. If the application is not present, Google Play is started to download the application. Therefore, the following intent filter is not technically necessary for Android 4.0 devices or later if the AAR is used:

```

<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="application/vnd.com.example.android.beam"/>
</intent-filter>

```

With this intent filter, the `com.example.android.beam` application now can be started when it scans an NFC tag or receives an Android Beam with an AAR of type `com.example.android.beam`, or when an NDEF formatted message contains a MIME record of type `application/vnd.com.example.android.beam`.

Even though AARs guarantee an application is started or downloaded, intent filters are recommended, because they let you start an Activity of your choice in your application instead of always starting the main Activity within the package specified by an AAR. AARs do not have Activity level granularity. Also, because some Android-powered devices do not support AARs, you should also embed identifying information in the first NDEF record of your NDEF messages and filter for that as well, just in case. See [Creating Common Types of NDEF records](#) for more information on how to create records.

# Advanced NFC

## In this document

1. [Working with Supported Tag Technologies](#)
  1. [Working with tag technologies and the ACTION\\_TECH\\_DISCOVERED intent](#)
  2. [Reading and writing to tags](#)
2. [Using the Foreground Dispatch System](#)

This document describes advanced NFC topics, such as working with various tag technologies, writing to NFC tags, and foreground dispatching, which allows an application in the foreground to handle intents even when other applications filter for the same ones.

## Working with Supported Tag Technologies

When working with NFC tags and Android-powered devices, the main format you use to read and write data on tags is NDEF. When a device scans a tag with NDEF data, Android provides support in parsing the message and delivering it in an [NdefMessage](#) when possible. There are cases, however, when you scan a tag that does not contain NDEF data or when the NDEF data could not be mapped to a MIME type or URI. In these cases, you need to open communication directly with the tag and read and write to it with your own protocol (in raw bytes). Android provides generic support for these use cases with the [android.nfc.tech](#) package, which is described in [Table 1](#). You can use the [getTechList\(\)](#) method to determine the technologies supported by the tag and create the corresponding [TagTechnology](#) object with one of classes provided by [android.nfc.tech](#)

**Table 1.** Supported tag technologies

Class	Description
<a href="#">TagTechnology</a>	The interface that all tag technology classes must implement.
<a href="#">NfcA</a>	Provides access to NFC-A (ISO 14443-3A) properties and I/O operations.
<a href="#">NfcB</a>	Provides access to NFC-B (ISO 14443-3B) properties and I/O operations.
<a href="#">NfcF</a>	Provides access to NFC-F (JIS 6319-4) properties and I/O operations.
<a href="#">NfcV</a>	Provides access to NFC-V (ISO 15693) properties and I/O operations.
<a href="#">IsoDep</a>	Provides access to ISO-DEP (ISO 14443-4) properties and I/O operations.
<a href="#">Ndef</a>	Provides access to NDEF data and operations on NFC tags that have been formatted as NDEF.
<a href="#">NdefFormattable</a>	Provides a format operations for tags that may be NDEF formattable.

The following tag technologies are not required to be supported by Android-powered devices.

**Table 2.** Optional supported tag technologies

Class	Description
<a href="#">MifareClassic</a>	Provides access to MIFARE Classic properties and I/O operations, if this Android device supports MIFARE.
<a href="#">MifareUltralight</a>	Provides access to MIFARE Ultralight properties and I/O operations, if this Android device supports MIFARE.

## Working with tag technologies and the ACTION\_TECH\_DISCOVERED intent

When a device scans a tag that has NDEF data on it, but could not be mapped to a MIME or URI, the tag dispatch system tries to start an activity with the [ACTION\\_TECH\\_DISCOVERED](#) intent. The [ACTION\\_TECH\\_DISCOVERED](#) is also used when a tag with non-NDEF data is scanned. Having this fallback allows you to work with the data on the tag directly if the tag dispatch system could not parse it for you. The basic steps when working with tag technologies are as follows:

1. Filter for an [ACTION\\_TECH\\_DISCOVERED](#) intent specifying the tag technologies that you want to handle. See [Filtering for NFC intents](#) for more information. In general, the tag dispatch system tries to start a [ACTION\\_TECH\\_DISCOVERED](#) intent when an NDEF message cannot be mapped to a MIME type or URI, or if the tag scanned did not contain NDEF data. For more information on how this is determined, see [The Tag Dispatch System](#).
2. When your application receives the intent, obtain the [Tag](#) object from the intent:

```
Tag tagFromIntent = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
```

3. Obtain an instance of a [TagTechnology](#), by calling one of the `get` factory methods of the classes in the [android.nfc.tech](#) package. You can enumerate the supported technologies of the tag by calling [getTechList\(\)](#) before calling a `get` factory method. For example, to obtain an instance of [MifareUltralight](#) from a [Tag](#), do the following:

```
MifareUltralight.get(intent.getParcelableExtra(NfcAdapter.EXTRA_TAG));
```

## Reading and writing to tags

Reading and writing to an NFC tag involves obtaining the tag from the intent and opening communication with the tag. You must define your own protocol stack to read and write data to the tag. Keep in mind, however, that you can still read and write NDEF data when working directly with a tag. It is up to you how you want to structure things. The following example shows how to work with a MIFARE Ultralight tag.

```
package com.example.android.nfc;

import android.nfc.Tag;
import android.nfc.tech.MifareUltralight;
import android.util.Log;
import java.io.IOException;
import java.nio.charset.Charset;

public class MifareUltralightTagTester {

    private static final String TAG = MifareUltralightTagTester.class.getSimpleName();

    public void writeTag(Tag tag, String tagText) {
        MifareUltralight ultralight = MifareUltralight.get(tag);
        try {
            ultralight.connect();
            ultralight.writePage(4, "abcd".getBytes(Charset.forName("US-ASCII")));
            ultralight.writePage(5, "efgh".getBytes(Charset.forName("US-ASCII")));
            ultralight.writePage(6, "ijkl".getBytes(Charset.forName("US-ASCII")));
            ultralight.writePage(7, "mnop".getBytes(Charset.forName("US-ASCII")));
        } catch (IOException e) {
            Log.e(TAG, "IOException while closing MifareUltralight...", e);
        } finally {
```

```

        try {
            ultralight.close();
        } catch (IOException e) {
            Log.e(TAG, "IOException while closing MifareUltralight...", e);
        }
    }

public String readTag(Tag tag) {
    MifareUltralight mifare = MifareUltralight.get(tag);
    try {
        mifare.connect();
        byte[] payload = mifare.readPages(4);
        return new String(payload, Charset.forName("US-ASCII"));
    } catch (IOException e) {
        Log.e(TAG, "IOException while writing MifareUltralight
message...", e);
    } finally {
        if (mifare != null) {
            try {
                mifare.close();
            } catch (IOException e) {
                Log.e(TAG, "Error closing tag...", e);
            }
        }
    }
    return null;
}
}

```

## Using the Foreground Dispatch System

The foreground dispatch system allows an activity to intercept an intent and claim priority over other activities that handle the same intent. Using this system involves constructing a few data structures for the Android system to be able to send the appropriate intents to your application. To enable the foreground dispatch system:

1. Add the following code in the `onCreate()` method of your activity:
  1. Create a [PendingIntent](#) object so the Android system can populate it with the details of the tag when it is scanned.

```

PendingIntent pendingIntent = PendingIntent.getActivity(
    this, 0, new Intent(this, getClass()).addFlags(Intent.FLAG_ACTIVITY_

```

2. Declare intent filters to handle the intents that you want to intercept. The foreground dispatch system checks the specified intent filters with the intent that is received when the device scans a tag. If it matches, then your application handles the intent. If it does not match, the foreground dispatch system falls back to the intent dispatch system. Specifying a `null` array of intent filters and technology filters, specifies that you want to filter for all tags that fallback to the `TAG_DISCOVERED` intent. The code snippet below handles all MIME types for `NDEF_DISCOVERED`. You should only handle the ones that you need.

```

IntentFilter ndef = new IntentFilter(NfcAdapter.ACTION_NDEF_DISCOVERED);
try {
    ndef.addDataType("*/*");      /* Handles all MIME based dispatches.
                                     You should specify only the types
                                     your app cares about. */
}
catch (MalformedMimeTypeException e) {
    throw new RuntimeException("fail", e);
}
intentFiltersArray = new IntentFilter[] {ndef, };

```

3. Set up an array of tag technologies that your application wants to handle. Call the `Object.class.getName()` method to obtain the class of the technology that you want to support.

```
techListsArray = new String[][] { new String[] { NfcF.class.getName() } }
```

2. Override the following activity lifecycle callbacks and add logic to enable and disable the foreground dispatch when the activity loses (`onPause()`) and regains (`onResume()`) focus. `enableForegroundDispatch()` must be called from the main thread and only when the activity is in the foreground (calling in `onResume()` guarantees this). You also need to implement the `onNewIntent` callback to process the data from the scanned NFC tag.

```

public void onPause() {
    super.onPause();
    mAdapter.disableForegroundDispatch(this);
}

public void onResume() {
    super.onResume();
    mAdapter.enableForegroundDispatch(this, pendingIntent, intentFiltersArray);
}

public void onNewIntent(Intent intent) {
    Tag tagFromIntent = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    //do something with tagFromIntent
}

```

See the [ForegroundDispatch](#) sample from API Demos for the complete sample.

# Wi-Fi Direct

## In this document

1. [API Overview](#)
2. [Creating a Broadcast Receiver for Wi-Fi Direct Intents](#)
3. [Creating a Wi-Fi Direct Application](#)
  1. [Initial setup](#)
  2. [Discovering peers](#)
  3. [Connecting to peers](#)
  4. [Transferring data](#)

## Related Samples

1. [Wi-Fi Direct Demo](#)

Wi-Fi Direct allows Android 4.0 (API level 14) or later devices with the appropriate hardware to connect directly to each other via Wi-Fi without an intermediate access point. Using these APIs, you can discover and connect to other devices when each device supports Wi-Fi Direct, then communicate over a speedy connection across distances much longer than a Bluetooth connection. This is useful for applications that share data among users, such as a multiplayer game or a photo sharing application.

The Wi-Fi Direct APIs consist of the following main parts:

- Methods that allow you to discover, request, and connect to peers are defined in the [WifiP2pManager](#) class.
- Listeners that allow you to be notified of the success or failure of [WifiP2pManager](#) method calls. When calling [WifiP2pManager](#) methods, each method can receive a specific listener passed in as a parameter.
- Intents that notify you of specific events detected by the Wi-Fi Direct framework, such as a dropped connection or a newly discovered peer.

You often use these three main components of the APIs together. For example, you can provide a [WifiP2pManager.ActionListener](#) to a call to [discoverPeers\(\)](#), so that you can be notified with the [ActionListener.onSuccess\(\)](#) and [ActionListener.onFailure\(\)](#) methods. A [WIFI\\_P2P\\_PEERS\\_CHANGED\\_ACTION](#) intent is also broadcast if the [discoverPeers\(\)](#) method discovers that the peers list has changed.

## API Overview

The [WifiP2pManager](#) class provides methods to allow you to interact with the Wi-Fi hardware on your device to do things like discover and connect to peers. The following actions are available:

**Table 1.** Wi-Fi Direct Methods

Method	Description
<a href="#">initialize()</a>	Registers the application with the Wi-Fi framework. This must be called before calling any other Wi-Fi Direct method.
<a href="#">connect()</a>	Starts a peer-to-peer connection with a device with the specified configuration.
<a href="#">cancelConnect()</a>	Cancels any ongoing peer-to-peer group negotiation.

<a href="#">requestConnectInfo()</a>	Requests a device's connection information.
<a href="#">createGroup()</a>	Creates a peer-to-peer group with the current device as the group owner.
<a href="#">removeGroup()</a>	Removes the current peer-to-peer group.
<a href="#">requestGroupInfo()</a>	Requests peer-to-peer group information.
<a href="#">discoverPeers()</a>	Initiates peer discovery
<a href="#">requestPeers()</a>	Requests the current list of discovered peers.

[WifiP2pManager](#) methods let you pass in a listener, so that the Wi-Fi Direct framework can notify your activity of the status of a call. The available listener interfaces and the corresponding [WifiP2pManager](#) method calls that use the listeners are described in the following table:

**Table 2.** Wi-Fi Direct Listeners

Listener interface	Associated actions
<a href="#">WifiP2pManager.ActionListener</a>	<a href="#">connect()</a> , <a href="#">cancelConnect()</a> , <a href="#">createGroup()</a> , <a href="#">removeGroup()</a> , and <a href="#">discoverPeers()</a>
<a href="#">WifiP2pManager.ChannelListener</a>	<a href="#">initialize()</a>
<a href="#">WifiP2pManager.ConnectionInfoListener</a>	<a href="#">requestConnectInfo()</a>
<a href="#">WifiP2pManager.GroupInfoListener</a>	<a href="#">requestGroupInfo()</a>
<a href="#">WifiP2pManager.PeerListListener</a>	<a href="#">requestPeers()</a>

The Wi-Fi Direct APIs define intents that are broadcast when certain Wi-Fi Direct events happen, such as when a new peer is discovered or when a device's Wi-Fi state changes. You can register to receive these intents in your application by [creating a broadcast receiver](#) that handles these intents:

**Table 3.** Wi-Fi Direct Intents

Intent	Description
<a href="#">WIFI_P2P_CONNECTION_CHANGED_ACTION</a>	Broadcast when the state of the device's Wi-Fi connection changes.
<a href="#">WIFI_P2P_PEERS_CHANGED_ACTION</a>	Broadcast when you call <a href="#">discoverPeers()</a> . You usually want to call <a href="#">requestPeers()</a> to get an updated list of peers if you handle this intent in your application.
<a href="#">WIFI_P2P_STATE_CHANGED_ACTION</a>	Broadcast when Wi-Fi Direct is enabled or disabled on the device.
<a href="#">WIFI_P2P_THIS_DEVICE_CHANGED_ACTION</a>	Broadcast when a device's details have changed, such as the device's name.

## Creating a Broadcast Receiver for Wi-Fi Direct Intents

A broadcast receiver allows you to receive intents broadcast by the Android system, so that your application can respond to events that you are interested in. The basic steps for creating a broadcast receiver to handle Wi-Fi Direct intents are as follows:

1. Create a class that extends the [BroadcastReceiver](#) class. For the class' constructor, you most likely want to have parameters for the [WifiP2pManager](#), [WifiP2pManager.Channel](#), and the activity that this broadcast receiver will be registered in. This allows the broadcast receiver to send updates to the activity as well as have access to the Wi-Fi hardware and a communication channel if needed.

2. In the broadcast receiver, check for the intents that you are interested in `onReceive()`. Carry out any necessary actions depending on the intent that is received. For example, if the broadcast receiver receives a `WIFI_P2P_PEERS_CHANGED_ACTION` intent, you can call the `requestPeers()` method to get a list of the currently discovered peers.

The following code shows you how to create a typical broadcast receiver. The broadcast receiver takes a `WifiP2pManager` object and an activity as arguments and uses these two classes to appropriately carry out the needed actions when the broadcast receiver receives an intent:

```
/**  
 * A BroadcastReceiver that notifies of important Wi-Fi p2p events.  
 */  
public class WiFiDirectBroadcastReceiver extends BroadcastReceiver {  
  
    private WifiP2pManager mManager;  
    private Channel mChannel;  
    private MyWiFiActivity mActivity;  
  
    public WiFiDirectBroadcastReceiver(WifiP2pManager manager, Channel channel,  
                                       MyWiFiActivity activity) {  
        super();  
        this.mManager = manager;  
        this.mChannel = channel;  
        this.mActivity = activity;  
    }  
  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        String action = intent.getAction();  
  
        if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {  
            // Check to see if Wi-Fi is enabled and notify appropriate activity  
        } else if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action))  
            // Call WifiP2pManager.requestPeers() to get a list of current peer  
        } else if (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action))  
            // Respond to new connection or disconnections  
        } else if (WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION.equals(action))  
            // Respond to this device's wifi state changing  
    }  
}
```

## Creating a Wi-Fi Direct Application

Creating a Wi-Fi Direct application involves creating and registering a broadcast receiver for your application, discovering peers, connecting to a peer, and transferring data to a peer. The following sections describe how to do this.

### Initial setup

Before using the Wi-Fi Direct APIs, you must ensure that your application can access the hardware and that the device supports the Wi-Fi Direct protocol. If Wi-Fi Direct is supported, you can obtain an instance of `WifiP2pManager`, create and register your broadcast receiver, and begin using the Wi-Fi Direct APIs.

1. Request permission to use the Wi-Fi hardware on the device and also declare your application to have the correct minimum SDK version in the Android manifest:

```
<uses-sdk android:minSdkVersion="14" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

2. Check to see if Wi-Fi Direct is on and supported. A good place to check this is in your broadcast receiver when it receives the [WIFI\\_P2P\\_STATE\\_CHANGED\\_ACTION](#) intent. Notify your activity of the Wi-Fi Direct state and react accordingly:

```
@Override
public void onReceive(Context context, Intent intent) {
    ...
    String action = intent.getAction();
    if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {
        int state = intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE, -1);
        if (state == WifiP2pManager.WIFI_P2P_STATE_ENABLED) {
            // Wifi Direct is enabled
        } else {
            // Wi-Fi Direct is not enabled
        }
    }
    ...
}
```

3. In your activity's [onCreate\(\)](#) method, obtain an instance of [WifiP2pManager](#) and register your application with the Wi-Fi Direct framework by calling [initialize\(\)](#). This method returns a [WifiP2pManager.Channel](#), which is used to connect your application to the Wi-Fi Direct framework. You should also create an instance of your broadcast receiver with the [WifiP2pManager](#) and [WifiP2pManager.Channel](#) objects along with a reference to your activity. This allows your broadcast receiver to notify your activity of interesting events and update it accordingly. It also lets you manipulate the device's Wi-Fi state if necessary:

```
WifiP2pManager mManager;
Channel mChannel;
BroadcastReceiver mReceiver;
...
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    mManager = (WifiP2pManager) getSystemService(Context.WIFI_P2P_SERVICE);
    mChannel = mManager.initialize(this, getMainLooper(), null);
    mReceiver = new WiFiDirectBroadcastReceiver(mManager, mChannel, this);
    ...
}
```

4. Create an intent filter and add the same intents that your broadcast receiver checks for:

```
IntentFilter mIntentFilter;
...
```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    mIntentFilter = new IntentFilter();
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION)
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION)
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION)
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION)
    ...
}

```

5. Register the broadcast receiver in the [onResume\(\)](#) method of your activity and unregister it in the [onPause\(\)](#) method of your activity:

```

/* register the broadcast receiver with the intent values to be matched */
@Override
protected void onResume() {
    super.onResume();
    registerReceiver(mReceiver, mIntentFilter);
}
/* unregister the broadcast receiver */
@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(mReceiver);
}

```

When you have obtained a [WifiP2pManager.Channel](#) and set up a broadcast receiver, your application can make Wi-Fi Direct method calls and receive Wi-Fi Direct intents.

You can now implement your application and use the Wi-Fi Direct features by calling the methods in [WifiP2pManager](#). The next sections describe how to do common actions such as discovering and connecting to peers.

## Discovering peers

To discover peers that are available to connect to, call [discoverPeers\(\)](#) to detect available peers that are in range. The call to this function is asynchronous and a success or failure is communicated to your application with [onSuccess\(\)](#) and [onFailure\(\)](#) if you created a [WifiP2pManager.ActionListener](#). The [onSuccess\(\)](#) method only notifies you that the discovery process succeeded and does not provide any information about the actual peers that it discovered, if any:

```

mManager.discoverPeers(channel, new WifiP2pManager.ActionListener() {
    @Override
    public void onSuccess() {
        ...
    }
    @Override
    public void onFailure(int reasonCode) {
        ...
    }
}) ;

```

If the discovery process succeeds and detects peers, the system broadcasts the [WIFI\\_P2P\\_PEERS\\_CHANGED\\_ACTION](#) intent, which you can listen for in a broadcast receiver to obtain a list of peers. When your application receives the [WIFI\\_P2P\\_PEERS\\_CHANGED\\_ACTION](#) intent, you can request a list of the discovered peers with [requestPeers\(\)](#). The following code shows how to set this up:

```
PeerListListener myPeerListListener;
...
if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {

    // request available peers from the wifi p2p manager. This is an
    // asynchronous call and the calling activity is notified with a
    // callback on PeerListListener.onPeersAvailable()
    if (mManager != null) {
        mManager.requestPeers(mChannel, myPeerListListener);
    }
}
```

The [requestPeers\(\)](#) method is also asynchronous and can notify your activity when a list of peers is available with [onPeersAvailable\(\)](#), which is defined in the [WifiP2pManager.PeerListListener](#) interface. The [onPeersAvailable\(\)](#) method provides you with an [WifiP2pDeviceList](#), which you can iterate through to find the peer that you want to connect to.

## Connecting to peers

When you have figured out the device that you want to connect to after obtaining a list of possible peers, call the [connect\(\)](#) method to connect to the device. This method call requires a [WifiP2pConfig](#) object that contains the information of the device to connect to. You can be notified of a connection success or failure through the [WifiP2pManager.ActionListener](#). The following code shows you how to create a connection to a desired device:

```
//obtain a peer from the WifiP2pDeviceList
WifiP2pDevice device;
WifiP2pConfig config = new WifiP2pConfig();
config.deviceAddress = device.deviceAddress;
mManager.connect(mChannel, config, new ActionListener() {

    @Override
    public void onSuccess() {
        //success logic
    }

    @Override
    public void onFailure(int reason) {
        //failure logic
    }
});
```

## Transferring data

Once a connection is established, you can transfer data between the devices with sockets. The basic steps of transferring data are as follows:

1. Create a [ServerSocket](#). This socket waits for a connection from a client on a specified port and blocks until it happens, so do this in a background thread.
2. Create a client [Socket](#). The client uses the IP address and port of the server socket to connect to the server device.
3. Send data from the client to the server. When the client socket successfully connects to the server socket, you can send data from the client to the server with byte streams.
4. The server socket waits for a client connection (with the [accept \(\)](#) method). This call blocks until a client connects, so call this in another thread. When a connection happens, the server device can receive the data from the client. Carry out any actions with this data, such as saving it to a file or presenting it to the user.

The following example, modified from the [Wi-Fi Direct Demo](#) sample, shows you how to create this client-server socket communication and transfer JPEG images from a client to a server with a service. For a complete working example, compile and run the [Wi-Fi Direct Demo](#) sample.

```
public static class FileServerAsyncTask extends AsyncTask {

    private Context context;
    private TextView statusText;

    public FileServerAsyncTask(Context context, View statusText) {
        this.context = context;
        this.statusText = (TextView) statusText;
    }

    @Override
    protected String doInBackground(Void... params) {
        try {

            /**
             * Create a server socket and wait for client connections. This
             * call blocks until a connection is accepted from a client
             */
            ServerSocket serverSocket = new ServerSocket(8888);
            Socket client = serverSocket.accept();

            /**
             * If this code is reached, a client has connected and transferred
             * Save the input stream from the client as a JPEG file
             */
            final File f = new File(Environment.getExternalStorageDirectory() +
                    context.getPackageName() + "/wifip2pshared-" + System.currentTimeMillis()
                    + ".jpg");

            File dirs = new File(f.getParent());
            if (!dirs.exists())
                dirs.mkdirs();
            f.createNewFile();
            InputStream inputstream = client.getInputStream();
            copyFile(inputstream, new FileOutputStream(f));
            serverSocket.close();
            return f.getAbsolutePath();
        } catch (IOException e) {
            Log.e(WiFiDirectActivity.TAG, e.getMessage());
        }
    }
}
```

```

        return null;
    }

}

/**
 * Start activity that can handle the JPEG image
 */
@Override
protected void onPostExecute(String result) {
    if (result != null) {
        statusText.setText("File copied - " + result);
        Intent intent = new Intent();
        intent.setAction(android.content.Intent.ACTION_VIEW);
        intent.setDataAndType(Uri.parse("file://" + result), "image/*");
        context.startActivity(intent);
    }
}
}
}

```

On the client, connect to the server socket with a client socket and transfer data. This example transfers a JPEG file on the client device's file system.

```

Context context = this.getApplicationContext();
String host;
int port;
int len;
Socket socket = new Socket();
byte buf[] = new byte[1024];
...
try {
    /**
     * Create a client socket with the host,
     * port, and timeout information.
     */
    socket.bind(null);
    socket.connect((new InetSocketAddress(host, port)), 500);

    /**
     * Create a byte stream from a JPEG file and pipe it to the output stream
     * of the socket. This data will be retrieved by the server device.
     */
    OutputStream outputStream = socket.getOutputStream();
    ContentResolver cr = context.getContentResolver();
    InputStream inputStream = null;
    inputStream = cr.openInputStream(Uri.parse("path/to/picture.jpg"));
    while ((len = inputStream.read(buf)) != -1) {
        outputStream.write(buf, 0, len);
    }
    outputStream.close();
    inputStream.close();
} catch (FileNotFoundException e) {
    //catch logic
} catch (IOException e) {
    //catch logic
}

```

```
}

/***
 * Clean up any open sockets when done
 * transferring or if an exception occurred.
 */
finally {
    if (socket != null) {
        if (socket.isConnected()) {
            try {
                socket.close();
            } catch (IOException e) {
                //catch logic
            }
        }
    }
}
```

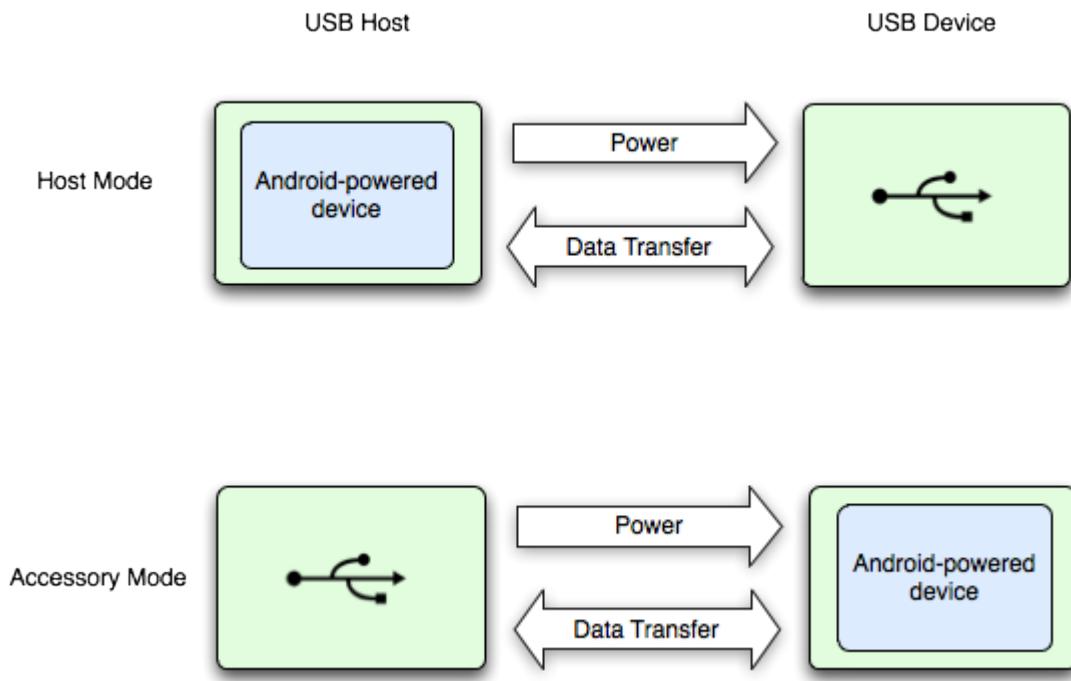
# USB Host and Accessory

## Topics

1. [USB Accessory](#)
2. [USB Host](#)

Android supports a variety of USB peripherals and Android USB accessories (hardware that implements the Android accessory protocol) through two modes: USB accessory and USB host. In USB accessory mode, the external USB hardware act as the USB hosts. Examples of accessories might include robotics controllers; docking stations; diagnostic and musical equipment; kiosks; card readers; and much more. This gives Android-powered devices that do not have host capabilities the ability to interact with USB hardware. Android USB accessories must be designed to work with Android-powered devices and must adhere to the [Android accessory communication protocol](#). In USB host mode, the Android-powered device acts as the host. Examples of devices include digital cameras, keyboards, mice, and game controllers. USB devices that are designed for a wide range of applications and environments can still interact with Android applications that can correctly communicate with the device.

Figure 1 shows the differences between the two modes. When the Android-powered device is in host mode, it acts as the USB host and powers the bus. When the Android-powered device is in USB accessory mode, the connected USB hardware (an Android USB accessory in this case) acts as the host and powers the bus.



**Figure 1.** USB Host and Accessory Modes

USB accessory and host modes are directly supported in Android 3.1 (API level 12) or newer platforms. USB accessory mode is also backported to Android 2.3.4 (API level 10) as an add-on library to support a broader range of devices. Device manufacturers can choose whether or not to include the add-on library on the device's system image.

**Note:** Support for USB host and accessory modes are ultimately dependant on the device's hardware, regardless of platform level. You can filter for devices that support USB host and accessory through a `<uses-feature>` element. See the USB [accessory](#) and [host](#) documentation for more details.

## Debugging considerations

When debugging applications that use USB accessory or host features, you most likely will have USB hardware connected to your Android-powered device. This will prevent you from having an adb connection to the Android-powered device via USB. You can still access adb over a network connection. To enable adb over a network connection:

1. Connect the Android-powered device via USB to your computer.
2. From your SDK platform-tools/ directory, enter `adb tcpip 5555` at the command prompt.
3. Enter `adb connect <device-ip-address>:5555` You should now be connected to the Android-powered device and can issue the usual adb commands like `adb logcat`.
4. To set your device to listen on USB, enter `adb usb`.

# USB Accessory

## In this document

1. [Choosing the Right USB Accessory APIs](#)
  1. [Installing the Google APIs add-on library](#)
2. [API Overview](#)
  1. [Usage differences between the add-on library and the platform APIs](#)
3. [Android Manifest Requirements](#)
4. [Working with accessories](#)
  1. [Discovering an accessory](#)
  2. [Obtaining permission to communicate with an accessory](#)
  3. [Communicating with an accessory](#)
  4. [Terminating communication with an accessory](#)

## See also

1. [Android USB Accessory Development Kit](#)

USB accessory mode allows users to connect USB host hardware specifically designed for Android-powered devices. The accessories must adhere to the Android accessory protocol outlined in the [Android Accessory Development Kit](#) documentation. This allows Android-powered devices that cannot act as a USB host to still interact with USB hardware. When an Android-powered device is in USB accessory mode, the attached Android USB accessory acts as the host, provides power to the USB bus, and enumerates connected devices. Android 3.1 (API level 12) supports USB accessory mode and the feature is also backported to Android 2.3.4 (API level 10) to enable support for a broader range of devices.

## Choosing the Right USB Accessory APIs

Although the USB accessory APIs were introduced to the platform in Android 3.1, they are also available in Android 2.3.4 using the Google APIs add-on library. Because these APIs were backported using an external library, there are two packages that you can import to support USB accessory mode. Depending on what Android-powered devices you want to support, you might have to use one over the other:

- `com.android.future.usb`: To support USB accessory mode in Android 2.3.4, the [Google APIs add-on library](#) includes the backported USB accessory APIs and they are contained in this namespace. Android 3.1 also supports importing and calling the classes within this namespace to support applications written with the add-on library. This add-on library is a thin wrapper around the [android.hardware.usb](#) accessory APIs and does not support USB host mode. If you want to support the widest range of devices that support USB accessory mode, use the add-on library and import this package. It is important to note that not all Android 2.3.4 devices are required to support the USB accessory feature. Each individual device manufacturer decides whether or not to support this capability, which is why you must declare it in your manifest file.
- `android.hardware.usb`: This namespace contains the classes that support USB accessory mode in Android 3.1. This package is included as part of the framework APIs, so Android 3.1 supports USB accessory mode without the use of an add-on library. Use this package if you only care about Android 3.1 or newer devices that have hardware support for USB accessory mode, which you can declare in your manifest file.

## Installing the Google APIs add-on library

If you want to install the add-on, you can do so by installing the Google APIs Android API 10 package with the SDK Manager. See [Installing the Google APIs Add-on](#) for more information on installing the add-on library.

## API Overview

Because the add-on library is a wrapper for the framework APIs, the classes that support the USB accessory feature are similar. You can use the reference documentation for the `android.hardware.usb` even if you are using the add-on library.

**Note:** There is, however, a minor [usage difference](#) between the add-on library and framework APIs that you should be aware of.

The following table describes the classes that support the USB accessory APIs:

Class	Description
<a href="#">UsbManager</a>	Allows you to enumerate and communicate with connected USB accessories.
<a href="#">UsbAccessory</a>	Represents a USB accessory and contains methods to access its identifying information.

## Usage differences between the add-on library and platform APIs

There are two usage differences between using the Google APIs add-on library and the platform APIs.

If you are using the add-on library, you must obtain the [UsbManager](#) object in the following manner:

```
UsbManager manager = UsbManager.getInstance(this);
```

If you are not using the add-on library, you must obtain the [UsbManager](#) object in the following manner:

```
UsbManager manager = (UsbManager) getSystemService(Context.USB_SERVICE);
```

When you filter for a connected accessory with an intent filter, the [UsbAccessory](#) object is contained inside the intent that is passed to your application. If you are using the add-on library, you must obtain the [UsbAccessory](#) object in the following manner:

```
UsbAccessory accessory = UsbManager.getAccessory(intent);
```

If you are not using the add-on library, you must obtain the [UsbAccessory](#) object in the following manner:

```
UsbAccessory accessory = (UsbAccessory) intent.getParcelableExtra(UsbManager.EXTRA_AC
```

## Android Manifest requirements

The following list describes what you need to add to your application's manifest file before working with the USB accessory APIs. The [manifest and resource file examples](#) show how to declare these items:

- Because not all Android-powered devices are guaranteed to support the USB accessory APIs, include a `<uses-feature>` element that declares that your application uses the `android.hardware.usb.accessory` feature.
- If you are using the [add-on library](#), add the `<uses-library>` element specifying `com.android.future.usb.accessory` for the library.

- Set the minimum SDK of the application to API Level 10 if you are using the add-on library or 12 if you are using the [android.hardware.usb](#) package.
- If you want your application to be notified of an attached USB accessory, specify an <intent-filter> and <meta-data> element pair for the android.hardware.usb.action.USB\_ACCESSORY\_ATTACHED intent in your main activity. The <meta-data> element points to an external XML resource file that declares identifying information about the accessory that you want to detect.

In the XML resource file, declare <usb-accessory> elements for the accessories that you want to filter. Each <usb-accessory> can have the following attributes:

- manufacturer
- model
- version

Save the resource file in the `res/xml/` directory. The resource file name (without the `.xml` extension) must be the same as the one you specified in the <meta-data> element. The format for the XML resource file is also shown in the [example](#) below.

## Manifest and resource file examples

The following example shows a sample manifest and its corresponding resource file:

```
<manifest ...>
    <uses-feature android:name="android.hardware.usb.accessory" />

    <uses-sdk android:minSdkVersion="<version>" />
    ...
<application>
    <uses-library android:name="com.android.future.usb.accessory" />
    <activity ...>
        ...
        <intent-filter>
            <action android:name="android.hardware.usb.action.USB_ACCESSORY_ATTACHED" />
        </intent-filter>

        <meta-data android:name="android.hardware.usb.action.USB_ACCESSORY_ATTACHED"
                  android:resource="@xml/accessory_filter" />
    </activity>
</application>
</manifest>
```

In this case, the following resource file should be saved in `res/xml/accessory_filter.xml` and specifies that any accessory that has the corresponding model, manufacturer, and version should be filtered. The accessory sends these attributes the Android-powered device:

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

<resources>
    <usb-accessory model="DemoKit" manufacturer="Google" version="1.0"/>
</resources>
```

# Working with Accessories

When users connect USB accessories to an Android-powered device, the Android system can determine whether your application is interested in the connected accessory. If so, you can set up communication with the accessory if desired. To do this, your application has to:

1. Discover connected accessories by using an intent filter that filters for accessory attached events or by enumerating connected accessories and finding the appropriate one.
2. Ask the user for permission to communicate with the accessory, if not already obtained.
3. Communicate with the accessory by reading and writing data on the appropriate interface endpoints.

## Discovering an accessory

Your application can discover accessories by either using an intent filter to be notified when the user connects an accessory or by enumerating accessories that are already connected. Using an intent filter is useful if you want to be able to have your application automatically detect a desired accessory. Enumerating connected accessories is useful if you want to get a list of all connected accessories or if your application did not filter for an intent.

### Using an intent filter

To have your application discover a particular USB accessory, you can specify an intent filter to filter for the `android.hardware.usb.action.USB_ACCESSORY_ATTACHED` intent. Along with this intent filter, you need to specify a resource file that specifies properties of the USB accessory, such as manufacturer, model, and version. When users connect an accessory that matches your accessory filter,

The following example shows how to declare the intent filter:

```
<activity ...>
    ...
    <intent-filter>
        <action android:name="android.hardware.usb.action.USB_ACCESSORY_ATTACHED" />
    </intent-filter>

    <meta-data android:name="android.hardware.usb.action.USB_ACCESSORY_ATTACHED"
              android:resource="@xml/accessory_filter" />
</activity>
```

The following example shows how to declare the corresponding resource file that specifies the USB accessories that you're interested in:

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

<resources>
    <usb-accessory manufacturer="Google, Inc." model="DemoKit" version="1.0" />
</resources>
```

In your activity, you can obtain the `UsbAccessory` that represents the attached accessory from the intent like this (with the add-on library):

```
UsbAccessory accessory = UsbManager.getAccessory(intent);
```

or like this (with the platform APIs):

```
UsbAccessory accessory = (UsbAccessory) intent.getParcelableExtra(UsbManager.EXTRA_
```

## Enumerating accessories

You can have your application enumerate accessories that have identified themselves while your application is running.

Use the [getAccessoryList\(\)](#) method to get an array all the USB accessories that are connected:

```
UsbManager manager = (UsbManager) getSystemService(Context.USB_SERVICE);  
UsbAccessory[] accessoryList = manager.getAccessoryList();
```

**Note:** Currently, only one connected accessory is supported at one time, but the API is designed to support multiple accessories in the future.

## Obtaining permission to communicate with an accessory

Before communicating with the USB accessory, your application must have permission from your users.

**Note:** If your application [uses an intent filter](#) to discover accessories as they're connected, it automatically receives permission if the user allows your application to handle the intent. If not, you must request permission explicitly in your application before connecting to the accessory.

Explicitly asking for permission might be necessary in some situations such as when your application enumerates accessories that are already connected and then wants to communicate with one. You must check for permission to access an accessory before trying to communicate with it. If not, you will receive a runtime error if the user denied permission to access the accessory.

To explicitly obtain permission, first create a broadcast receiver. This receiver listens for the intent that gets broadcast when you call [requestPermission\(\)](#). The call to [requestPermission\(\)](#) displays a dialog to the user asking for permission to connect to the accessory. The following sample code shows how to create the broadcast receiver:

```
private static final String ACTION_USB_PERMISSION =  
    "com.android.example.USB_PERMISSION";  
private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {  
  
    public void onReceive(Context context, Intent intent) {  
        String action = intent.getAction();  
        if (ACTION_USB_PERMISSION.equals(action)) {  
            synchronized (this) {  
                UsbAccessory accessory = (UsbAccessory) intent.getParcelableExt  
  
                if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED,  
                    false)) {  
                    if (accessory != null){  
                        //call method to set up accessory communication  
                    }  
                }  
                else {  
                    Log.d(TAG, "permission denied for accessory " + accessory);  
                }  
            }  
        }  
    }  
}
```

```
    }  
};
```

To register the broadcast receiver, put this in your `onCreate()` method in your activity:

```
UsbManager mUsbManager = (UsbManager) getSystemService(Context.USB_SERVICE);  
private static final String ACTION_USB_PERMISSION =  
    "com.android.example.USB_PERMISSION";  
...  
mPermissionIntent = PendingIntent.getBroadcast(this, 0, new Intent(ACTION_USB_PERMISSION),  
IntentFilter filter = new IntentFilter(ACTION_USB_PERMISSION);  
registerReceiver(mUsbReceiver, filter);
```

To display the dialog that asks users for permission to connect to the accessory, call the [requestPermissions\(\)](#) method:

```
UsbAccessory accessory;  
...  
mUsbManager.requestPermission(accessory, mPermissionIntent);
```

When users reply to the dialog, your broadcast receiver receives the intent that contains the [EXTRA\\_PERMISSION\\_GRANTED](#) extra, which is a boolean representing the answer. Check this extra for a value of true before connecting to the accessory.

## Communicating with an accessory

You can communicate with the accessory by using the [UsbManager](#) to obtain a file descriptor that you can set up input and output streams to read and write data to descriptor. The streams represent the accessory's input and output bulk endpoints. You should set up the communication between the device and accessory in another thread, so you don't lock the main UI thread. The following example shows how to open an accessory to communicate with:

```
UsbAccessory mAccessory;  
ParcelFileDescriptor mFileDescriptor;  
FileInputStream mInputStream;  
FileOutputStream mOutputStream;  
...  
private void openAccessory() {  
    Log.d(TAG, "openAccessory: " + accessory);  
    mFileDescriptor = mUsbManager.openAccessory(mAccessory);  
    if (mFileDescriptor != null) {  
        FileDescriptor fd = mFileDescriptor.getFileDescriptor();  
        mInputStream = new FileInputStream(fd);  
        mOutputStream = new FileOutputStream(fd);  
        Thread thread = new Thread(null, this, "AccessoryThread");  
        thread.start();  
    }  
}
```

In the thread's `run()` method, you can read and write to the accessory by using the [FileInputStream](#) or [FileOutputStream](#) objects. When reading data from an accessory with a [FileInputStream](#) object, ensure that the buffer that you use is big enough to store the USB packet data. The Android accessory protocol

supports packet buffers up to 16384 bytes, so you can choose to always declare your buffer to be of this size for simplicity.

**Note:** At a lower level, the packets are 64 bytes for USB full-speed accessories and 512 bytes for USB high-speed accessories. The Android accessory protocol bundles the packets together for both speeds into one logical packet for simplicity.

For more information about using threads in Android, see [Processes and Threads](#).

## Terminating communication with an accessory

When you are done communicating with an accessory or if the accessory was detached, close the file descriptor that you opened by calling [close\(\)](#). To listen for detached events, create a broadcast receiver like below:

```
BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {  
    public void onReceive(Context context, Intent intent) {  
        String action = intent.getAction();  
  
        if (UsbManager.ACTION_USB_ACCESSORY_DETACHED.equals(action)) {  
            UsbAccessory accessory = (UsbAccessory)intent.getParcelableExtra(Us  
                if (accessory != null) {  
                    // call your method that cleans up and closes communication with  
                }  
            }  
        }  
    } ;
```

Creating the broadcast receiver within the application, and not the manifest, allows your application to only handle detached events while it is running. This way, detached events are only sent to the application that is currently running and not broadcast to all applications.

# USB Host

## In this document

1. [API Overview](#)
2. [Android Manifest Requirements](#)
3. [Working with devices](#)
  1. [Discovering a device](#)
  2. [Obtaining permission to communicate with a device](#)
  3. [Communicating with a device](#)
  4. [Terminating communication with a device](#)

## Related Samples

1. [AdbTest](#)
2. [MissleLauncher](#)

When your Android-powered device is in USB host mode, it acts as the USB host, powers the bus, and enumerates connected USB devices. USB host mode is supported in Android 3.1 and higher.

## API Overview

Before you begin, it is important to understand the classes that you need to work with. The following table describes the USB host APIs in the [android.hardware.usb](#) package.

**Table 1.** USB Host APIs

Class	Description
<a href="#">UsbManager</a>	Allows you to enumerate and communicate with connected USB devices.
<a href="#">UsbDevice</a>	Represents a connected USB device and contains methods to access its identifying information, interfaces, and endpoints.
<a href="#">UsbInterface</a>	Represents an interface of a USB device, which defines a set of functionality for the device. A device can have one or more interfaces on which to communicate on.
<a href="#">UsbEndpoint</a>	Represents an interface endpoint, which is a communication channel for this interface. An interface can have one or more endpoints, and usually has input and output endpoints for two-way communication with the device.
<a href="#">UsbDeviceConnection</a>	Represents a connection to the device, which transfers data on endpoints. This class allows you to send data back and forth synchronously or asynchronously.
<a href="#">UsbRequest</a>	Represents an asynchronous request to communicate with a device through a <a href="#">UsbDeviceConnection</a> .
<a href="#">UsbConstants</a>	Defines USB constants that correspond to definitions in linux/usb/ch9.h of the Linux kernel.

In most situations, you need to use all of these classes ([UsbRequest](#) is only required if you are doing asynchronous communication) when communicating with a USB device. In general, you obtain a [UsbManager](#) to retrieve the desired [UsbDevice](#). When you have the device, you need to find the appropriate [UsbInterface](#) and the [UsbEndpoint](#) of that interface to communicate on. Once you obtain the correct endpoint, open a [UsbDeviceConnection](#) to communicate with the USB device.

# Android Manifest Requirements

The following list describes what you need to add to your application's manifest file before working with the USB host APIs:

- Because not all Android-powered devices are guaranteed to support the USB host APIs, include a `<uses-feature>` element that declares that your application uses the `android.hardware.usb.host` feature.
- Set the minimum SDK of the application to API Level 12 or higher. The USB host APIs are not present on earlier API levels.
- If you want your application to be notified of an attached USB device, specify an `<intent-filter>` and `<meta-data>` element pair for the `android.hardware.usb.action.USB_DEVICE_ATTACHED` intent in your main activity. The `<meta-data>` element points to an external XML resource file that declares identifying information about the device that you want to detect.

In the XML resource file, declare `<usb-device>` elements for the USB devices that you want to filter. The following list describes the attributes of `<usb-device>`. In general, use vendor and product ID if you want to filter for a specific device and use class, subclass, and protocol if you want to filter for a group of USB devices, such as mass storage devices or digital cameras. You can specify none or all of these attributes. Specifying no attributes matches every USB device, so only do this if your application requires it:

- `vendor-id`
- `product-id`
- `class`
- `subclass`
- `protocol` (device or interface)

Save the resource file in the `res/xml/` directory. The resource file name (without the `.xml` extension) must be the same as the one you specified in the `<meta-data>` element. The format for the XML resource file is in the [example](#) below.

## Manifest and resource file examples

The following example shows a sample manifest and its corresponding resource file:

```
<manifest ...>
    <uses-feature android:name="android.hardware.usb.host" />
    <uses-sdk android:minSdkVersion="12" />
    ...
    <application>
        <activity ...>
            ...
            <intent-filter>
                <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
                <meta-data android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
                    android:resource="@xml/device_filter" />
            </intent-filter>

        </activity>
    </application>
</manifest>
```

In this case, the following resource file should be saved in `res/xml/device_filter.xml` and specifies that any USB device with the specified attributes should be filtered:

```
<?xml version="1.0" encoding="utf-8"?>  
  
<resources>  
    <usb-device vendor-id="1234" product-id="5678" class="255" subclass="66" pr</resources>
```

## Working with Devices

When users connect USB devices to an Android-powered device, the Android system can determine whether your application is interested in the connected device. If so, you can set up communication with the device if desired. To do this, your application has to:

1. Discover connected USB devices by using an intent filter to be notified when the user connects a USB device or by enumerating USB devices that are already connected.
2. Ask the user for permission to connect to the USB device, if not already obtained.
3. Communicate with the USB device by reading and writing data on the appropriate interface endpoints.

### Discovering a device

Your application can discover USB devices by either using an intent filter to be notified when the user connects a device or by enumerating USB devices that are already connected. Using an intent filter is useful if you want to be able to have your application automatically detect a desired device. Enumerating connected USB devices is useful if you want to get a list of all connected devices or if your application did not filter for an intent.

### Using an intent filter

To have your application discover a particular USB device, you can specify an intent filter to filter for the `android.hardware.usb.action.USB_DEVICE_ATTACHED` intent. Along with this intent filter, you need to specify a resource file that specifies properties of the USB device, such as product and vendor ID. When users connect a device that matches your device filter, the system presents them with a dialog that asks if they want to start your application. If users accept, your application automatically has permission to access the device until the device is disconnected.

The following example shows how to declare the intent filter:

```
<activity ...>  
...  
    <intent-filter>  
        <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />  
        <meta-data android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"  
            android:resource="@xml/device_filter" />  
    </intent-filter>  
</activity>
```

The following example shows how to declare the corresponding resource file that specifies the USB devices that you're interested in:

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

```
<resources>
    <usb-device vendor-id="1234" product-id="5678" />
</resources>
```

In your activity, you can obtain the [UsbDevice](#) that represents the attached device from the intent like this:

```
UsbDevice device = (UsbDevice) intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
```

## Enumerating devices

If your application is interested in inspecting all of the USB devices currently connected while your application is running, it can enumerate devices on the bus. Use the [getDeviceList\(\)](#) method to get a hash map of all the USB devices that are connected. The hash map is keyed by the USB device's name if you want to obtain a device from the map.

```
UsbManager manager = (UsbManager) getSystemService(Context.USB_SERVICE);
...
HashMap<String, UsbDevice> deviceList = manager.getDeviceList();
UsbDevice device = deviceList.get("deviceName");
```

If desired, you can also just obtain an iterator from the hash map and process each device one by one:

```
UsbManager manager = (UsbManager) getSystemService(Context.USB_SERVICE);
...
HashMap<String, UsbDevice> deviceList = manager.getDeviceList();
Iterator<UsbDevice> deviceIterator = deviceList.values().iterator();
while(deviceIterator.hasNext()){
    UsbDevice device = deviceIterator.next()
    //your code
}
```

## Obtaining permission to communicate with a device

Before communicating with the USB device, your applicaton must have permission from your users.

**Note:** If your application [uses an intent filter](#) to discover USB devices as they're connected, it automatically receives permission if the user allows your application to handle the intent. If not, you must request permission explicitly in your application before connecting to the device.

Explicitly asking for permission might be neccessary in some situations such as when your application enumerates USB devices that are already connected and then wants to communicate with one. You must check for permission to access a device before trying to communicate with it. If not, you will receive a runtime error if the user denied permission to access the device.

To explicitly obtain permission, first create a broadcast receiver. This receiver listens for the intent that gets broadcast when you call [requestPermission\(\)](#). The call to [requestPermission\(\)](#) displays a dialog to the user asking for permission to connect to the device. The following sample code shows how to create the broadcast receiver:

```
private static final String ACTION_USB_PERMISSION =
    "com.android.example.USB_PERMISSION";
private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {

    public void onReceive(Context context, Intent intent) {
```

```

        String action = intent.getAction();
        if (ACTION_USB_PERMISSION.equals(action)) {
            synchronized (this) {
                UsbDevice device = (UsbDevice) intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);

                if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED, false)) {
                    if(device != null){
                        //call method to set up device communication
                    }
                } else {
                    Log.d(TAG, "permission denied for device " + device);
                }
            }
        }
    }
}

```

To register the broadcast receiver, add this in your `onCreate()` method in your activity:

```

UsbManager mUsbManager = (UsbManager) getSystemService(Context.USB_SERVICE);
private static final String ACTION_USB_PERMISSION =
    "com.android.example.USB_PERMISSION";
...
mPermissionIntent = PendingIntent.getBroadcast(this, 0, new Intent(ACTION_USB_PERMISSION),
IntentFilter filter = new IntentFilter(ACTION_USB_PERMISSION);
registerReceiver(mUsbReceiver, filter);

```

To display the dialog that asks users for permission to connect to the device, call the [requestPermissions\(\)](#) method:

```

UsbDevice device;
...
mUsbManager.requestPermission(device, mPermissionIntent);

```

When users reply to the dialog, your broadcast receiver receives the intent that contains the [EXTRA\\_PERMISSION\\_GRANTED](#) extra, which is a boolean representing the answer. Check this extra for a value of true before connecting to the device.

## Communicating with a device

Communication with a USB device can be either synchronous or asynchronous. In either case, you should create a new thread on which to carry out all data transmissions, so you don't block the UI thread. To properly set up communication with a device, you need to obtain the appropriate [UsbInterface](#) and [UsbEndpoint](#) of the device that you want to communicate on and send requests on this endpoint with a [UsbDeviceConnection](#). In general, your code should:

- Check a [UsbDevice](#) object's attributes, such as product ID, vendor ID, or device class to figure out whether or not you want to communicate with the device.
- When you are certain that you want to communicate with the device, find the appropriate [UsbInterface](#) that you want to use to communicate along with the appropriate [UsbEndpoint](#) of that interface. Interfaces can have one or more endpoints, and commonly will have an input and output endpoint for two-way communication.
- When you find the correct endpoint, open a [UsbDeviceConnection](#) on that endpoint.

- Supply the data that you want to transmit on the endpoint with the [bulkTransfer\(\)](#) or [controlTransfer\(\)](#) method. You should carry out this step in another thread to prevent blocking the main UI thread. For more information about using threads in Android, see [Processes and Threads](#).

The following code snippet is a trivial way to do a synchronous data transfer. Your code should have more logic to correctly find the correct interface and endpoints to communicate on and also should do any transferring of data in a different thread than the main UI thread:

```
private Byte[] bytes
private static int TIMEOUT = 0;
private boolean forceClaim = true;

...
UsbInterface intf = device.getInterface(0);
UsbEndpoint endpoint = intf.getEndpoint(0);
UsbDeviceConnection connection = mUsbManager.openDevice(device);
connection.claimInterface(intf, forceClaim);
connection.bulkTransfer(endpoint, bytes, bytes.length, TIMEOUT); //do in another thread
```

To send data asynchronously, use the [UsbRequest](#) class to [initialize](#) and [queue](#) an asynchronous request, then wait for the result with [requestWait\(\)](#).

For more information, see the [AdbTest sample](#), which shows how to do asynchronous bulk transfers, and the [MissleLauncher sample](#), which shows how to listen on an interrupt endpoint asynchronously.

## Terminating communication with a device

When you are done communicating with a device or if the device was detached, close the [UsbInterface](#) and [UsbDeviceConnection](#) by calling [releaseInterface\(\)](#) and [close\(\)](#). To listen for detached events, create a broadcast receiver like below:

```
BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();

        if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
            UsbDevice device = (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
            if (device != null) {
                // call your method that cleans up and closes communication with the device
            }
        }
    }
};
```

Creating the broadcast receiver within the application, and not the manifest, allows your application to only handle detached events while it is running. This way, detached events are only sent to the application that is currently running and not broadcast to all applications.

# Session Initiation Protocol

## In this document

1. [Requirements and Limitations](#)
2. [Classes and Interfaces](#)
3. [Creating the Manifest](#)
4. [Creating a SIP Manager](#)
5. [Registering with a SIP Server](#)
6. [Making an Audio Call](#)
7. [Receiving Calls](#)
8. [Testing SIP Applications](#)

## Key classes

1. [SipManager](#)
2. [SipProfile](#)
3. [SipAudioCall](#)

## Related samples

1. [SipDemo](#)

Android provides an API that supports the Session Initiation Protocol (SIP). This lets you add SIP-based internet telephony features to your applications. Android includes a full SIP protocol stack and integrated call management services that let applications easily set up outgoing and incoming voice calls, without having to manage sessions, transport-level communication, or audio record or playback directly.

Here are examples of the types of applications that might use the SIP API:

- Video conferencing.
- Instant messaging.

## Requirements and Limitations

Here are the requirements for developing a SIP application:

- You must have a mobile device that is running Android 2.3 or higher.
- SIP runs over a wireless data connection, so your device must have a data connection (with a mobile data service or Wi-Fi). This means that you can't test on AVD—you can only test on a physical device. For details, see [Testing SIP Applications](#).
- Each participant in the application's communication session must have a SIP account. There are many different SIP providers that offer SIP accounts.

## SIP API Classes and Interfaces

Here is a summary of the classes and one interface (`SipRegistrationListener`) that are included in the Android SIP API:

Class/Interface	Description
<a href="#">SipAudioCall</a>	Handles an Internet audio call over SIP.
<a href="#">SipAudioCall.Listener</a>	Listener for events relating to a SIP call, such as when a call is being received ("on ringing") or a call is outgoing ("on calling").
<a href="#">SipErrorCode</a>	Defines error codes returned during SIP actions.
<a href="#">SipManager</a>	Provides APIs for SIP tasks, such as initiating SIP connections, and provides access to related SIP services.
<a href="#">SipProfile</a>	Defines a SIP profile, including a SIP account, domain and server information.
<a href="#">SipProfile.Builder</a>	Helper class for creating a SipProfile.
<a href="#">SipSession</a>	Represents a SIP session that is associated with a SIP dialog or a standalone transaction not within a dialog.
<a href="#">SipSession.Listener</a>	Listener for events relating to a SIP session, such as when a session is being registered ("on registering") or a call is outgoing ("on calling").
<a href="#">SipSession.State</a>	Defines SIP session states, such as "registering", "outgoing call", and "in call".
<a href="#">SipRegistrationListener</a>	An interface that is a listener for SIP registration events.

## Creating the Manifest

If you are developing an application that uses the SIP API, remember that the feature is supported only on Android 2.3 (API level 9) and higher versions of the platform. Also, among devices running Android 2.3 (API level 9) or higher, not all devices will offer SIP support.

To use SIP, add the following permissions to your application's manifest:

- android.permission.USE\_SIP
- android.permission.INTERNET

To ensure that your application can only be installed on devices that are capable of supporting SIP, add the following to your application's manifest:

- <uses-sdk android:minSdkVersion="9" />. This indicates that your application requires Android 2.3 or higher. For more information, see [API Levels](#) and the documentation for the <uses-sdk> element.

To control how your application is filtered from devices that do not support SIP (for example, on Google Play), add the following to your application's manifest:

- <uses-feature android:name="android.hardware.sip.voip" />. This states that your application uses the SIP API. The declaration should include an android:required attribute that indicates whether you want the application to be filtered from devices that do not offer SIP support. Other <uses-feature> declarations may also be needed, depending on your implementation. For more information, see the documentation for the <uses-feature> element.

If your application is designed to receive calls, you must also define a receiver ([BroadcastReceiver](#) subclass) in the application's manifest:

- <receiver android:name=".IncomingCallReceiver" android:label="Call Receiver"/>

Here are excerpts from the **SipDemo** manifest:

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.sip">
    ...
        <receiver android:name=".IncomingCallReceiver" android:label="Call Receiver" />
    ...
        <uses-sdk android:minSdkVersion="9" />
        <uses-permission android:name="android.permission.USE_SIP" />
        <uses-permission android:name="android.permission.INTERNET" />
    ...
        <uses-feature android:name="android.hardware.sip.voip" android:required="true" />
        <uses-feature android:name="android.hardware.wifi" android:required="true" />
        <uses-feature android:name="android.hardware.microphone" android:required="true" />
</manifest>

```

## Creating a SipManager

To use the SIP API, your application must create a [SipManager](#) object. The [SipManager](#) takes care of the following in your application:

- Initiating SIP sessions.
- Initiating and receiving calls.
- Registering and unregistering with a SIP provider.
- Verifying session connectivity.

You instantiate a new [SipManager](#) as follows:

```

public SipManager mSipManager = null;
...
if(mSipManager == null) {
    mSipManager = SipManager.newInstance(this);
}

```

## Registering with a SIP Server

A typical Android SIP application involves one or more users, each of whom has a SIP account. In an Android SIP application, each SIP account is represented by a [SipProfile](#) object.

A [SipProfile](#) defines a SIP profile, including a SIP account, and domain and server information. The profile associated with the SIP account on the device running the application is called the *local profile*. The profile that the session is connected to is called the *peer profile*. When your SIP application logs into the SIP server with the local [SipProfile](#), this effectively registers the device as the location to send SIP calls to for your SIP address.

This section shows how to create a [SipProfile](#), register it with a SIP server, and track registration events.

You create a [SipProfile](#) object as follows:

```

public SipProfile mSipProfile = null;
...
SipProfile.Builder builder = new SipProfile.Builder(username, domain);

```

```
builder.setPassword(password);
mSipProfile = builder.build();
```

The following code excerpt opens the local profile for making calls and/or receiving generic SIP calls. The caller can make subsequent calls through `mSipManager.makeAudioCall`. This excerpt also sets the action `android.SipDemo.INCOMING_CALL`, which will be used by an intent filter when the device receives a call (see [Setting up an intent filter to receive calls](#)). This is the registration step:

```
Intent intent = new Intent();
intent.setAction("android.SipDemo.INCOMING_CALL");
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0, intent, Intent.FLAG_ACTIVITY_NEW_TASK);
mSipManager.open(mSipProfile, pendingIntent, null);
```

Finally, this code sets a `SipRegistrationListener` on the [SipManager](#). This tracks whether the [SipProfile](#) was successfully registered with your SIP service provider:

```
mSipManager.setRegistrationListener(mSipProfile.getUriString(), new SipRegistrationListener() {
    public void onRegistering(String localProfileUri) {
        updateStatus("Registering with SIP Server...");
    }

    public void onRegistrationDone(String localProfileUri, long expiryTime) {
        updateStatus("Ready");
    }

    public void onRegistrationFailed(String localProfileUri, int errorCode,
                                    String errorMessage) {
        updateStatus("Registration failed. Please check settings.");
    }
})
```

When your application is done using a profile, it should close it to free associated objects into memory and unregister the device from the server. For example:

```
public void closeLocalProfile() {
    if (mSipManager == null) {
        return;
    }
    try {
        if (mSipProfile != null) {
            mSipManager.close(mSipProfile.getUriString());
        }
    } catch (Exception ee) {
        Log.d("WalkieTalkieActivity/onDestroy", "Failed to close local profile.");
    }
}
```

## Making an Audio Call

To make an audio call, you must have the following in place:

- A [SipProfile](#) that is making the call (the "local profile"), and a valid SIP address to receive the call (the "peer profile").
- A [SipManager](#) object.

To make an audio call, you should set up a [SipAudioCall.Listener](#). Much of the client's interaction with the SIP stack happens through listeners. In this snippet, you see how the [SipAudioCall.Listener](#) sets things up after the call is established:

```
SipAudioCall.Listener listener = new SipAudioCall.Listener() {  
  
    @Override  
    public void onCallEstablished(SipAudioCall call) {  
        call.startAudio();  
        call.setSpeakerMode(true);  
        call.toggleMute();  
        ...  
    }  
  
    @Override  
    public void onCallEnded(SipAudioCall call) {  
        // Do something.  
    }  
};
```

Once you've set up the [SipAudioCall.Listener](#), you can make the call. The [SipManager](#) method `makeAudioCall` takes the following parameters:

- A local SIP profile (the caller).
- A peer SIP profile (the user being called).
- A [SipAudioCall.Listener](#) to listen to the call events from [SipAudioCall](#). This can be null, but as shown above, the listener is used to set things up once the call is established.
- The timeout value, in seconds.

For example:

```
call = mSipManager.makeAudioCall(mSipProfile.getUriString(), sipAddress, liste...
```

## Receiving Calls

To receive calls, a SIP application must include a subclass of [BroadcastReceiver](#) that has the ability to respond to an intent indicating that there is an incoming call. Thus, you must do the following in your application:

- In `AndroidManifest.xml`, declare a `<receiver>`. In **SipDemo**, this is `<receiver android:name=".IncomingCallReceiver" android:label="Call Receiver"/>`.
- Implement the receiver, which is a subclass of [BroadcastReceiver](#). In **SipDemo**, this is `IncomingCallReceiver`.
- Initialize the local profile ([SipProfile](#)) with a pending intent that fires your receiver when someone calls the local profile.
- Set up an intent filter that filters by the action that represents an incoming call. In **SipDemo**, this action is `android.SipDemo.INCOMING_CALL`.

### Subclassing BroadcastReceiver

To receive calls, your SIP application must subclass [BroadcastReceiver](#). The Android system handles incoming SIP calls and broadcasts an "incoming call" intent (as defined by the application) when it receives a call. Here is the subclassed [BroadcastReceiver](#) code from **SipDemo**. To see the full example, go to

[SipDemo sample](#), which is included in the SDK samples. For information on downloading and installing the SDK samples, see [Getting the Samples](#).

```
/** Listens for incoming SIP calls, intercepts and hands them off to WalkieTalkieActivity */
public class IncomingCallReceiver extends BroadcastReceiver {
    /**
     * Processes the incoming call, answers it, and hands it over to the
     * WalkieTalkieActivity.
     * @param context The context under which the receiver is running.
     * @param intent The intent being received.
     */
    @Override
    public void onReceive(Context context, Intent intent) {
        SipAudioCall incomingCall = null;
        try {
            SipAudioCall.Listener listener = new SipAudioCall.Listener() {
                @Override
                public void onRinging(SipAudioCall call, SipProfile caller) {
                    try {
                        call.answerCall(30);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            };
            WalkieTalkieActivity wtActivity = (WalkieTalkieActivity) context;
            incomingCall = wtActivity.mSipManager.takeAudioCall(intent, listener);
            incomingCall.answerCall(30);
            incomingCall.startAudio();
            incomingCall.setSpeakerMode(true);
            if(incomingCall.isMuted()) {
                incomingCall.toggleMute();
            }
            wtActivity.call = incomingCall;
            wtActivity.updateStatus(incomingCall);
        } catch (Exception e) {
            if (incomingCall != null) {
                incomingCall.close();
            }
        }
    }
}
```

## Setting up an intent filter to receive calls

When the SIP service receives a new call, it sends out an intent with the action string provided by the application. In SipDemo, this action string is `android.SipDemo.INCOMING_CALL`.

This code excerpt from **SipDemo** shows how the [SipProfile](#) object gets created with a pending intent based on the action string `android.SipDemo.INCOMING_CALL`. The `PendingIntent` object will perform a broadcast when the [SipProfile](#) receives a call:

```

public SipManager mSipManager = null;
public SipProfile mSipProfile = null;
...

Intent intent = new Intent();
intent.setAction("android.SipDemo.INCOMING_CALL");
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0, intent, Intent.FILL_IN_DATA);
mSipManager.open(mSipProfile, pendingIntent, null);

```

The broadcast will be intercepted by the intent filter, which will then fire the receiver (`IncomingCallReceiver`). You can specify an intent filter in your application's manifest file, or do it in code as in the **SipDemo** sample application's `onCreate()` method of the application's Activity:

```

public class WalkieTalkieActivity extends Activity implements View.OnTouchListener {
    ...
    public IncomingCallReceiver callReceiver;
    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        IntentFilter filter = new IntentFilter();
        filter.addAction("android.SipDemo.INCOMING_CALL");
        callReceiver = new IncomingCallReceiver();
        this.registerReceiver(callReceiver, filter);
        ...
    }
    ...
}

```

## Testing SIP Applications

To test SIP applications, you need the following:

- A mobile device that is running Android 2.3 or higher. SIP runs over wireless, so you must test on an actual device. Testing on AVD won't work.
- A SIP account. There are many different SIP providers that offer SIP accounts.
- If you are placing a call, it must also be to a valid SIP account.

To test a SIP application:

1. On your device, connect to wireless (**Settings > Wireless & networks > Wi-Fi > Wi-Fi settings**)
2. Set up your mobile device for testing, as described in [Developing on a Device](#).
3. Run your application on your mobile device, as described in [Developing on a Device](#).
4. If you are using Eclipse, you can view the application log output in Eclipse using LogCat (**Window > Show View > Other > Android > LogCat**).

# Text and Input

Use text services to add convenient features such as copy/paste and spell checking to your app. You can also develop your own text services to offer custom IMEs, dictionaries, and spelling checkers that you can distribute to users as applications.

## Blog Articles

### Add Voice Typing To Your IME

A new feature available in Android 4.0 is voice typing: the difference for users is that the recognition results appear in the text box while they are still speaking. If you are an IME developer, you can easily integrate with voice typing.

### Speech Input API for Android

We believe speech can fundamentally change the mobile experience. We would like to invite every Android application developer to consider integrating speech input capabilities via the Android SDK.

### Making Sense of Multitouch

The word "multitouch" gets thrown around quite a bit and it's not always clear what people are referring to. For some it's about hardware capability, for others it refers to specific gesture support in software. Whatever you decide to call it, today we're going to look at how to make your apps and views behave nicely with multiple fingers on the screen.

# Copy and Paste

## Quickview

- A clipboard-based framework for copying and pasting data.
- Supports both simple and complex data, including text strings, complex data structures, text and binary stream data, and application assets.
- Copies and pastes simple text directly to and from the clipboard.
- Copies and pastes complex data using a content provider.
- Requires API 11.

## In this document

1. [The Clipboard Framework](#)
2. [Clipboard Classes](#)
  1. [ClipboardManager](#)
  2. [ClipData, ClipDescription, and ClipData.Item](#)
  3. [ClipData convenience methods](#)
  4. [Coercing the clipboard data to text](#)
3. [Copying to the Clipboard](#)
4. [Pasting from the Clipboard](#)
  1. [Pasting plain text](#)
  2. [Pasting data from a content URI](#)
  3. [Pasting an Intent](#)
5. [Using Content Providers to Copy Complex Data](#)
  1. [Encoding an identifier on the URI](#)
  2. [Copying data structures](#)
  3. [Copying data streams](#)
6. [Designing Effective Copy/Paste Functionality](#)

## Key classes

1. [ClipboardManager](#)
2. [ClipData](#)
3. [ClipData.Item](#)
4. [ClipDescription](#)
5. [Uri](#)
6. [ContentProvider](#)
7. [Intent](#)

## Related Samples

1. [Note Pad sample application](#)

## See also

1. [Content Providers](#)

Android provides a powerful clipboard-based framework for copying and pasting. It supports both simple and complex data types, including text strings, complex data structures, text and binary stream data, and even appli-

cation assets. Simple text data is stored directly in the clipboard, while complex data is stored as a reference that the pasting application resolves with a content provider. Copying and pasting works both within an application and between applications that implement the framework.

Since a part of the framework uses content providers, this topic assumes some familiarity with the Android Content Provider API, which is described in the topic [Content Providers](#).

## The Clipboard Framework

When you use the clipboard framework, you put data into a clip object, and then put the clip object on the system-wide clipboard. The clip object can take one of three forms:

### Text

A text string. You put the string directly into the clip object, which you then put onto the clipboard. To paste the string, you get the clip object from the clipboard and copy the string to into your application's storage.

### URI

A [Uri](#) object representing any form of URI. This is primarily for copying complex data from a content provider. To copy data, you put a [Uri](#) object into a clip object and put the clip object onto the clipboard. To paste the data, you get the clip object, get the [Uri](#) object, resolve it to a data source such as a content provider, and copy the data from the source into your application's storage.

### Intent

An [Intent](#). This supports copying application shortcuts. To copy data, you create an Intent, put it into a clip object, and put the clip object onto the clipboard. To paste the data, you get the clip object and then copy the Intent object into your application's memory area.

The clipboard holds only one clip object at a time. When an application puts a clip object on the clipboard, the previous clip object disappears.

If you want to allow users to paste data into your application, you don't have to handle all types of data. You can examine the data on the clipboard before you give users the option to paste it. Besides having a certain data form, the clip object also contains metadata that tells you what MIME type or types are available. This metadata helps you decide if your application can do something useful with the clipboard data. For example, if you have an application that primarily handles text you may want to ignore clip objects that contain a URI or Intent.

You may also want to allow users to paste text regardless of the form of data on the clipboard. To do this, you can force the clipboard data into a text representation, and then paste this text. This is described in the section [Coercing the clipboard to text](#).

## Clipboard Classes

This section describes the classes used by the clipboard framework.

### ClipboardManager

In the Android system, the system clipboard is represented by the global [ClipboardManager](#) class. You do not instantiate this class directly; instead, you get a reference to it by invoking [getSystemService\(CLIPBOARD\\_SERVICE\)](#).

## ClipData, ClipData.Item, and ClipDescription

To add data to the clipboard, you create a [ClipData](#) object that contains both a description of the data and the data itself. The clipboard holds only one [ClipData](#) at a time. A [ClipData](#) contains a [ClipDescription](#) object and one or more [ClipData.Item](#) objects.

A [ClipDescription](#) object contains metadata about the clip. In particular, it contains an array of available MIME types for the clip's data. When you put a clip on the clipboard, this array is available to pasting applications, which can examine it to see if they can handle any of available the MIME types.

A [ClipData.Item](#) object contains the text, URI, or Intent data:

### Text

A [CharSequence](#).

### URI

A [Uri](#). This usually contains a content provider URI, although any URI is allowed. The application that provides the data puts the URI on the clipboard. Applications that want to paste the data get the URI from the clipboard and use it to access the content provider (or other data source) and retrieve the data.

### Intent

An [Intent](#). This data type allows you to copy an application shortcut to the clipboard. Users can then paste the shortcut into their applications for later use.

You can add more than one [ClipData.Item](#) object to a clip. This allows users to copy and paste multiple selections as a single clip. For example, if you have a list widget that allows the user to select more than one item at a time, you can copy all the items to the clipboard at once. To do this, you create a separate [ClipData.Item](#) for each list item, and then you add the [ClipData.Item](#) objects to the [ClipData](#) object.

## ClipData convenience methods

The [ClipData](#) class provides static convenience methods for creating a [ClipData](#) object with a single [ClipData.Item](#) object and a simple [ClipDescription](#) object:

### [newPlainText\(label, text\)](#)

Returns a [ClipData](#) object whose single [ClipData.Item](#) object contains a text string. The [ClipDescription](#) object's label is set to `label`. The single MIME type in [ClipDescription](#) is [MIME\\_TYPE\\_TEXT\\_PLAIN](#).

Use [newPlainText\(\)](#) to create a clip from a text string.

### [newUri\(resolver, label, URI\)](#)

Returns a [ClipData](#) object whose single [ClipData.Item](#) object contains a URI. The [ClipDescription](#) object's label is set to `label`. If the URI is a content URI ([Uri.getScheme\(\)](#) returns `content:`), the method uses the [ContentResolver](#) object provided in `resolver` to retrieve the available MIME types from the content provider and store them in [ClipDescription](#). For a URI that is not a content : URI, the method sets the MIME type to [MIMETYPE\\_TEXT\\_URLLIST](#).

Use [newUri\(\)](#) to create a clip from a URI, particularly a content : URI.

### [newIntent\(label, intent\)](#)

Returns a [ClipData](#) object whose single [ClipData.Item](#) object contains an [Intent](#). The [ClipDescription](#) object's label is set to `label`. The MIME type is set to [MIMETYPE\\_TEXT\\_INTENT](#).

Use `newIntent()` to create a clip from an Intent object.

## Coercing the clipboard data to text

Even if your application only handles text, you can copy non-text data from the clipboard by converting it with the method `ClipData.Item.coerceToText()`.

This method converts the data in `ClipData.Item` to text and returns a `CharSequence`. The value that `ClipData.Item.coerceToText()` returns is based on the form of data in `ClipData.Item`:

### Text

If `ClipData.Item` is text (`getText()` is not null), `coerceToText()` returns the text.

### URI

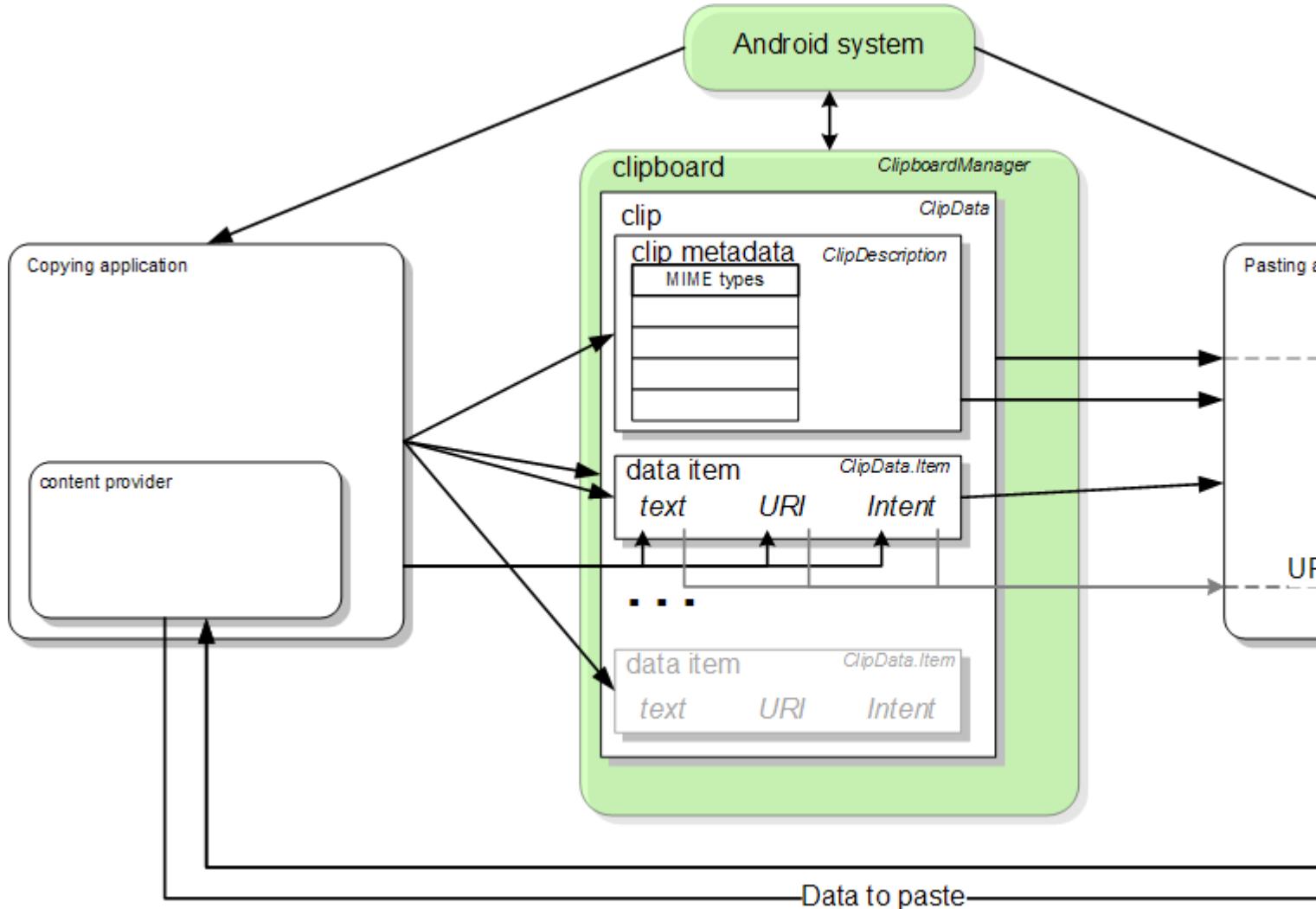
If `ClipData.Item` is a URI (`getUri()` is not null), `coerceToText()` tries to use it as a content URI:

- If the URI is a content URI and the provider can return a text stream, `coerceToText()` returns a text stream.
- If the URI is a content URI but the provider does not offer a text stream, `coerceToText()` returns a representation of the URI. The representation is the same as that returned by `Uri.toString()`.
- If the URI is not a content URI, `coerceToText()` returns a representation of the URI. The representation is the same as that returned by `Uri.toString()`.

### Intent

If `ClipData.Item` is an Intent (`getIntent()` is not null), `coerceToText()` converts it to an Intent URI and returns it. The representation is the same as that returned by `Intent.toUri(URI_INTENT_SCHEME)`.

The clipboard framework is summarized in Figure 1. To copy data, an application puts a `ClipData` object on the `ClipboardManager` global clipboard. The `ClipData` contains one or more `ClipData.Item` objects and one `ClipDescription` object. To paste data, an application gets the `ClipData`, gets its MIME type from the `ClipDescription`, and gets the data either from the `ClipData.Item` or from the content provider referred to by `ClipData.Item`.



**Figure 1.** The Android clipboard framework

## Copying to the Clipboard

As described previously, to copy data to the clipboard you get a handle to the global [ClipboardManager](#) object, create a [ClipData](#) object, add a [ClipDescription](#) and one or more [ClipData.Item](#) objects to it, and add the finished [ClipData](#) object to the [ClipboardManager](#) object. This is described in detail in the following procedure:

1. If you are copying data using a content URI, set up a content provider.

The [Note Pad](#) sample application is an example of using a content provider for copying and pasting. The [NotePadProvider](#) class implements the content provider. The [NotePad](#) class defines a contract between the provider and other applications, including the supported MIME types.

2. Get the system clipboard:

...

```
// if the user selects copy
case R.id.menu_copy:

    // Gets a handle to the clipboard service.
```

```
ClipboardManager clipboard = (ClipboardManager)
    getSystemService(Context.CLIPBOARD_SERVICE);
```

### 3. Copy the data to a new [ClipData](#) object:

- **For text**

```
// Creates a new text clip to put on the clipboard
ClipData clip = ClipData.newPlainText("simple text", "Hello, World!");
```

- **For a URI**

This snippet constructs a URI by encoding a record ID onto the content URI for the provider. This technique is covered in more detail in the section [Encoding an identifier on the URI](#):

```
// Creates a Uri based on a base Uri and a record ID based on the contact
// Declares the base URI string
private static final String CONTACTS = "content://com.example.contacts";

// Declares a path string for URIs that you use to copy data
private static final String COPY_PATH = "/copy";

// Declares the Uri to paste to the clipboard
Uri copyUri = Uri.parse(CONTACTS + COPY_PATH + "/" + lastName);

...

// Creates a new URI clip object. The system uses the anonymous getContact()
// get MIME types from provider. The clip object's label is "URI", as
// the Uri previously created.
ClipData clip = ClipData.newUri(getApplicationContext(), "URI", copyUri);
```

- **For an Intent**

This snippet constructs an Intent for an application and then puts it in the clip object:

```
// Creates the Intent
Intent appIntent = new Intent(this, com.example.demo.myapplication.*;

...

// Creates a clip object with the Intent in it. Its label is "Intent"
// the Intent object created previously
ClipData clip = ClipData.newIntent("Intent", appIntent);
```

### 4. Put the new clip object on the clipboard:

```
// Set the clipboard's primary clip.
clipboard.setPrimaryClip(clip);
```

# Pasting from the Clipboard

As described previously, you paste data from the clipboard by getting the global clipboard object, getting the clip object, looking at its data, and if possible copying the data from the clip object to your own storage. This section describes in detail how to do this for the three forms of clipboard data.

## Pasting plain text

To paste plain text, first get the global clipboard and verify that it can return plain text. Then get the clip object and copy its text to your own storage using [getText\(\)](#), as described in the following procedure:

1. Get the global [ClipboardManager](#) object using [getSystemService\(CLIPBOARD\\_SERVICE\)](#). Also declare a global variable to contain the pasted text:

```
ClipboardManager clipboard = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);
String pasteData = "";
```

2. Next, determine if you should enable or disable the "paste" option in the current Activity. You should verify that the clipboard contains a clip and that you can handle the type of data represented by the clip:

```
// Gets the ID of the "paste" menu item
MenuItem mPasteItem = menu.findItem(R.id.menu_paste);

// If the clipboard doesn't contain data, disable the paste menu item.
// If it does contain data, decide if you can handle the data.
if (!clipboard.hasPrimaryClip()) {

    mPasteItem.setEnabled(false);

} else if (!(clipboard.getPrimaryClipDescription().hasMimeType(MIMETYPE_PLAIN_TEXT))) {

    // This disables the paste menu item, since the clipboard has data
    mPasteItem.setEnabled(false);
} else {

    // This enables the paste menu item, since the clipboard contains data
    mPasteItem.setEnabled(true);
}
```

3. Copy the data from the clipboard. This point in the program is only reachable if the "paste" menu item is enabled, so you can assume that the clipboard contains plain text. You do not yet know if it contains a text string or a URI that points to plain text. The following snippet tests this, but it only shows the code for handling plain text:

```
// Responds to the user selecting "paste"
case R.id.menu_paste:

// Examines the item on the clipboard. If getText() does not return null,
// text. Assumes that this application can only handle one item at a time
ClipData.Item item = clipboard.getPrimaryClip().getItemAt(0);
```

```

// Gets the clipboard as text.
pasteData = item.getText();

// If the string contains data, then the paste operation is done
if (pasteData != null) {
    return;

// The clipboard does not contain text. If it contains a URI, attempts to
} else {
    Uri pasteUri = item.getUri();

    // If the URI contains something, try to get text from it
    if (pasteUri != null) {

        // calls a routine to resolve the URI and get data from it. This
        // presented here.
        pasteData = resolveUri(Uri);
        return;
    } else {

        // Something is wrong. The MIME type was plain text, but the clipboard
        // text or a Uri. Report an error.
        Log.e("Clipboard contains an invalid data type");
        return;
    }
}

```

## Pasting data from a content URI

If the [ClipData.Item](#) object contains a content URI and you have determined that you can handle one of its MIME types, create a [ContentResolver](#) and then call the appropriate content provider method to retrieve the data.

The following procedure describes how to get data from a content provider based on a content URI on the clipboard. It checks that a MIME type that the application can use is available from the provider:

1. Declare a global variable to contain the MIME type:

```

// Declares a MIME type constant to match against the MIME types offered by the provider
public static final String MIME_TYPE_CONTACT = "vnd.android.cursor.item/contact";

```

2. Get the global clipboard. Also get a content resolver so you can access the content provider:

```

// Gets a handle to the Clipboard Manager
ClipboardManager clipboard = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);

// Gets a content resolver instance
ContentResolver cr = getContentResolver();

```

3. Get the primary clip from the clipboard, and get its contents as a URI:

```

// Gets the clipboard data from the clipboard
ClipData clip = clipboard.getPrimaryClip();

```

```

if (clip != null) {

    // Gets the first item from the clipboard data
    ClipData.Item item = clip.getItemAt(0);

    // Tries to get the item's contents as a Uri
    Uri pasteUri = item.getUri();

```

4. Test to see if the URI is a content URI by calling [getUri\(Uri\)](#). This method returns null if Uri does not point to a valid content provider:

```

// If the clipboard contains a Uri reference
if (pasteUri != null) {

    // Is this a content Uri?
    String uriMimeType = cr.getType(pasteUri);

```

5. Test to see if the content provider supports a MIME type that the current application understands. If it does, call [ContentResolver.query\(\)](#) to get the data. The return value is a [Cursor](#):

```

// If the return value is not null, the Uri is a content Uri
if (uriMimeType != null) {

    // Does the content provider offer a MIME type that the current application
    if (uriMimeType.equals(MIME_TYPE_CONTACT)) {

        // Get the data from the content provider.
        Cursor pasteCursor = cr.query(uri, null, null, null, null);

        // If the Cursor contains data, move to the first record
        if (pasteCursor != null) {
            if (pasteCursor.moveToFirst()) {

                // get the data from the Cursor here. The code will vary based on the
                // format of the data model.
            }
        }

        // close the Cursor
        pasteCursor.close();
    }
}
}

```

## Pasting an Intent

To paste an Intent, first get the global clipboard. Examine the [ClipData.Item](#) object to see if it contains an Intent. Then call [getIntent\(\)](#) to copy the Intent to your own storage. The following snippet demonstrates this:

```

// Gets a handle to the Clipboard Manager
ClipboardManager clipboard = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);

```

```
// Checks to see if the clip item contains an Intent, by testing to see if getIntent()
Intent pasteIntent = clipboard.getPrimaryClip().getItemAt(0).getIntent();

if (pasteIntent != null) {
    // handle the Intent
} else {
    // ignore the clipboard, or issue an error if your application was expecting
    // on the clipboard
}
```

## Using Content Providers to Copy Complex Data

Content providers support copying complex data such as database records or file streams. To copy the data, you put a content URI on the clipboard. Pasting applications then get this URI from the clipboard and use it to retrieve database data or file stream descriptors.

Since the pasting application only has the content URI for your data, it needs to know which piece of data to retrieve. You can provide this information by encoding an identifier for the data on the URI itself, or you can provide a unique URI that will return the data you want to copy. Which technique you choose depends on the organization of your data.

The following sections describe how to set up URIs, how to provide complex data, and how to provide file streams. The descriptions assume that you are familiar with the general principles of content provider design.

### Encoding an identifier on the URI

A useful technique for copying data to the clipboard with a URI is to encode an identifier for the data on the URI itself. Your content provider can then get the identifier from the URI and use it to retrieve the data. The pasting application doesn't have to know that the identifier exists; all it has to do is get your "reference" (the URI plus the identifier) from the clipboard, give it your content provider, and get back the data.

You usually encode an identifier onto a content URI by concatenating it to the end of the URI. For example, suppose you define your provider URI as the following string:

```
"content://com.example.contacts"
```

If you want to encode a name onto this URI, you would use the following snippet:

```
String uriString = "content://com.example.contacts" + "/" + "Smith"

// uriString now contains content://com.example.contacts/Smith.

// Generates a Uri object from the string representation
Uri copyUri = Uri.parse(uriString);
```

If you are already using a content provider, you may want to add a new URI path that indicates the URI is for copying. For example, suppose you already have the following URI paths:

```
"content://com.example.contacts"/people
"content://com.example.contacts"/people/detail
"content://com.example.contacts"/people/images
```

You could add another path that is specific to copy URIs:

```
"content://com.example.contacts/copying"
```

You could then detect a "copy" URI by pattern-matching and handle it with code that is specific for copying and pasting.

You normally use the encoding technique if you're already using a content provider, internal database, or internal table to organize your data. In these cases, you have multiple pieces of data you want to copy, and presumably a unique identifier for each piece. In response to a query from the pasting application, you can look up the data by its identifier and return it.

If you don't have multiple pieces of data, then you probably don't need to encode an identifier. You can simply use a URI that is unique to your provider. In response to a query, your provider would return the data it currently contains.

Getting a single record by ID is used in the [Note Pad](#) sample application to open a note from the notes list. The sample uses the `_id` field from an SQL database, but you can have any numeric or character identifier you want.

## Copying data structures

You set up a content provider for copying and pasting complex data as a subclass of the [ContentProvider](#) component. You should also encode the URI you put on the clipboard so that it points to the exact record you want to provide. In addition, you have to consider the existing state of your application:

- If you already have a content provider, you can add to its functionality. You may only need to modify its [query\(\)](#) method to handle URIs coming from applications that want to paste data. You will probably want to modify the method to handle a "copy" URI pattern.
- If your application maintains an internal database, you may want to move this database into a content provider to facilitate copying from it.
- If you are not currently using a database, you can implement a simple content provider whose sole purpose is to offer data to applications that are pasting from the clipboard.

In the content provider, you will want to override at least the following methods:

### [query\(\)](#)

Pasting applications will assume that they can get your data by using this method with the URI you put on the clipboard. To support copying, you should have this method detect URIs that contain a special "copy" path. Your application can then create a "copy" URI to put on the clipboard, containing the copy path and a pointer to the exact record you want to copy.

### [getType\(\)](#)

This method should return the MIME type or types for the data you intend to copy. The method [newUri\(\)](#) calls [getType\(\)](#) in order to put the MIME types into the new [ClipData](#) object.

MIME types for complex data are described in the topic [Content Providers](#).

Notice that you don't have to have any of the other content provider methods such as [insert\(\)](#) or [update\(\)](#). A pasting application only needs to get your supported MIME types and copy data from your provider. If you already have these methods, they won't interfere with copy operations.

The following snippets demonstrate how to set up your application to copy complex data:

1. In the global constants for your application, declare a base URI string and a path that identifies URI strings you are using to copy data. Also declare a MIME type for the copied data:

```
// Declares the base URI string
private static final String CONTACTS = "content://com.example.contacts";

// Declares a path string for URIs that you use to copy data
private static final String COPY_PATH = "/copy";

// Declares a MIME type for the copied data
public static final String MIME_TYPE_CONTACT = "vnd.android.cursor.item/v
```

2. In the Activity from which users copy data, set up the code to copy data to the clipboard. In response to a copy request, put the URI on the clipboard:

```
public class MyCopyActivity extends Activity {

    ...

    // The user has selected a name and is requesting a copy.
    case R.id.menu_copy:

        // Appends the last name to the base URI
        // The name is stored in "lastName"
        uriString = CONTACTS + COPY_PATH + "/" + lastName;

        // Parses the string into a URI
        Uri copyUri = Uri.parse(uriString);

        // Gets a handle to the clipboard service.
        ClipboardManager clipboard = (ClipboardManager)
            getSystemService(Context.CLIPBOARD_SERVICE);

        ClipData clip = ClipData.newUri(getApplicationContext(), "URI", copyUri)

        // Set the clipboard's primary clip.
        clipboard.setPrimaryClip(clip);
```

3. In the global scope of your content provider, create a URI matcher and add a URI pattern that will match URIs you put on the clipboard:

```
public class MyCopyProvider extends ContentProvider {

    ...

    // A Uri Match object that simplifies matching content URIs to patterns.
    private static final UriMatcher sURIMatcher = new UriMatcher(UriMatcher.N

    // An integer to use in switching based on the incoming URI pattern
    private static final int GET_SINGLE_CONTACT = 0;

    ...

    // Adds a matcher for the content URI. It matches
```

```
// "content://com.example.contacts/copy/*"  
sUriMatcher.addURI(CONTACTS, "names/*", GET_SINGLE_CONTACT);
```

4. Set up the [query\(\)](#) method. This method can handle different URI patterns, depending on how you code it, but only the pattern for the clipboard copying operation is shown:

```
// Sets up your provider's query() method.  
public Cursor query(Uri uri, String[] projection, String selection, String  
String sortOrder) {  
  
    ...  
  
    // Switch based on the incoming content URI  
    switch (sUriMatcher.match(uri)) {  
  
        case GET_SINGLE_CONTACT:  
  
            // query and return the contact for the requested name. Here you  
            // the incoming URI, query the data model based on the last name,  
            // as a Cursor.  
  
            ...  
  
    }  
}
```

5. Set up the [getType\(\)](#) method to return an appropriate MIME type for copied data:

```
// Sets up your provider's getType() method.  
public String getType(Uri uri) {  
  
    ...  
  
    switch (sUriMatcher.match(uri)) {  
  
        case GET_SINGLE_CONTACT:  
  
            return (MIME_TYPE_CONTACT);  
    }  
}
```

The section [Pasting data from a content URI](#) describes how to get a content URI from the clipboard and use it to get and paste data.

## Copying data streams

You can copy and paste large amounts of text and binary data as streams. The data can have forms such as the following:

- Files stored on the actual device.
- Streams from sockets.
- Large amounts of data stored in a provider's underlying database system.

A content provider for data streams provides access to its data with a file descriptor object such as [Asset-FileDescriptor](#) instead of a [Cursor](#) object. The pasting application reads the data stream using this file descriptor.

To set up your application to copy a data stream with a provider, follow these steps:

1. Set up a content URI for the data stream you are putting on the clipboard. Options for doing this include the following:
  - Encode an identifier for the data stream onto the URI, as described in the section [Encoding an identifier on the URI](#), and then maintain a table in your provider that contains identifiers and the corresponding stream name.
  - Encode the stream name directly on the URI.
  - Use a unique URI that always returns the current stream from the provider. If you use this option, you have to remember to update your provider to point to a different stream whenever you copy the stream to the clipboard via the URI.
2. Provide a MIME type for each type of data stream you plan to offer. Pasting applications need this information to determine if they can paste the data on the clipboard.
3. Implement one of the [ContentProvider](#) methods that returns a file descriptor for a stream. If you encode identifiers on the content URI, use this method to determine which stream to open.
4. To copy the data stream to the clipboard, construct the content URI and place it on the clipboard.

To paste a data stream, an application gets the clip from the clipboard, gets the URI, and uses it in a call to a [ContentResolver](#) file descriptor method that opens the stream. The [ContentResolver](#) method calls the corresponding [ContentProvider](#) method, passing it the content URI. Your provider returns the file descriptor to [ContentResolver](#) method. The pasting application then has the responsibility to read the data from the stream.

The following list shows the most important file descriptor methods for a content provider. Each of these has a corresponding [ContentResolver](#) method with the string "Descriptor" appended to the method name; for example, the [ContentResolver](#) analog of [openAssetFile\(\)](#) is [openAssetFileDescriptor\(\)](#):

#### [openTypedAssetFile\(\)](#)

This method should return an asset file descriptor, but only if the provided MIME type is supported by the provider. The caller (the application doing the pasting) provides a MIME type pattern. The content provider (of the application that has copied a URI to the clipboard) returns an [AssetFileDescriptor](#) file handle if it can provide that MIME type, or throws an exception if it can not.

This method handles subsections of files. You can use it to read assets that the content provider has copied to the clipboard.

#### [openAssetFile\(\)](#)

This method is a more general form of [openTypedAssetFile\(\)](#). It does not filter for allowed MIME types, but it can read subsections of files.

#### [openFile\(\)](#)

This is a more general form of [openAssetFile\(\)](#). It can't read subsections of files.

You can optionally use the [openPipeHelper\(\)](#) method with your file descriptor method. This allows the pasting application to read the stream data in a background thread using a pipe. To use this method, you need to implement the [ContentProvider.PipeDataWriter](#) interface. An example of doing this is given in the [Note Pad](#) sample application, in the [openTypedAssetFile\(\)](#) method of `NotePadProvider.java`.

## Designing Effective Copy/Paste Functionality

To design effective copy and paste functionality for your application, remember these points:

- At any time, there is only one clip on the clipboard. A new copy operation by any application in the system overwrites the previous clip. Since the user may navigate away from your application and do a copy before returning, you can't assume that the clipboard contains the clip that the user previously copied in *your* application.
- The intended purpose of multiple [ClipData.Item](#) objects per clip is to support copying and pasting of multiple selections rather than different forms of reference to a single selection. You usually want all of the [ClipData.Item](#) objects in a clip to have the same form, that is, they should all be simple text, content URI, or [Intent](#), but not a mixture.
- When you provide data, you can offer different MIME representations. Add the MIME types you support to the [ClipDescription](#), and then implement the MIME types in your content provider.
- When you get data from the clipboard, your application is responsible for checking the available MIME types and then deciding which one, if any, to use. Even if there is a clip on the clipboard and the user requests a paste, your application is not required to do the paste. You *should* do the paste if the MIME type is compatible. You may choose to coerce the data on the clipboard to text using [coerceToText\(\)](#) if you choose. If your application supports more than one of the available MIME types, you can allow the user to choose which one to use.

# Creating an Input Method

## See also

1. [Onscreen Input Methods](#)
2. [Soft Keyboard sample](#)

An input method editor (IME) is a user control that enables users to enter text. Android provides an extensible input method framework that allows applications to provide users alternative input methods, such as on-screen keyboards or even speech input. Once installed, users can select which IME they want to use from the system settings and use it across the entire system; only one IME may be enabled at a time.

To add an IME to the Android system, you create an Android application containing a class that extends [InputMethodService](#). In addition, you usually create a "settings" activity that passes options to the IME service. You can also define a settings UI that's displayed as part of the system settings.

This article covers the following:

- The IME lifecycle.
- Declaring IME components in the application manifest.
- The IME API.
- Designing an IME UI.
- Sending text from an IME to an application.
- Working with IME subtypes.

If you haven't worked with IMEs before, you should read the introductory article [Onscreen Input Methods](#) first. Also, the Soft Keyboard sample app included in the SDK contains sample code that you can modify to start building your own IME.

## The IME Lifecycle

The following diagram describes the life cycle of an IME:



**Figure 1.** The life cycle of an IME.

The following sections describe how to implement the UI and code associated with an IME that follows this lifecycle.

# Declaring IME Components in the Manifest

In the Android system, an IME is an Android application that contains a special IME service. The application's manifest file must declare the service, request the necessary permissions, provide an intent filter that matches the action `action.view.InputMethod`, and provide metadata that defines characteristics of the IME. In addition, to provide a settings interface that allows the user to modify the behavior of the IME, you can define a "settings" activity that can be launched from System Settings.

The following snippet declares IME service. It requests the permission `BIND_INPUT_METHOD` to allow the service to connect the IME to the system, sets up an intent filter that matches the action `android.view.InputMethod`, and defines metadata for the IME:

```
<!-- Declares the input method service -->
<service android:name="FastInputIME"
    android:label="@string/fast_input_label"
    android:permission="android.permission.BIND_INPUT_METHOD">
    <intent-filter>
        <action android:name="android.view.InputMethod" />
    </intent-filter>
    <meta-data android:name="android.view.im" android:resource="@xml/method"
</service>
```

This next snippet declares the settings activity for the IME. It has an intent filter for `ACTION_MAIN` that indicates this activity is the main entry point for the IME application:

```
<!-- Optional: an activity for controlling the IME settings -->
<activity android:name="FastInputIMESettings"
    android:label="@string/fast_input_settings">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
    </intent-filter>
</activity>
```

You can also provide access to the IME's settings directly from its UI.

## The Input Method API

Classes specific to IMEs are found in the `android.inputmethodservice` and `android.view.inputmethod` packages. The `KeyEvent` class is important for handling keyboard characters.

The central part of an IME is a service component, a class that extends `InputMethodService`. In addition to implementing the normal service lifecycle, this class has callbacks for providing your IME's UI, handling user input, and delivering text to the field that currently has focus. By default, the `InputMethodService` class provides most of the implementation for managing the state and visibility of the IME and communicating with the current input field.

The following classes are also important:

### [BaseInputConnection](#)

Defines the communication channel from an `InputMethod` back to the application that is receiving its input. You use it to read text around the cursor, commit text to the text box, and send raw key events to the

application. Applications should extend this class rather than implementing the base interface [InputConnection](#).

## [KeyboardView](#)

An extension of [View](#) that renders a keyboard and responds to user input events. The keyboard layout is specified by an instance of [Keyboard](#), which you can define in an XML file.

# Designing the Input Method UI

There are two main visual elements for an IME: the **input** view and the **candidates** view. You only have to implement the elements that are relevant to the input method you're designing.

## Input view

The input view is the UI where the user inputs text, in the form of keyclicks, handwriting or gestures. When the iIME is displayed for the first time, the system calls the [onCreateInputView\(\)](#) callback. In your implementation of this method, you create the layout you want to display in the IME window and return the layout to the system. This snippet is an example of implementing the [onCreateInputView\(\)](#) method:

```
@Override
public View onCreateInputView() {
    MyKeyboardView inputView =
        (MyKeyboardView) getLayoutInflater().inflate( R.layout.input, null )

    inputView.setOnKeyboardActionListener(this); inputView.setKeyboard(mLat)

    return mInputView;
}
```

In this example, `MyKeyboardView` is an instance of a custom implementation of [KeyboardView](#) that renders a [Keyboard](#). If you're building a traditional QWERTY keyboard, see the Soft Keyboard [sample app](#) for an example of how to extend the [KeyboardView](#) class.

## Candidates view

The candidates view is the UI where the IME displays potential word corrections or suggestions for the user to select. In the IME lifecycle, the system calls [onCreateCandidatesView\(\)](#) when it's ready to display the candidate view. In your implementation of this method, return a layout that shows word suggestions, or return null if you don't want to show anything (a null response is the default behavior, so you don't have to implement this if you don't provide suggestions).

For an example implementation that provides user suggestions, see the Soft Keyboard [sample app](#).

## UI design considerations

This section describes some specific UI design considerations for IMEs.

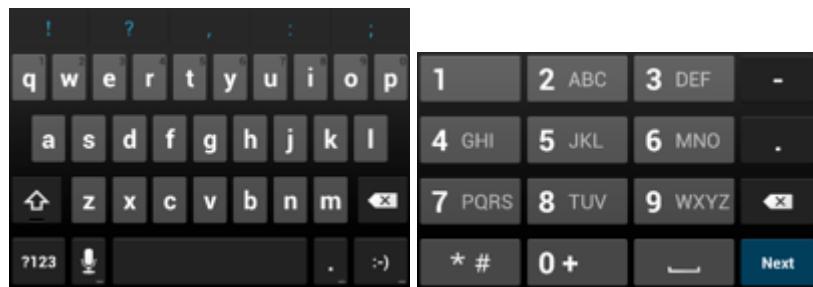
### Handling multiple screen sizes

The UI for your IME must be able to scale for different screen sizes, and it also must handle both landscape and portrait orientations. In non-fullscreen IME mode, leave sufficient space for the application to show the text field and any associated context, so that no more than half the screen is occupied by the IME. In fullscreen IME mode this is not an issue.

## Handling different input types

Android text fields allow you to set a specific input type, such as free form text, numbers, URLs, email addresses, and search strings. When you implement a new IME, you need to detect the input type of each field and provide the appropriate interface for it. However, you don't have to set up your IME to check that the user entered text that's valid for the input type; that's the responsibility of the application that owns the text field.

For example, here are screenshots of the interfaces that the Latin IME provided with the Android platform provides for text and phone number inputs:



**Figure 2.** Latin IME input types.

When an input field receives focus and your IME starts, the system calls `onStartInputView()`, passing in an `EditorInfo` object that contains details about the input type and other attributes of the text field. In this object, the `inputType` field contains the text field's input type.

The `inputType` field is an `int` that contains bit patterns for various input type settings. To test it for the text field's input type, mask it with the constant `TYPE_MASK_CLASS`, like this:

```
inputType & InputType.TYPE_MASK_CLASS
```

The input type bit pattern can have one of several values, including:

### TYPE CLASS NUMBER

A text field for entering numbers. As illustrated in the previous screen shot, the Latin IME displays a number pad for fields of this type.

### TYPE CLASS DATETIME

A text field for entering a date and time.

### TYPE CLASS PHONE

A text field for entering telephone numbers.

### TYPE CLASS TEXT

A text field for entering all supported characters.

These constants are described in more detail in the reference documentation for `InputType`.

The `inputType` field can contain other bits that indicate a variant of the text field type, such as:

### TYPE TEXT VARIATION PASSWORD

A variant of `TYPE_CLASS_TEXT` for entering passwords. The input method will display dingbats instead of the actual text.

### TYPE TEXT VARIATION URI

A variant of `TYPE_CLASS_TEXT` for entering web URLs and other Uniform Resource Identifiers (URIs).

## TYPE\_TEXT\_FLAG\_AUTO\_COMPLETE

A variant of [TYPE\\_CLASS\\_TEXT](#) for entering text that the application "auto-completes" from a dictionary, search, or other facility.

Remember to mask [inputType](#) with the appropriate constant when you test for these variants. The available mask constants are listed in the reference documentation for [InputType](#).

**Caution:** In your own IME, make sure you handle text correctly when you send it to a password field. Hide the password in your UI both in the input view and in the candidates view. Also remember that you shouldn't store passwords on a device. To learn more, see the [Designing for Security](#) guide.

## Sending Text to the Application

As the user inputs text with your IME, you can send text to the application by sending individual key events or by editing the text around the cursor in the application's text field. In either case, you use an instance of [InputConnection](#) to deliver the text. To get this instance, call [InputMethodManager.getCurrentInputConnection\(\)](#).

### Editing the text around the cursor

When you're handling the editing of existing text in a text field, some of the more useful methods in [BaseInputConnection](#) are:

#### [getTextBeforeCursor\(\)](#)

Returns a [CharSequence](#) containing the number of requested characters before the current cursor position.

#### [getTextAfterCursor\(\)](#)

Returns a [CharSequence](#) containing the number of requested characters following the current cursor position.

#### [deleteSurroundingText\(\)](#)

Deletes the specified number of characters before and following the current cursor position.

#### [commitText\(\)](#)

Commit a [CharSequence](#) to the text field and set a new cursor position.

For example, the following snippet shows how to replace the text "Fell" to the left of the with the text "Hello!":

```
InputConnection ic = getCurrentInputConnection();  
  
ic.deleteSurroundingText(4, 0);  
  
ic.commitText("Hello", 1);  
  
ic.commitText("!", 1);
```

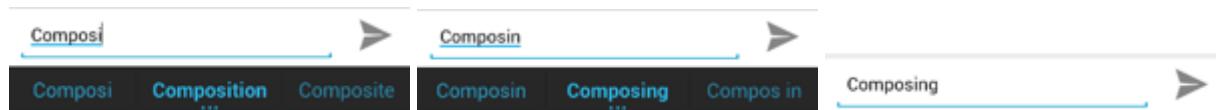
### Composing text before committing

If your IME does text prediction or requires multiple steps to compose a glyph or word, you can show the progress in the text field until the user commits the word, and then you can replace the partial composition with the completed text. You may give special treatment to the text by adding a "span" to it when you pass it to [InputConnection#setComposingText\(\)](#).

The following snippet shows how to show progress in a text field:

```
InputConnection ic = getCurrentInputConnection();  
  
ic.setComposingText("Composi", 1);  
...  
  
ic.setComposingText("Composin", 1);  
  
...  
  
ic.commitText("Composing ", 1);
```

The following screenshots show how this appears to the user:



**Figure 3.** Composing text before committing.

## Intercepting hardware key events

Even though the input method window doesn't have explicit focus, it receives hardware key events first and can choose to consume them or forward them along to the application. For example, you may want to consume the directional keys to navigate within your UI for candidate selection during composition. You may also want to trap the back key to dismiss any popups originating from the input method window.

To intercept hardware keys, override [onKeyDown\(\)](#) and [onKeyUp\(\)](#). See the Soft Keyboard [sample app](#) for an example.

Remember to call the `super()` method for keys you don't want to handle yourself.

## Creating an IME Subtype

Subtypes allow the IME to expose multiple input modes and languages supported by an IME. A subtype can represent:

- A locale such as `en_US` or `fr_FR`
- An input mode such as `voice`, `keyboard`, or `handwriting`
- Other input styles, forms, or properties specific to the IME, such as 10-key or qwerty keyboard layouts.

Basically, the mode can be any text such as "keyboard", "voice", and so forth.

A subtype can also expose a combination of these.

Subtype information is used for an IME switcher dialog that's available from the notification bar and also for IME settings. The information also allows the framework to bring up a specific subtype of an IME directly. When you build an IME, use the subtype facility, because it helps the user identify and switch between different IME languages and modes.

You define subtypes in one of the input method's XML resource files, using the `<subtype>` element. The following snippet defines an IME with two subtypes: a keyboard subtype for the US English locale, and another keyboard subtype for the French language locale for France:

```

<input-method xmlns:android="http://schemas.android.com/apk/res/android"
    android:settingsActivity="com.example.softkeyboard.Settings"
    android:icon="@drawable/ime_icon"
    <subtype android:name="@string/display_name_english_keyboard_ime"
        android:icon="@drawable/subtype_icon_english_keyboard_ime"
        android:imeSubtypeLanguage="en_US"
        android:imeSubtypeMode="keyboard"
        android:imeSubtypeExtraValue="somePrivateOption=true"
    />
    <subtype android:name="@string/display_name_french_keyboard_ime"
        android:icon="@drawable/subtype_icon_french_keyboard_ime"
        android:imeSubtypeLanguage="fr_FR"
        android:imeSubtypeMode="keyboard"
        android:imeSubtypeExtraValue="foobar=30, someInternalOption=false"
    />
    <subtype android:name="@string/display_name_german_keyboard_ime"
        ...
    />
/>

```

To ensure that your subtypes are labeled correctly in the UI, use %s to get a subtype label that is the same as the subtype's locale label. This is demonstrated in the next two snippets. The first snippet shows part of the input method's XML file:

```

<subtype
    android:label="@string/label_subtype_generic"
    android:imeSubtypeLocale="en_US"
    android:icon="@drawable/icon_en_us"
    android:imeSubtypeMode="keyboard" />

```

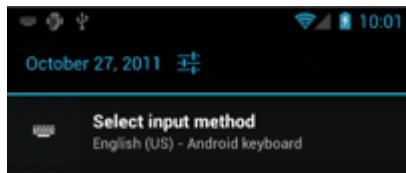
The next snippet is part of the IME's strings.xml file. The string resource label\_subtype\_generic, which is used by the input method UI definition to set the subtype's label, is defined as:

```
<string name="label_subtype_generic">%s</string>
```

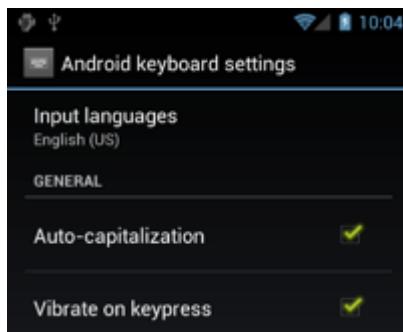
This sets the subtype's display name to "English (United States)" in any English language locale, or to the appropriate localization in other locales.

## Choosing IME subtypes from the notification bar

The Android system manages all subtypes exposed by all IMEs. IME subtypes are treated as modes of the IME they belong to. In the notification bar, a user can select an available subtype for the currently-set IME, as shown in the following screenshot:



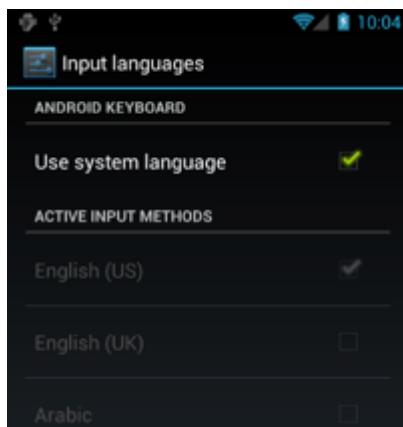
**Figure 4.** Choosing an IME subtype from the notification bar.



**Figure 5.** Setting subtype preferences in System Settings.

## Choosing IME subtypes from System Settings

A user can control how subtypes are used in the “Language & input” settings panel in the System Settings area. In the Soft Keyboard sample, the file `InputMethodSettingsFragment.java` contains an implementation that facilitates a subtype enabler in the IME settings. Please refer to the SoftKeyboard sample in the Android SDK for more information about how to support Input Method Subtypes in your IME.



**Figure 6.** Choosing a language for the IME.

## General IME Considerations

Here are some other things to consider as you're implementing your IME:

- Provide a way for users to set options directly from the IME's UI.
- Because multiple IMEs may be installed on the device, provide a way for the user to switch to a different IME directly from the input method UI.
- Bring up the IME's UI quickly. Preload or load on demand any large resources so that users see the IME as soon as they tap on a text field. Cache resources and views for subsequent invocations of the input method.
- Conversely, you should release large memory allocations soon after the input method window is hidden, so that applications can have sufficient memory to run. Consider using a delayed message to release resources if the IME is in a hidden state for a few seconds.
- Make sure that users can enter as many characters as possible for the language or locale associated with the IME. Remember that users may use punctuation in passwords or user names, so your IME has to provide many different characters to allow users to enter a password and get access to the device.

# Spelling Checker Framework

## In This Document

1. [Spell Check Lifecycle](#)
2. [Implementing a Spell Checker Service](#)
3. [Implementing a Spell Checker Client](#)

## See also

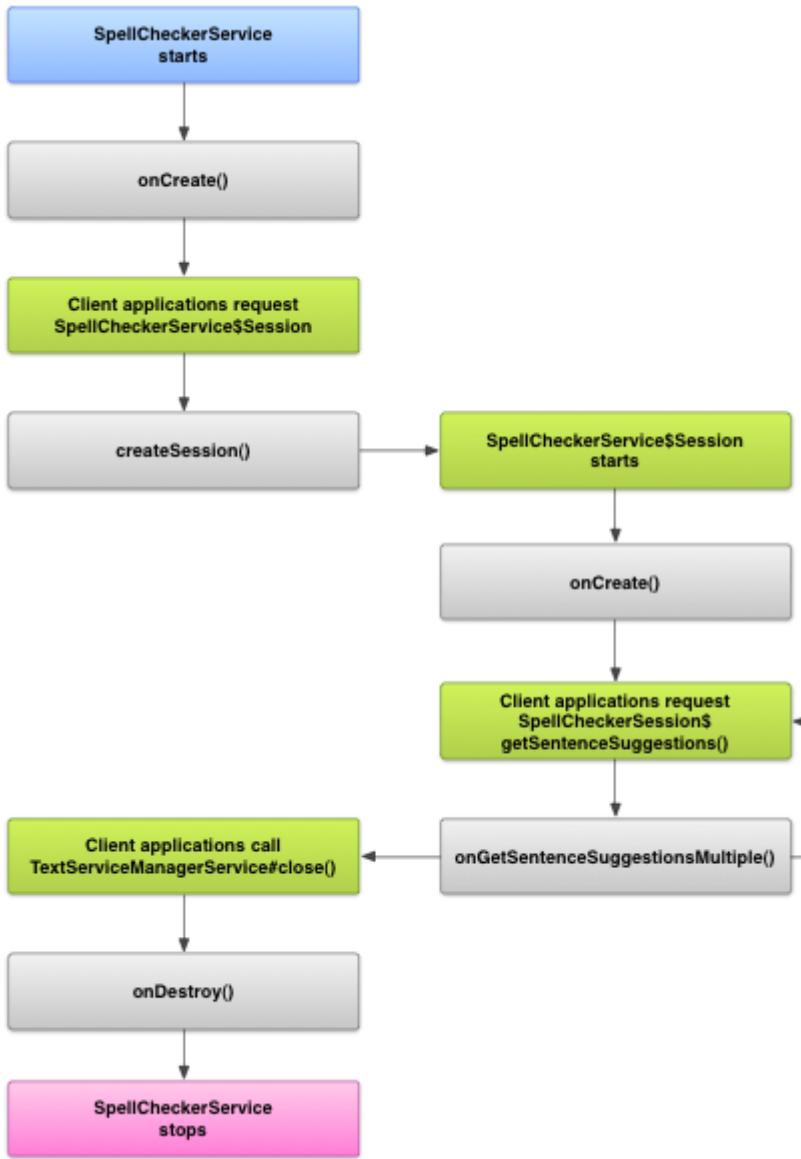
1. [Spell Checker Service](#) sample app
2. [Spell Checker Client](#) sample app

The Android platform offers a spelling checker framework that lets you implement and access spell checking in your application. The framework is one of the Text Service APIs offered by the Android platform.

To use the framework in your app, you create a special type of Android service that generates a spelling checker **session** object. Based on text you provide, the session object returns spelling suggestions generated by the spelling checker.

## Spell Checker Lifecycle

The following diagram shows the lifecycle of the spelling checker service:



**Figure 1.** The spelling checker service lifecycle.

To initiate spell checking, your app starts its implementation of the spelling checker service. Clients in your app, such as activities or individual UI elements, request a spelling checker session from the service, then use the session to get suggestions for text. As a client terminates its operation, it closes its spelling checker session. If necessary, your app can shut down the spelling checker service at any time.

## Implementing a Spell Checker Service

To use the spelling checker framework in your app, add a spelling checker service component including the session object definition. You can also add to your app an optional activity that controls settings. You must also add an XML metadata file that describes the spelling checker service, and add the appropriate elements to your manifest file.

### Spell checker classes

Define the service and session object with the following classes:

#### A subclass of [SpellCheckerService](#)

The [SpellCheckerService](#) implements both the [Service](#) class and the spelling checker framework interface. Within your subclass, you must implement the following method:

## [createSession\(\)](#)

A factory method that returns a [SpellCheckerService.Session](#) object to a client that wants to do spell checking.

See the [Spell Checker Service](#) sample app to learn more about implementing this class.

## An implementation of [SpellCheckerService.Session](#)

An object that the spelling checker service provides to clients, to let them pass text to the spelling checker and receive suggestions. Within this class, you must implement the following methods:

### [onCreate\(\)](#)

Called by the system in response to [createSession\(\)](#). In this method, you can initialize the [SpellCheckerService.Session](#) object based on the current locale and so forth.

### [onGetSentenceSuggestionsMultiple\(\)](#)

Does the actual spell checking. This method returns an array of [SentenceSuggestionsInfo](#) containing suggestions for the sentences passed to it.

Optionally, you can implement [onCancel\(\)](#), which handles requests to cancel spell checking, [onGetSuggestions\(\)](#), which handles a word suggestion request, or [onGetSuggestionsMultiple\(\)](#), which handles batches of word suggestion requests.

See the [Spell Checker Client](#) sample app to learn more about implementing this class.

**Note:** You must implement all aspects of spell checking as asynchronous and thread-safe. A spelling checker may be called simultaneously by different threads running on different cores. The [SpellCheckerService](#) and [SpellCheckerService.Session](#) take care of this automatically.

## Spell checker manifest and metadata

In addition to code, you need to provide the appropriate manifest file and a metadata file for the spelling checker.

The manifest file defines the application, the service, and the activity for controlling settings, as shown in the following snippet:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.samplespellcheckerservice" >
    <application
        android:label="@string/app_name" >
        <service
            android:label="@string/app_name"
            android:name=".SampleSpellCheckerService"
            android:permission="android.permission.BIND_TEXT_SERVICE" >
            <intent-filter >
                <action android:name="android.service.textservice.SpellCheckerS
            </intent-filter>

            <meta-data
                android:name="android.view.textservice.scs"
                android:resource="@xml/spellchecker" />
        </service>

        <activity
            android:label="@string/sample_settings"
```

```

        android:name="SpellCheckerSettingsActivity" >
    <intent-filter >
        <action android:name="android.intent.action.MAIN" />
    </intent-filter>
    </activity>
</application>
</manifest>

```

Notice that components that want to use the service must request the permission [BIND\\_TEXT\\_SERVICE](#) to ensure that only the system binds to the service. The service's definition also specifies the `spellchecker.xml` metadata file, which is described in the next section.

The metadata file `spellchecker.xml` contains the following XML:

```

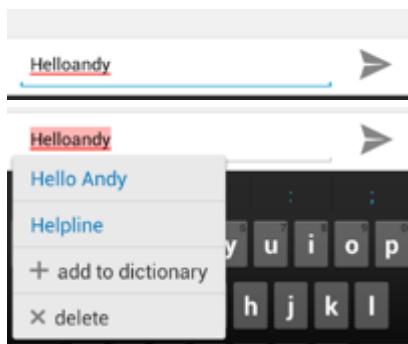
<spell-checker xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/spellchecker_name"
    android:settingsActivity="com.example.SpellCheckerSettingsActivity">
    <subtype
        android:label="@string/subtype_generic"
        android:subTypeLocale="en"
    />
    <subtype
        android:label="@string/subtype_generic"
        android:subTypeLocale="fr"
    />
</spell-checker>

```

The metadata specifies the activity that the spelling checker uses for controlling settings. It also defines subtypes for the spelling checker; in this case, the subtypes define locales that the spelling checker can handle.

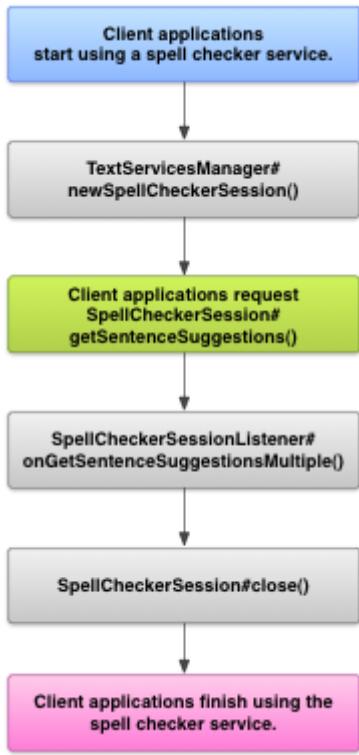
## Accessing the Spell Checker Service from a Client

Applications that use [TextView](#) views automatically benefit from spell checking, because [TextView](#) automatically uses a spelling checker. The following screenshots show this:



**Figure 2.** Spell checking in TextView.

However, you may want to interact directly with a spelling checker service in other cases as well. The following diagram shows the flow of control for interacting with a spelling checker service:



**Figure 3.** Interacting with a spelling checker service.

The [Spell Checker Client](#) sample app shows how to interact with a spelling checker service. The LatinIME input method editor in the Android Open Source Project also contains an example of spell checking.

# Data Storage

Store application data in databases, files, or preferences, in internal or removeable storage. You can also add a data backup service to let users store and recover application and system data.

## Training

### [Syncing to the Cloud](#)

This class covers different strategies for cloud enabled applications. It covers syncing data with the cloud using your own back-end web application, and backing up data using the cloud so that users can restore their data when installing your application on a new device.

# Storage Options

## Storage quickview

- Use Shared Preferences for primitive data
- Use internal device storage for private data
- Use external storage for large data sets that are not private
- Use SQLite databases for structured storage

## In this document

1. [Using Shared Preferences](#)
2. [Using the Internal Storage](#)
3. [Using the External Storage](#)
4. [Using Databases](#)
5. [Using a Network Connection](#)

## See also

1. [Content Providers and Content Resolvers](#)

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:

### [Shared Preferences](#)

Store private primitive data in key-value pairs.

### [Internal Storage](#)

Store private data on the device memory.

### [External Storage](#)

Store public data on the shared external storage.

### [SQLite Databases](#)

Store structured data in a private database.

### [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. For more information about using content providers, see the [Content Providers](#) documentation.

## Using 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).

## User Preferences

Shared preferences are not strictly for saving "user preferences," such as what ringtone a user has chosen. If you're interested in creating user preferences for your application, see [PreferenceActivity](#), which provides an Activity framework for you to create user preferences, which will be automatically persisted (using shared preferences).

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:

1. Call [edit\(\)](#) to get a [SharedPreferences.Editor](#).
2. Add values with methods such as [putBoolean\(\)](#) and [putString\(\)](#).
3. Commit the new values with [commit\(\)](#)

To read values, use [SharedPreferences](#) methods such as [getBoolean\(\)](#) and [getString\(\)](#).

Here is an example that saves a preference for silent keypress mode in a calculator:

```
public class Calc extends Activity {  
    public static final String PREFS_NAME = "MyPrefsFile";  
  
    @Override  
    protected void onCreate(Bundle state) {  
        super.onCreate(state);  
        . . .  
  
        // Restore preferences  
        SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);  
        boolean silent = settings.getBoolean("silentMode", false);  
        setSilent(silent);  
    }  
  
    @Override  
    protected void onStop() {  
        super.onStop();  
  
        // We need an Editor object to make preference changes.  
        // All objects are from android.context.Context  
        SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);  
        SharedPreferences.Editor editor = settings.edit();  
        editor.putBoolean("silentMode", mSilentMode);  
  
        // Commit the edits!  
        editor.commit();  
    }  
}
```

```
    }  
}
```

## Using the 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:

1. Call [openFileOutput\(\)](#) with the name of the file and the operating mode. This returns a [FileOutputStream](#).
2. Write to the file with [write\(\)](#).
3. Close the stream with [close\(\)](#).

For example:

```
String FILENAME = "hello_file";  
String string = "hello world!";  
  
FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);  
fos.write(string.getBytes());  
fos.close();
```

[MODE\\_PRIVATE](#) will create the file (or replace a file of the same name) and make it private to your application. Other modes available are: [MODE\\_APPEND](#), [MODE\\_WORLD\\_READABLE](#), and [MODE\\_WORLD\\_WRITEABLE](#).

To read a file from internal storage:

1. Call [openFileInput\(\)](#) and pass it the name of the file to read. This returns a [InputStream](#).
2. Read bytes from the file with [read\(\)](#).
3. Then close the stream with [close\(\)](#).

**Tip:** If you want to save a static file in your application at compile time, save the file in your project `res/raw/` directory. You can open it with [openRawResource\(\)](#), passing the `R.raw.<filename>` resource ID. This method returns an [InputStream](#) that you can use to read the file (but you cannot write to the original file).

## Saving cache files

If you'd like to cache some data, rather than store it persistently, you should use [getCacheDir\(\)](#) to open a [File](#) that represents the internal directory where your application should save temporary cache files.

When the device is low on internal storage space, Android may delete these cache files to recover space. However, you should not rely on the system to clean up these files for you. You should always maintain the cache files yourself and stay within a reasonable limit of space consumed, such as 1MB. When the user uninstalls your application, these files are removed.

## Other useful methods

### [getFilesDir\(\)](#)

Gets the absolute path to the filesystem directory where your internal files are saved.

### [getDir\(\)](#)

Creates (or opens an existing) directory within your internal storage space.

### [deleteFile\(\)](#)

Deletes a file saved on the internal storage.

### [fileList\(\)](#)

Returns an array of files currently saved by your application.

## Using the 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.

It's possible that a device using a partition of the internal storage for the external storage may also offer an SD card slot. In this case, the SD card is *not* part of the external storage and your app cannot access it (the extra storage is intended only for user-provided media that the system scans).

**Caution:** External storage can become unavailable if the user mounts the external storage on a computer or removes the media, and there's no security enforced upon files you save to the external storage. All applications can read and write files placed on the external storage and the user can remove them.

### Checking media availability

Before you do any work with the external storage, you should always call [getExternalStorageState\(\)](#) to check whether the media is available. The media might be mounted to a computer, missing, read-only, or in some other state. For example, here's how you can check the availability:

```
boolean mExternalStorageAvailable = false;
boolean mExternalStorageWriteable = false;
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // We can read and write the media
    mExternalStorageAvailable = mExternalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    // We can only read the media
    mExternalStorageAvailable = true;
    mExternalStorageWriteable = false;
} else {
    // Something else is wrong. It may be one of many other states, but all we
    // to know is we can neither read nor write
    mExternalStorageAvailable = mExternalStorageWriteable = false;
}
```

This example checks whether the external storage is available to read and write. The [getExternalStorageState\(\)](#) method returns other states that you might want to check, such as whether the media is being

shared (connected to a computer), is missing entirely, has been removed badly, etc. You can use these to notify the user with more information when your application needs to access the media.

## Accessing files on external storage

If you're using API Level 8 or greater, use [getExternalFilesDir\(\)](#) to open a [File](#) that represents the external storage directory where you should save your files. This method takes a `type` parameter that specifies the type of subdirectory you want, such as [DIRECTORY\\_MUSIC](#) and [DIRECTORY\\_RINGTONES](#) (pass `null` to receive the root of your application's file directory). This method will create the appropriate directory if necessary. By specifying the type of directory, you ensure that the Android's media scanner will properly categorize your files in the system (for example, ringtones are identified as ringtones and not music). If the user uninstalls your application, this directory and all its contents will be deleted.

If you're using API Level 7 or lower, use [getExternalStorageDirectory\(\)](#), to open a [File](#) representing the root of the external storage. You should then write your data in the following directory:

```
/Android/data/<package_name>/files/
```

The `<package_name>` is your Java-style package name, such as "com.example.android.app". If the user's device is running API Level 8 or greater and they uninstall your application, this directory and all its contents will be deleted.

## Hiding your files from the Media Scanner

Include an empty file named `.nomedia` in your external files directory (note the dot prefix in the filename). This will prevent Android's media scanner from reading your media files and including them in apps like Gallery or Music.

## Saving files that should be shared

If you want to save files that are not specific to your application and that should *not* be deleted when your application is uninstalled, save them to one of the public directories on the external storage. These directories lay at the root of the external storage, such as `Music/`, `Pictures/`, `Ringtones/`, and others.

In API Level 8 or greater, use [getExternalStoragePublicDirectory\(\)](#), passing it the type of public directory you want, such as [DIRECTORY\\_MUSIC](#), [DIRECTORY\\_PICTURE](#), [DIRECTORY\\_RINGTONE](#), or others. This method will create the appropriate directory if necessary.

If you're using API Level 7 or lower, use [getExternalStorageDirectory\(\)](#) to open a [File](#) that represents the root of the external storage, then save your shared files in one of the following directories:

- `Music/` - Media scanner classifies all media found here as user music.
- `Podcasts/` - Media scanner classifies all media found here as a podcast.
- `Ringtones/` - Media scanner classifies all media found here as a ringtone.
- `Alarms/` - Media scanner classifies all media found here as an alarm sound.
- `Notifications/` - Media scanner classifies all media found here as a notification sound.
- `Pictures/` - All photos (excluding those taken with the camera).
- `Movies/` - All movies (excluding those taken with the camcorder).
- `Download/` - Miscellaneous downloads.

## Saving cache files

If you're using API Level 8 or greater, use [getExternalCacheDir\(\)](#) to open a [File](#) that represents the external storage directory where you should save cache files. If the user uninstalls your application, these files will be automatically deleted. However, during the life of your application, you should manage these cache files and remove those that aren't needed in order to preserve file space.

If you're using API Level 7 or lower, use [getExternalStorageDirectory\(\)](#) to open a [File](#) that represents the root of the external storage, then write your cache data in the following directory:

```
/Android/data/<package_name>/cache/
```

The `<package_name>` is your Java-style package name, such as "com.example.android.app".

## Using 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:

```
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.

Android does not impose any limitations beyond the standard SQLite concepts. We do recommend including an autoincrement value key field that can be used as a unique ID to quickly find a record. This is not required for private data, but if you implement a [content provider](#), you must include a unique ID using the [BaseColumns.\\_ID](#) constant.

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.

For sample apps that demonstrate how to use SQLite databases in Android, see the [Note Pad](#) and [Searchable Dictionary](#) applications.

## 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. See [Examining sqlite3 databases from a remote shell](#) to learn how to run this tool.

# Using a Network Connection

You can use the network (when it's available) to store and retrieve data on your own web-based services. To do network operations, use classes in the following packages:

- [java.net.\\*](#)
- [android.net.\\*](#)

# Data Backup

## Quickview

- Back up the user's data to the cloud in case the user loses it
- If the user upgrades to a new Android-powered device, your app can restore the user's data onto the new device
- Easily back up SharedPreferences and private files with BackupAgentHelper
- Requires API Level 8

## In this document

1. [The Basics](#)
2. [Declaring the Backup Agent in Your Manifest](#)
3. [Registering for Android Backup Service](#)
4. [Extending BackupAgent](#)
  1. [Required Methods](#)
  2. [Performing backup](#)
  3. [Performing restore](#)
5. [Extending BackupAgentHelper](#)
  1. [Backing up SharedPreferences](#)
  2. [Backing up Private Files](#)
6. [Checking the Restore Data Version](#)
7. [Requesting Backup](#)
8. [Requesting Restore](#)
9. [Testing Your Backup Agent](#)

## Key classes

1. [BackupManager](#)
2. [BackupAgent](#)
3. [BackupAgentHelper](#)

## See also

1. [bmgr tool](#)

Android's [backup](#) service allows you to copy your persistent application data to remote "cloud" storage, in order to provide a restore point for the application data and settings. If a user performs a factory reset or converts to a new Android-powered device, the system automatically restores your backup data when the application is re-installed. This way, your users don't need to reproduce their previous data or application settings. This process is completely transparent to the user and does not affect the functionality or user experience in your application.

During a backup operation (which your application can request), Android's Backup Manager ([BackupManager](#)) queries your application for backup data, then hands it to a backup transport, which then delivers the data to the cloud storage. During a restore operation, the Backup Manager retrieves the backup data from the backup transport and returns it to your application so your application can restore the data to the device. It's possible for your application to request a restore, but that shouldn't be necessary—Android automatically performs a restore operation when your application is installed and there exists backup data associated with the

user. The primary scenario in which backup data is restored is when a user resets their device or upgrades to a new device and their previously installed applications are re-installed.

**Note:** The backup service is *not* designed for synchronizing application data with other clients or saving data that you'd like to access during the normal application lifecycle. You cannot read or write backup data on demand and cannot access it in any way other than through the APIs provided by the Backup Manager.

The backup transport is the client-side component of Android's backup framework, which is customizable by the device manufacturer and service provider. The backup transport may differ from device to device and which backup transport is available on any given device is transparent to your application. The Backup Manager APIs isolate your application from the actual backup transport available on a given device—your application communicates with the Backup Manager through a fixed set of APIs, regardless of the underlying transport.

Data backup is *not* guaranteed to be available on all Android-powered devices. However, your application is not adversely affected in the event that a device does not provide a backup transport. If you believe that users will benefit from data backup in your application, then you can implement it as described in this document, test it, then publish your application without any concern about which devices actually perform backup. When your application runs on a device that does not provide a backup transport, your application operates normally, but will not receive callbacks from the Backup Manager to backup data.

Although you cannot know what the current transport is, you are always assured that your backup data cannot be read by other applications on the device. Only the Backup Manager and backup transport have access to the data you provide during a backup operation.

**Caution:** Because the cloud storage and transport service can differ from device to device, Android makes no guarantees about the security of your data while using backup. You should always be cautious about using backup to store sensitive data, such as usernames and passwords.

## The Basics

To backup your application data, you need to implement a backup agent. Your backup agent is called by the Backup Manager to provide the data you want to back up. It is also called to restore your backup data when the application is re-installed. The Backup Manager handles all your data transactions with the cloud storage (using the backup transport) and your backup agent handles all your data transactions on the device.

To implement a backup agent, you must:

1. Declare your backup agent in your manifest file with the [android:backupAgent](#) attribute.
2. Register your application with a backup service. Google offers [Android Backup Service](#) as a backup service for most Android-powered devices, which requires that you register your application in order for it to work. Any other backup services available might also require you to register in order to store your data on their servers.
3. Define a backup agent by either:
  - a. [Extending BackupAgent](#)

The [BackupAgent](#) class provides the central interface with which your application communicates with the Backup Manager. If you extend this class directly, you must override [onBackup\(\)](#) and [onRestore\(\)](#) to handle the backup and restore operations for your data.

*Or*

- b. [Extending BackupAgentHelper](#)

The [BackupAgentHelper](#) class provides a convenient wrapper around the [BackupAgent](#) class, which minimizes the amount of code you need to write. In your [BackupAgentHelper](#), you must use one or more "helper" objects, which automatically backup and restore certain types of data, so that you do not need to implement [onBackup\(\)](#) and [onRestore\(\)](#).

Android currently provides backup helpers that will backup and restore complete files from [SharedPreferences](#) and [internal storage](#).

## Declaring the Backup Agent in Your Manifest

This is the easiest step, so once you've decided on the class name for your backup agent, declare it in your manifest with the [android:backupAgent](#) attribute in the [application](#) tag.

For example:

```
<manifest ... >
  ...
  <application android:label="MyApplication"
    android:backupAgent="MyBackupAgent">
    <activity ... >
      ...
    </activity>
  </application>
</manifest>
```

Another attribute you might want to use is [android:restoreAnyVersion](#). This attribute takes a boolean value to indicate whether you want to restore the application data regardless of the current application version compared to the version that produced the backup data. (The default value is "false".) See [Checking the Restore Data Version](#) for more information.

**Note:** The backup service and the APIs you must use are available only on devices running API Level 8 (Android 2.2) or greater, so you should also set your [android:minSdkVersion](#) attribute to "8".

## Registering for Android Backup Service

Google provides a backup transport with [Android Backup Service](#) for most Android-powered devices running Android 2.2 or greater.

In order for your application to perform backup using Android Backup Service, you must register your application with the service to receive a Backup Service Key, then declare the Backup Service Key in your Android manifest.

To get your Backup Service Key, [register for Android Backup Service](#). When you register, you will be provided a Backup Service Key and the appropriate <meta-data> XML code for your Android manifest file, which you must include as a child of the <application> element. For example:

```
<application android:label="MyApplication"
  android:backupAgent="MyBackupAgent">
  ...
  <meta-data android:name="com.google.android.backup.api_key"
    android:value="AEdPqrEAAAIDayEVgU6DJnyJdBmU7KLH3kszDXLw_4DIsEIyQ" />
</application>
```

The `android:name` must be "com.google.android.backup.api\_key" and the `android:value` must be the Backup Service Key received from the Android Backup Service registration.

If you have multiple applications, you must register each one, using the respective package name.

**Note:** The backup transport provided by Android Backup Service is not guaranteed to be available on all Android-powered devices that support backup. Some devices might support backup using a different transport, some devices might not support backup at all, and there is no way for your application to know what transport is used on the device. However, if you implement backup for your application, you should always include a Backup Service Key for Android Backup Service so your application can perform backup when the device uses the Android Backup Service transport. If the device does not use Android Backup Service, then the `<meta-data>` element with the Backup Service Key is ignored.

## Extending BackupAgent

Most applications shouldn't need to extend the [BackupAgent](#) class directly, but should instead [extend BackupAgentHelper](#) to take advantage of the built-in helper classes that automatically backup and restore your files. However, you might want to extend [BackupAgent](#) directly if you need to:

- Version your data format. For instance, if you anticipate the need to revise the format in which you write your application data, you can build a backup agent to cross-check your application version during a restore operation and perform any necessary compatibility work if the version on the device is different than that of the backup data. For more information, see [Checking the Restore Data Version](#).
- Instead of backing up an entire file, you can specify the portions of data that should be backed up and how each portion is then restored to the device. (This can also help you manage different versions, because you read and write your data as unique entities, rather than complete files.)
- Back up data in a database. If you have an SQLite database that you want to restore when the user re-installs your application, you need to build a custom [BackupAgent](#) that reads the appropriate data during a backup operation, then create your table and insert the data during a restore operation.

If you don't need to perform any of the tasks above and want to back up complete files from [SharedPreferences](#) or [internal storage](#), you should skip to [Extending BackupAgentHelper](#).

## Required Methods

When you create a backup agent by extending [BackupAgent](#), you must implement the following callback methods:

### [onBackup\(\)](#)

The Backup Manager calls this method after you [request a backup](#). In this method, you read your application data from the device and pass the data you want to back up to the Backup Manager, as described below in [Performing backup](#).

### [onRestore\(\)](#)

The Backup Manager calls this method during a restore operation (you can [request a restore](#), but the system automatically performs restore when the user re-installs your application). When it calls this method, the Backup Manager delivers your backup data, which you then restore to the device, as described below in [Performing restore](#).

## Performing backup

When it's time to back up your application data, the Backup Manager calls your [onBackup\(\)](#) method. This is where you must provide your application data to the Backup Manager so it can be saved to cloud storage.

Only the Backup Manager can call your backup agent's [onBackup \(\)](#) method. Each time that your application data changes and you want to perform a backup, you must request a backup operation by calling [dataChanged \(\)](#) (see [Requesting Backup](#) for more information). A backup request does not result in an immediate call to your [onBackup \(\)](#) method. Instead, the Backup Manager waits for an appropriate time, then performs backup for all applications that have requested a backup since the last backup was performed.

**Tip:** While developing your application, you can initiate an immediate backup operation from the Backup Manager with the [bmgr tool](#).

When the Backup Manager calls your [onBackup \(\)](#) method, it passes three parameters:

#### **oldState**

An open, read-only [ParcelFileDescriptor](#) pointing to the last backup state provided by your application. This is not the backup data from cloud storage, but a local representation of the data that was backed up the last time [onBackup \(\)](#) was called (as defined by [newState](#), below, or from [onRestore \(\)](#)—more about this in the next section). Because [onBackup \(\)](#) does not allow you to read existing backup data in the cloud storage, you can use this local representation to determine whether your data has changed since the last backup.

#### **data**

A [BackupDataOutput](#) object, which you use to deliver your backup data to the Backup Manager.

#### **newState**

An open, read/write [ParcelFileDescriptor](#) pointing to a file in which you must write a representation of the data that you delivered to [data](#) (a representation can be as simple as the last-modified timestamp for your file). This object is returned as [oldState](#) the next time the Backup Manager calls your [onBackup \(\)](#) method. If you do not write your backup data to [newState](#), then [oldState](#) will point to an empty file next time Backup Manager calls [onBackup \(\)](#).

Using these parameters, you should implement your [onBackup \(\)](#) method to do the following:

1. Check whether your data has changed since the last backup by comparing [oldState](#) to your current data. How you read data in [oldState](#) depends on how you originally wrote it to [newState](#) (see step 3). The easiest way to record the state of a file is with its last-modified timestamp. For example, here's how you can read and compare a timestamp from [oldState](#):

```
// Get the oldState input stream
FileInputStream instream = new FileInputStream(oldState.getFileDescriptor());
DataInputStream in = new DataInputStream(instream);

try {
    // Get the last modified timestamp from the state file and data file
    long stateModified = in.readLong();
    long fileModified = mDataFile.lastModified();

    if (stateModified != fileModified) {
        // The file has been modified, so do a backup
        // Or the time on the device changed, so be safe and do a backup
    } else {
        // Don't back up because the file hasn't changed
        return;
    }
} catch (IOException e) {
```

```
// Unable to read state file... be safe and do a backup
}
```

If nothing has changed and you don't need to back up, skip to step 3.

2. If your data has changed, compared to `oldState`, write the current data to `data` to back it up to the cloud storage.

You must write each chunk of data as an "entity" in the [BackupDataOutput](#). An entity is a flattened binary data record that is identified by a unique key string. Thus, the data set that you back up is conceptually a set of key-value pairs.

To add an entity to your backup data set, you must:

1. Call [writeEntityHeader\(\)](#), passing a unique string key for the data you're about to write and the data size.
2. Call [writeEntityData\(\)](#), passing a byte buffer that contains your data and the number of bytes to write from the buffer (which should match the size passed to [writeEntityHeader\(\)](#)).

For example, the following code flattens some data into a byte stream and writes it into a single entity:

```
// Create buffer stream and data output stream for our data
ByteArrayOutputStream bufStream = new ByteArrayOutputStream();
DataOutputStream outWriter = new DataOutputStream(bufStream);
// Write structured data
outWriter.writeUTF(mPlayerName);
outWriter.writeInt(mPlayerScore);
// Send the data to the Backup Manager via the BackupDataOutput
byte[] buffer = bufStream.toByteArray();
int len = buffer.length;
data.writeEntityHeader(TOPSCORE_BACKUP_KEY, len);
data.writeEntityData(buffer, len);
```

Perform this for each piece of data that you want to back up. How you divide your data into entities is up to you (and you might use just one entity).

3. Whether or not you perform a backup (in step 2), write a representation of the current data to the `newState` [ParcelFileDescriptor](#). The Backup Manager retains this object locally as a representation of the data that is currently backed up. It passes this back to you as `oldState` the next time it calls [onBackup\(\)](#) so you can determine whether another backup is necessary (as handled in step 1). If you do not write the current data state to this file, then `oldState` will be empty during the next callback.

The following example saves a representation of the current data into `newState` using the file's last-modified timestamp:

```
FileOutputStream outstream = new FileOutputStream(newState.getFileDescriptor());
DataOutputStream out = new DataOutputStream(outstream);

long modified = mDataFile.lastModified();
out.writeLong(modified);
```

**Caution:** If your application data is saved to a file, make sure that you use synchronized statements while accessing the file so that your backup agent does not read the file while an Activity in your application is also writing the file.

## Performing restore

When it's time to restore your application data, the Backup Manager calls your backup agent's [onRestore\(\)](#) method. When it calls this method, the Backup Manager delivers your backup data so you can restore it onto the device.

Only the Backup Manager can call [onRestore\(\)](#), which happens automatically when the system installs your application and finds existing backup data. However, you can request a restore operation for your application by calling [requestRestore\(\)](#) (see [Requesting restore](#) for more information).

**Note:** While developing your application, you can also request a restore operation with the [bmgr tool](#).

When the Backup Manager calls your [onRestore\(\)](#) method, it passes three parameters:

### **data**

A [BackupDataInput](#), which allows you to read your backup data.

### **appVersionCode**

An integer representing the value of your application's [android:versionCode](#) manifest attribute, as it was when this data was backed up. You can use this to cross-check the current application version and determine if the data format is compatible. For more information about using this to handle different versions of restore data, see the section below about [Checking the Restore Data Version](#).

### **newState**

An open, read/write [ParcelFileDescriptor](#) pointing to a file in which you must write the final backup state that was provided with data. This object is returned as `oldState` the next time [onBackup\(\)](#) is called. Recall that you must also write the same `newState` object in the [onBackup\(\)](#) callback—also doing it here ensures that the `oldState` object given to [onBackup\(\)](#) is valid even the first time [onBackup\(\)](#) is called after the device is restored.

In your implementation of [onRestore\(\)](#), you should call [readNextHeader\(\)](#) on the `data` to iterate through all entities in the data set. For each entity found, do the following:

1. Get the entity key with [getKey\(\)](#).
2. Compare the entity key to a list of known key values that you should have declared as static final strings inside your [BackupAgent](#) class. When the key matches one of your known key strings, enter into a statement to extract the entity data and save it to the device:
  1. Get the entity data size with [getEntitySize\(\)](#) and create a byte array of that size.
  2. Call [readEntityData\(\)](#) and pass it the byte array, which is where the data will go, and specify the start offset and the size to read.
  3. Your byte array is now full and you can read the data and write it to the device however you like.
3. After you read and write your data back to the device, write the state of your data to the `newState` parameter the same as you do during [onBackup\(\)](#).

For example, here's how you can restore the data backed up by the example in the previous section:

```
@Override  
public void onRestore(BackupDataInput data, int appVersionCode,
```

```

        ParcelFileDescriptor newState) throws IOException {
    // There should be only one entity, but the safest
    // way to consume it is using a while loop
    while (data.readNextHeader()) {
        String key = data.getKey();
        int dataSize = data.getDataSize();

        // If the key is ours (for saving top score). Note this key was used wh...
        // we wrote the backup entity header
        if (TOPSCORE_BACKUP_KEY.equals(key)) {
            // Create an input stream for the BackupDataInput
            byte[] dataBuf = new byte[dataSize];
            data.readEntityData(dataBuf, 0, dataSize);
            ByteArrayInputStream baStream = new ByteArrayInputStream(dataBuf);
            DataInputStream in = new DataInputStream(baStream);

            // Read the player name and score from the backup data
            mPlayerName = in.readUTF();
            mPlayerScore = in.readInt();

            // Record the score on the device (to a file or something)
            recordScore(mPlayerName, mPlayerScore);
        } else {
            // We don't know this entity key. Skip it. (Shouldn't happen.)
            data.skipEntityData();
        }
    }

    // Finally, write to the state blob (newState) that describes the restored
    FileOutputStream outstream = new FileOutputStream(newState.getFileDescriptor());
    DataOutputStream out = new DataOutputStream(outstream);
    out.writeUTF(mPlayerName);
    out.writeInt(mPlayerScore);
}
}

```

In this example, the `appVersionCode` parameter passed to [onRestore\(\)](#) is not used. However, you might want to use it if you've chosen to perform backup when the user's version of the application has actually moved backward (for example, the user went from version 1.5 of your app to 1.0). For more information, see the section about [Checking the Restore Data Version](#).

For an example implementation of [BackupAgent](#), see the [ExampleAgent](#) class in the [Backup and Restore](#) sample application.

## Extending BackupAgentHelper

You should build your backup agent using [BackupAgentHelper](#) if you want to back up complete files (from either [SharedPreferences](#) or [internal storage](#)). Building your backup agent with [BackupAgentHelper](#) requires far less code than extending [BackupAgent](#), because you don't have to implement [onBackup\(\)](#) and [onRestore\(\)](#).

Your implementation of [BackupAgentHelper](#) must use one or more backup helpers. A backup helper is a specialized component that [BackupAgentHelper](#) summons to perform backup and restore operations for a particular type of data. The Android framework currently provides two different helpers:

- [SharedPreferencesBackupHelper](#) to backup [SharedPreferences](#) files.
- [FileBackupHelper](#) to backup files from [internal storage](#).

You can include multiple helpers in your [BackupAgentHelper](#), but only one helper is needed for each data type. That is, if you have multiple [SharedPreferences](#) files, then you need only one [SharedPreferencesBackupHelper](#).

For each helper you want to add to your [BackupAgentHelper](#), you must do the following during your [onCreate\(\)](#) method:

1. Instantiate an instance of the desired helper class. In the class constructor, you must specify the appropriate file(s) you want to backup.
2. Call [addHelper\(\)](#) to add the helper to your [BackupAgentHelper](#).

The following sections describe how to create a backup agent using each of the available helpers.

## Backing up SharedPreferences

When you instantiate a [SharedPreferencesBackupHelper](#), you must include the name of one or more [SharedPreferences](#) files.

For example, to back up a [SharedPreferences](#) file named "user\_preferences", a complete backup agent using [BackupAgentHelper](#) looks like this:

```
public class MyPrefsBackupAgent extends BackupAgentHelper {
    // The name of the SharedPreferences file
    static final String PREFS = "user_preferences";

    // A key to uniquely identify the set of backup data
    static final String PREFS_BACKUP_KEY = "prefs";

    // Allocate a helper and add it to the backup agent
    @Override
    public void onCreate() {
        SharedPreferencesBackupHelper helper = new SharedPreferencesBackupHelper();
        helper.addHelper(PREFS_BACKUP_KEY, helper);
    }
}
```

That's it! That's your entire backup agent. The [SharedPreferencesBackupHelper](#) includes all the code needed to backup and restore a [SharedPreferences](#) file.

When the Backup Manager calls [onBackup\(\)](#) and [onRestore\(\)](#), [BackupAgentHelper](#) calls your backup helpers to perform backup and restore for your specified files.

**Note:** [SharedPreferences](#) are threadsafe, so you can safely read and write the shared preferences file from your backup agent and other activities.

## Backing up other files

When you instantiate a [FileBackupHelper](#), you must include the name of one or more files that are saved to your application's [internal storage](#) (as specified by [getFilesDir\(\)](#), which is the same location where [openFileOutput\(\)](#) writes files).

For example, to backup two files named "scores" and "stats," a backup agent using [BackupAgentHelper](#) looks like this:

```
public class MyFileBackupAgent extends BackupAgentHelper {  
    // The name of the Shared Preferences file  
    static final String TOP_SCORES = "scores";  
    static final String PLAYER_STATS = "stats";  
  
    // A key to uniquely identify the set of backup data  
    static final String FILES_BACKUP_KEY = "myfiles";  
  
    // Allocate a helper and add it to the backup agent  
    void onCreate() {  
        FileBackupHelper helper = new FileBackupHelper(this, TOP_SCORES, PLAYER_STATS);  
        addHelper(FILES_BACKUP_KEY, helper);  
    }  
}
```

The [FileBackupHelper](#) includes all the code necessary to backup and restore files that are saved to your application's [internal storage](#).

However, reading and writing to files on internal storage is **not threadsafe**. To ensure that your backup agent does not read or write your files at the same time as your activities, you must use synchronized statements each time you perform a read or write. For example, in any Activity where you read and write the file, you need an object to use as the intrinsic lock for the synchronized statements:

```
// Object for intrinsic lock  
static final Object sDataLock = new Object();
```

Then create a synchronized statement with this lock each time you read or write the files. For example, here's a synchronized statement for writing the latest score in a game to a file:

```
try {  
    synchronized (MyActivity.sDataLock) {  
        File dataFile = new File(getFilesDir\(\), TOP_SCORES);  
        RandomAccessFile raFile = new RandomAccessFile(dataFile, "rw");  
        raFile.writeInt(score);  
    }  
} catch (IOException e) {  
    Log.e(TAG, "Unable to write to file");  
}
```

You should synchronize your read statements with the same lock.

Then, in your [BackupAgentHelper](#), you must override [onBackup\(\)](#) and [onRestore\(\)](#) to synchronize the backup and restore operations with the same intrinsic lock. For example, the MyFileBackupAgent example from above needs the following methods:

```
@Override  
public void onBackup(ParcelFileDescriptor oldState, BackupDataOutput data,  
    ParcelFileDescriptor newState) throws IOException {  
    // Hold the lock while the FileBackupHelper performs backup  
    synchronized (MyActivity.sDataLock) {  
        super.onBackup(oldState, data, newState);  
    }  
}
```

```

        }

    }

@Override
public void onRestore(BackupDataInput data, int appVersionCode,
        ParcelFileDescriptor newState) throws IOException {
    // Hold the lock while the FileBackupHelper restores the file
    synchronized (MyActivity.sDataLock) {
        super.onRestore(data, appVersionCode, newState);
    }
}

```

That's it. All you need to do is add your [FileBackupHelper](#) in the [onCreate\(\)](#) method and override [onBackup\(\)](#) and [onRestore\(\)](#) to synchronize read and write operations.

For an example implementation of [BackupAgentHelper](#) with [FileBackupHelper](#), see the [FileHelperExampleAgent](#) class in the [Backup and Restore](#) sample application.

## Checking the Restore Data Version

When the Backup Manager saves your data to cloud storage, it automatically includes the version of your application, as defined by your manifest file's [android:versionCode](#) attribute. Before the Backup Manager calls your backup agent to restore your data, it looks at the [android:versionCode](#) of the installed application and compares it to the value recorded in the restore data set. If the version recorded in the restore data set is *newer* than the application version on the device, then the user has downgraded their application. In this case, the Backup Manager will abort the restore operation for your application and not call your [onRestore\(\)](#) method, because the restore set is considered meaningless to an older version.

You can override this behavior with the [android:restoreAnyVersion](#) attribute. This attribute is either "true" or "false" to indicate whether you want to restore the application regardless of the restore set version. The default value is "false". If you define this to be "true" then the Backup Manager will ignore the [android:versionCode](#) and call your [onRestore\(\)](#) method in all cases. In doing so, you can manually check for the version difference in your [onRestore\(\)](#) method and take any steps necessary to make the data compatible if the versions conflict.

To help you handle different versions during a restore operation, the [onRestore\(\)](#) method passes you the version code included with the restore data set as the `appVersionCode` parameter. You can then query the current application's version code with the [PackageManager.versionCode](#) field. For example:

```

PackageManager info;
try {
    String name = getPackageName();
    info = getPackageManager().getPackageInfo(name, 0);
} catch (NameNotFoundException nnfe) {
    info = null;
}

int version;
if (info != null) {
    version = info.versionCode;
}

```

Then simply compare the `version` acquired from [PackageInfo](#) to the `appVersionCode` passed into [onRestore\(\)](#).

**Caution:** Be certain you understand the consequences of setting [android:restoreAnyVersion](#) to "true" for your application. If each version of your application that supports backup does not properly account for variations in your data format during [onRestore\(\)](#), then the data on the device could be saved in a format incompatible with the version currently installed on the device.

## Requesting Backup

You can request a backup operation at any time by calling [dataChanged\(\)](#). This method notifies the Backup Manager that you'd like to backup your data using your backup agent. The Backup Manager then calls your backup agent's [onBackup\(\)](#) method at an opportune time in the future. Typically, you should request a backup each time your data changes (such as when the user changes an application preference that you'd like to back up). If you call [dataChanged\(\)](#) several times consecutively, before the Backup Manager requests a backup from your agent, your agent still receives just one call to [onBackup\(\)](#).

**Note:** While developing your application, you can request a backup and initiate an immediate backup operation with the [bmgr tool](#).

## Requesting Restore

During the normal life of your application, you shouldn't need to request a restore operation. The system automatically checks for backup data and performs a restore when your application is installed. However, you can manually request a restore operation by calling [requestRestore\(\)](#), if necessary. In which case, the Backup Manager calls your [onRestore\(\)](#) implementation, passing the data from the current set of backup data.

**Note:** While developing your application, you can request a restore operation with the [bmgr tool](#).

## Testing Your Backup Agent

Once you've implemented your backup agent, you can test the backup and restore functionality with the following procedure, using [bmgr](#).

1. Install your application on a suitable Android system image
  - If using the emulator, create and use an AVD with Android 2.2 (API Level 8).
  - If using a device, the device must be running Android 2.2 or greater and have Google Play built in.
2. Ensure that backup is enabled
  - If using the emulator, you can enable backup with the following command from your SDK tools/ path:

```
adb shell bmgr enable true
```
  - If using a device, open the system **Settings**, select **Privacy**, then enable **Back up my data** and **Automatic restore**.
3. Open your application and initialize some data

If you've properly implemented backup in your application, then it should request a backup each time the data changes. For example, each time the user changes some data, your app should call [dataChanged\(\)](#), which adds a backup request to the Backup Manager queue. For testing purposes, you can also make a request with the following `bmgr` command:

```
adb shell bmgr backup your.package.name
```

4. Initiate a backup operation:

```
adb shell bmgr run
```

This forces the Backup Manager to perform all backup requests that are in its queue.

5. Uninstall your application:

```
adb uninstall your.package.name
```

6. Re-install your application.

If your backup agent is successful, all the data you initialized in step 4 is restored.

# App Install Location

## Quickview

- You can allow your application to install on the device's external storage.
- Some types of applications should **not** allow installation on the external storage.
- Installing on the external storage is ideal for large applications that are not tightly integrated with the system (most commonly, games).

## In this document

1. [Backward Compatibility](#)
2. [Applications That Should NOT Install on External Storage](#)
3. [Applications That Should Install on External Storage](#)

## See also

1. [`<manifest>`](#)

Beginning with API Level 8, you can allow your application to be installed on the external storage (for example, the device's SD card). This is an optional feature you can declare for your application with the [`an-droid:installLocation`](#) manifest attribute. If you do *not* declare this attribute, your application will be installed on the internal storage only and it cannot be moved to the external storage.

To allow the system to install your application on the external storage, modify your manifest file to include the [`android:installLocation`](#) attribute in the [`<manifest>`](#) element, with a value of either "preferExternal" or "auto". For example:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    android:installLocation="preferExternal"  
    ... >
```

If you declare "preferExternal", you request that your application be installed on the external storage, but the system does not guarantee that your application will be installed on the external storage. If the external storage is full, the system will install it on the internal storage. The user can also move your application between the two locations.

If you declare "auto", you indicate that your application may be installed on the external storage, but you don't have a preference of install location. The system will decide where to install your application based on several factors. The user can also move your application between the two locations.

When your application is installed on the external storage:

- There is no effect on the application performance so long as the external storage is mounted on the device.
- The .apk file is saved on the external storage, but all private user data, databases, optimized .dex files, and extracted native code are saved on the internal device memory.
- The unique container in which your application is stored is encrypted with a randomly generated key that can be decrypted only by the device that originally installed it. Thus, an application installed on an SD card works for only one device.
- The user can move your application to the internal storage through the system settings.

**Warning:** When the user enables USB mass storage to share files with a computer or unmounts the SD card via the system settings, the external storage is unmounted from the device and all applications running on the external storage are immediately killed.

## Backward Compatibility

The ability for your application to install on the external storage is a feature available only on devices running API Level 8 (Android 2.2) or greater. Existing applications that were built prior to API Level 8 will always install on the internal storage and cannot be moved to the external storage (even on devices with API Level 8). However, if your application is designed to support an API Level *lower than 8*, you can choose to support this feature for devices with API Level 8 or greater and still be compatible with devices using an API Level lower than 8.

To allow installation on external storage and remain compatible with versions lower than API Level 8:

1. Include the `android:installLocation` attribute with a value of "auto" or "preferExternal" in the [`<manifest>`](#) element.
2. Leave your `android:minSdkVersion` attribute as is (something *less than "8"*) and be certain that your application code uses only APIs compatible with that level.
3. In order to compile your application, change your build target to API Level 8. This is necessary because older Android libraries don't understand the `android:installLocation` attribute and will not compile your application when it's present.

When your application is installed on a device with an API Level lower than 8, the `android:installLocation` attribute is ignored and the application is installed on the internal storage.

**Caution:** Although XML markup such as this will be ignored by older platforms, you must be careful not to use programming APIs introduced in API Level 8 while your `minSdkVersion` is less than "8", unless you perform the work necessary to provide backward compatibility in your code.

## Applications That Should NOT Install on External Storage

When the user enables USB mass storage to share files with their computer (or otherwise unmounts or removes the external storage), any application installed on the external storage and currently running is killed. The system effectively becomes unaware of the application until mass storage is disabled and the external storage is remounted on the device. Besides killing the application and making it unavailable to the user, this can break some types of applications in a more serious way. In order for your application to consistently behave as expected, you **should not** allow your application to be installed on the external storage if it uses any of the following features, due to the cited consequences when the external storage is unmounted:

### Services

Your running [`Service`](#) will be killed and will not be restarted when external storage is remounted. You can, however, register for the [`ACTION\_EXTERNAL\_APPLICATIONS\_AVAILABLE`](#) broadcast Intent, which will notify your application when applications installed on external storage have become available to the system again. At which time, you can restart your Service.

### Alarm Services

Your alarms registered with [`AlarmManager`](#) will be cancelled. You must manually re-register any alarms when external storage is remounted.

### Input Method Engines

Your [`IME`](#) will be replaced by the default IME. When external storage is remounted, the user can open system settings to enable your IME again.

## Live Wallpapers

Your running [Live Wallpaper](#) will be replaced by the default Live Wallpaper. When external storage is remounted, the user can select your Live Wallpaper again.

## App Widgets

Your [App Widget](#) will be removed from the home screen. When external storage is remounted, your App Widget will *not* be available for the user to select until the system resets the home application (usually not until a system reboot).

## Account Managers

Your accounts created with [AccountManager](#) will disappear until external storage is remounted.

## Sync Adapters

Your [AbstractThreadedSyncAdapter](#) and all its sync functionality will not work until external storage is remounted.

## Device Administrators

Your [DeviceAdminReceiver](#) and all its admin capabilities will be disabled, which can have unforeseeable consequences for the device functionality, which may persist after external storage is remounted.

## Broadcast Receivers listening for "boot completed"

The system delivers the [ACTION\\_BOOT\\_COMPLETED](#) broadcast before the external storage is mounted to the device. If your application is installed on the external storage, it can never receive this broadcast.

If your application uses any of the features listed above, you **should not** allow your application to install on external storage. By default, the system *will not* allow your application to install on the external storage, so you don't need to worry about your existing applications. However, if you're certain that your application should never be installed on the external storage, then you should make this clear by declaring [an-droid:installLocation](#) with a value of "internalOnly". Though this does not change the default behavior, it explicitly states that your application should only be installed on the internal storage and serves as a reminder to you and other developers that this decision has been made.

# Applications That Should Install on External Storage

In simple terms, anything that does not use the features listed in the previous section are safe when installed on external storage. Large games are more commonly the types of applications that should allow installation on external storage, because games don't typically provide additional services when inactive. When external storage becomes unavailable and a game process is killed, there should be no visible effect when the storage becomes available again and the user restarts the game (assuming that the game properly saved its state during the normal [Activity lifecycle](#)).

If your application requires several megabytes for the APK file, you should carefully consider whether to enable the application to install on the external storage so that users can preserve space on their internal storage.

# Administration

If you are an enterprise administrator, you can take advantage of APIs and system capabilities to manage Android devices and control access.

## Blog Articles

### [Unifying Key Store Access in ICS](#)

Android 4.0 (ICS) comes with a number of enhancements that make it easier for people to bring their personal Android devices to work. In this post, we're going to have a look at the key store functionality.

# Device Administration

## In this document

1. [Device Administration API Overview](#)
  1. [How does it work?](#)
  2. [Policies](#)
2. [Sample Application](#)
3. [Developing a Device Administration Application](#)
  1. [Creating the manifest](#)
  2. [Implementing the code](#)

## Key classes

1. [DeviceAdminReceiver](#)
2. [DevicePolicyManager](#)
3. [DeviceAdminInfo](#)

## Related samples

1. [DeviceAdminSample](#)

Android 2.2 introduces support for enterprise applications by offering the Android Device Administration API. The Device Administration API provides device administration features at the system level. These APIs allow you to create security-aware applications that are useful in enterprise settings, in which IT professionals require rich control over employee devices. For example, the built-in Android Email application has leveraged the new APIs to improve Exchange support. Through the Email application, Exchange administrators can enforce password policies — including alphanumeric passwords or numeric PINs — across devices. Administrators can also remotely wipe (that is, restore factory defaults on) lost or stolen handsets. Exchange users can sync their email and calendar data.

This document is intended for developers who want to develop enterprise solutions for Android-powered devices. It discusses the various features provided by the Device Administration API to provide stronger security for employee devices that are powered by Android.

## Device Administration API Overview

Here are examples of the types of applications that might use the Device Administration API:

- Email clients.
- Security applications that do remote wipe.
- Device management services and applications.

## How does it work?

You use the Device Administration API to write device admin applications that users install on their devices. The device admin application enforces the desired policies. Here's how it works:

- A system administrator writes a device admin application that enforces remote/local device security policies. These policies could be hard-coded into the app, or the application could dynamically fetch policies from a third-party server.

- The application is installed on users' devices. Android does not currently have an automated provisioning solution. Some of the ways a sysadmin might distribute the application to users are as follows:
  - Google Play.
  - Enabling installation from another store.
  - Distributing the application through other means, such as email or websites.
- The system prompts the user to enable the device admin application. How and when this happens depends on how the application is implemented.
- Once users enable the device admin application, they are subject to its policies. Complying with those policies typically confers benefits, such as access to sensitive systems and data.

If users do not enable the device admin app, it remains on the device, but in an inactive state. Users will not be subject to its policies, and they will conversely not get any of the application's benefits—for example, they may not be able to sync data.

If a user fails to comply with the policies (for example, if a user sets a password that violates the guidelines), it is up to the application to decide how to handle this. However, typically this will result in the user not being able to sync data.

If a device attempts to connect to a server that requires policies not supported in the Device Administration API, the connection will not be allowed. The Device Administration API does not currently allow partial provisioning. In other words, if a device (for example, a legacy device) does not support all of the stated policies, there is no way to allow the device to connect.

If a device contains multiple enabled admin applications, the strictest policy is enforced. There is no way to target a particular admin application.

To uninstall an existing device admin application, users need to first unregister the application as an administrator.

## Policies

In an enterprise setting, it's often the case that employee devices must adhere to a strict set of policies that govern the use of the device. The Device Administration API supports the policies listed in Table 1. Note that the Device Administration API currently only supports passwords for screen lock:

**Table 1.** Policies supported by the Device Administration API.

Policy	Description
Password enabled	Requires that devices ask for PIN or passwords.
Minimum password length	Set the required number of characters for the password. For example, you can require PIN or passwords to have at least six characters.
Alphanumeric password required	Requires that passwords have a combination of letters and numbers. They may include symbolic characters.
Complex password required	Requires that passwords must contain at least a letter, a numerical digit, and a special symbol. Introduced in Android 3.0.
Minimum letters required in password	The minimum number of letters required in the password for all admins or a particular one. Introduced in Android 3.0.
Minimum lowercase letters re-	The minimum number of lowercase letters required in the password for all admins or a particular one. Introduced in Android 3.0.

quired in password	
Minimum non-letter characters required in password	The minimum number of non-letter characters required in the password for all admins or a particular one. Introduced in Android 3.0.
Minimum numerical digits required in password	The minimum number of numerical digits required in the password for all admins or a particular one. Introduced in Android 3.0.
Minimum symbols required in password	The minimum number of symbols required in the password for all admins or a particular one. Introduced in Android 3.0.
Minimum uppercase letters required in password	The minimum number of uppercase letters required in the password for all admins or a particular one. Introduced in Android 3.0.
Password expiration timeout	When the password will expire, expressed as a delta in milliseconds from when a device admin sets the expiration timeout. Introduced in Android 3.0.
Password history restriction	This policy prevents users from reusing the last $n$ unique passwords. This policy is typically used in conjunction with <a href="#">setPasswordExpirationTimeout()</a> , which forces users to update their passwords after a specified amount of time has elapsed. Introduced in Android 3.0.
Maximum failed password attempts	Specifies how many times a user can enter the wrong password before the device wipes its data. The Device Administration API also allows administrators to remotely reset the device to factory defaults. This secures data in case the device is lost or stolen.
Maximum inactivity time lock	Sets the length of time since the user last touched the screen or pressed a button before the device locks the screen. When this happens, users need to enter their PIN or passwords again before they can use their devices and access data. The value can be between 1 and 60 minutes.
Require storage encryption	Specifies that the storage area should be encrypted, if the device supports it. Introduced in Android 3.0.
Disable camera	Specifies that the camera should be disabled. Note that this doesn't have to be a permanent disabling. The camera can be enabled/disabled dynamically based on context, time, and so on. Introduced in Android 4.0.

## Other features

In addition to supporting the policies listed in the above table, the Device Administration API lets you do the following:

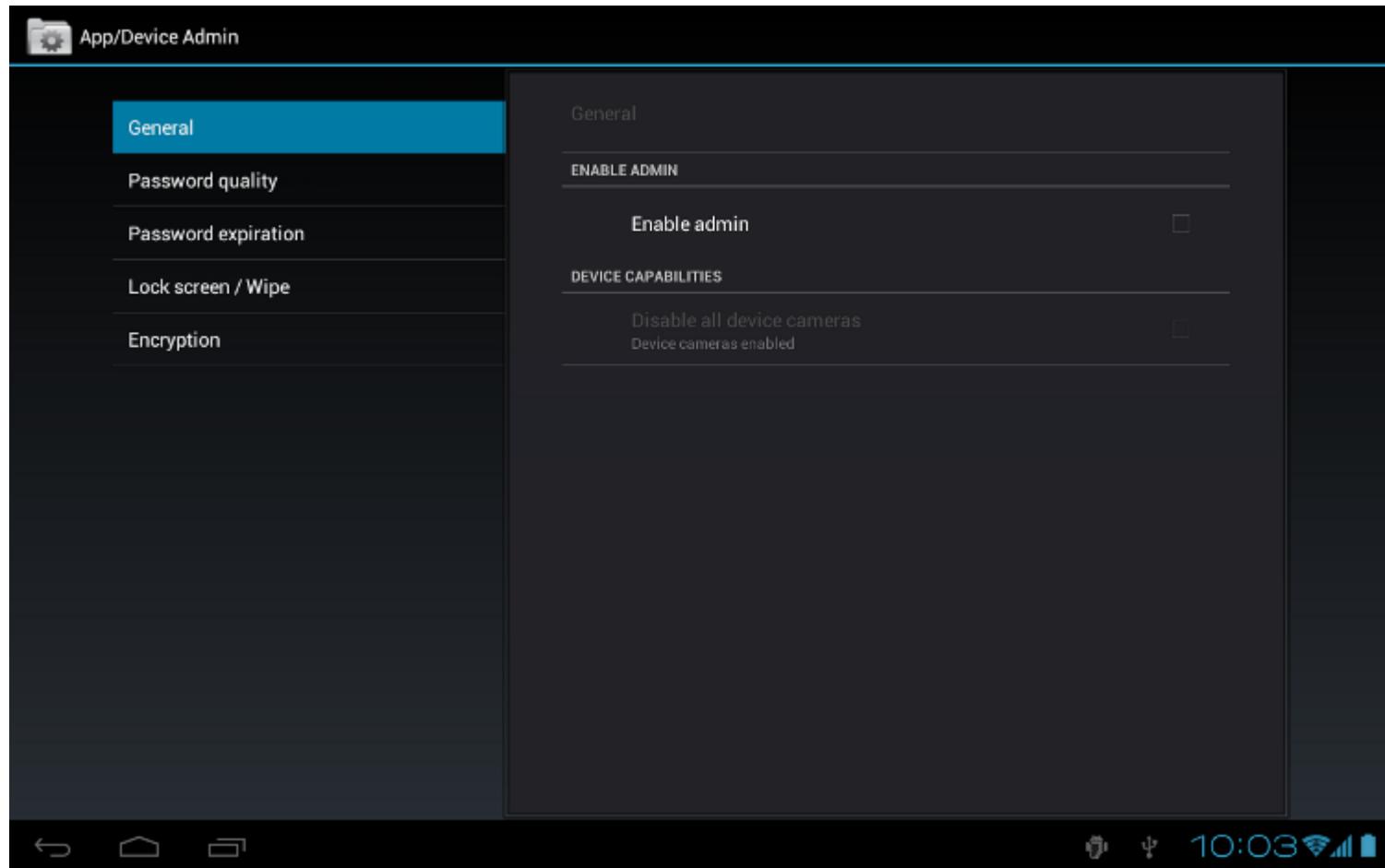
- Prompt user to set a new password.
- Lock device immediately.
- Wipe the device's data (that is, restore the device to its factory defaults).

## Sample Application

The examples used in this document are based on the [Device Administration API sample](#), which is included in the SDK samples. For information on downloading and installing the SDK samples, see [Getting the Samples](#). Here is the [complete code](#) for the sample.

The sample application offers a demo of device admin features. It presents users with a user interface that lets them enable the device admin application. Once they've enabled the application, they can use the buttons in the user interface to do the following:

- Set password quality.
- Specify requirements for the user's password, such as minimum length, the minimum number of numeric characters it must contain, and so on.
- Set the password. If the password does not conform to the specified policies, the system returns an error.
- Set how many failed password attempts can occur before the device is wiped (that is, restored to factory settings).
- Set how long from now the password will expire.
- Set the password history length (*length* refers to number of old passwords stored in the history). This prevents users from reusing one of the last *n* passwords they previously used.
- Specify that the storage area should be encrypted, if the device supports it.
- Set the maximum amount of inactive time that can elapse before the device locks.
- Make the device lock immediately.
- Wipe the device's data (that is, restore factory settings).
- Disable the camera.



**Figure 1.** Screenshot of the Sample Application

## Developing a Device Administration Application

System administrators can use the Device Administration API to write an application that enforces remote/local device security policy enforcement. This section summarizes the steps involved in creating a device administration application.

## Creating the manifest

To use the Device Administration API, the application's manifest must include the following:

- A subclass of [DeviceAdminReceiver](#) that includes the following:
  - The [BIND\\_DEVICE\\_ADMIN](#) permission.
  - The ability to respond to the [ACTION\\_DEVICE\\_ADMIN\\_ENABLED](#) intent, expressed in the manifest as an intent filter.
- A declaration of security policies used in metadata.

Here is an excerpt from the Device Administration sample manifest:

```
<activity android:name=".app.DeviceAdminSample"
          android:label="@string/activity_sample_device_admin">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.SAMPLE_CODE" />
    </intent-filter>
</activity>
<receiver android:name=".app.DeviceAdminSample$DeviceAdminSampleReceiver"
          android:label="@string/sample_device_admin"
          android:description="@string/sample_device_admin_description"
          android:permission="android.permission.BIND_DEVICE_ADMIN">
    <meta-data android:name="android.app.device_admin"
              android:resource="@xml/device_admin_sample" />
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

Note that:

- The following attributes refer to string resources that for the sample application reside in `ApiDemos/res/values/strings.xml`. For more information about resources, see [Application Resources](#).
  - `android:label="@string/activity_sample_device_admin"` refers to the user-readable label for the activity.
  - `android:label="@string/sample_device_admin"` refers to the user-readable label for the permission.
  - `android:description="@string/sample_device_admin_description"` refers to the user-readable description of the permission. A description is typically longer and more informative than a label.
- `android:permission="android.permission.BIND_DEVICE_ADMIN"` is a permission that a [DeviceAdminReceiver](#) subclass must have, to ensure that only the system can interact with the receiver (no application can be granted this permission). This prevents other applications from abusing your device admin app.
- `android.app.action.DEVICE_ADMIN_ENABLED` is the primary action that a [DeviceAdminReceiver](#) subclass must handle to be allowed to manage a device. This is set to the receiver when the user enables the device admin app. Your code typically handles this in [onEnabled\(\)](#). To be supported, the receiver must also require the [BIND\\_DEVICE\\_ADMIN](#) permission so that other applications cannot abuse it.
- When a user enables the device admin application, that gives the receiver permission to perform actions in response to the broadcast of particular system events. When suitable event arises, the application can impose a policy. For example, if the user attempts to set a new password that doesn't meet the policy re-

quirements, the application can prompt the user to pick a different password that does meet the requirements.

- `android:resource="@xml/device_admin_sample"` declares the security policies used in metadata. The metadata provides additional information specific to the device administrator, as parsed by the [DeviceAdminInfo](#) class. Here are the contents of `device_admin_sample.xml`:

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
<uses-policies>
    <limit-password />
    <watch-login />
    <reset-password />
    <force-lock />
    <wipe-data />
    <expire-password />
    <encrypted-storage />
    <disable-camera />
</uses-policies>
</device-admin>
```

In designing your device administration application, you don't need to include all of the policies, just the ones that are relevant for your app.

For more discussion of the manifest file, see the [Android Developers Guide](#).

## Implementing the code

The Device Administration API includes the following classes:

### [DeviceAdminReceiver](#)

Base class for implementing a device administration component. This class provides a convenience for interpreting the raw intent actions that are sent by the system. Your Device Administration application must include a [DeviceAdminReceiver](#) subclass.

### [DevicePolicyManager](#)

A class for managing policies enforced on a device. Most clients of this class must have published a [DeviceAdminReceiver](#) that the user has currently enabled. The [DevicePolicyManager](#) manages policies for one or more [DeviceAdminReceiver](#) instances

### [DeviceAdminInfo](#)

This class is used to specify metadata for a device administrator component.

These classes provide the foundation for a fully functional device administration application. The rest of this section describes how you use the [DeviceAdminReceiver](#) and [DevicePolicyManager](#) APIs to write a device admin application.

## Subclassing DeviceAdminReceiver

To create a device admin application, you must subclass [DeviceAdminReceiver](#). The [DeviceAdminReceiver](#) class consists of a series of callbacks that are triggered when particular events occur.

In its [DeviceAdminReceiver](#) subclass, the sample application simply displays a [Toast](#) notification in response to particular events. For example:

```

public class DeviceAdminSample extends DeviceAdminReceiver {

    void showToast(Context context, String msg) {
        String status = context.getString(R.string.admin_receiver_status, msg);
        Toast.makeText(context, status, Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onEnabled(Context context, Intent intent) {
        showToast(context, context.getString(R.string.admin_receiver_status_enabled));
    }

    @Override
    public CharSequence onDisableRequested(Context context, Intent intent) {
        return context.getString(R.string.admin_receiver_status_disable_warning);
    }

    @Override
    public void onDisabled(Context context, Intent intent) {
        showToast(context, context.getString(R.string.admin_receiver_status_disabled));
    }

    @Override
    public void onPasswordChanged(Context context, Intent intent) {
        showToast(context, context.getString(R.string.admin_receiver_status_password_changed));
    }
}

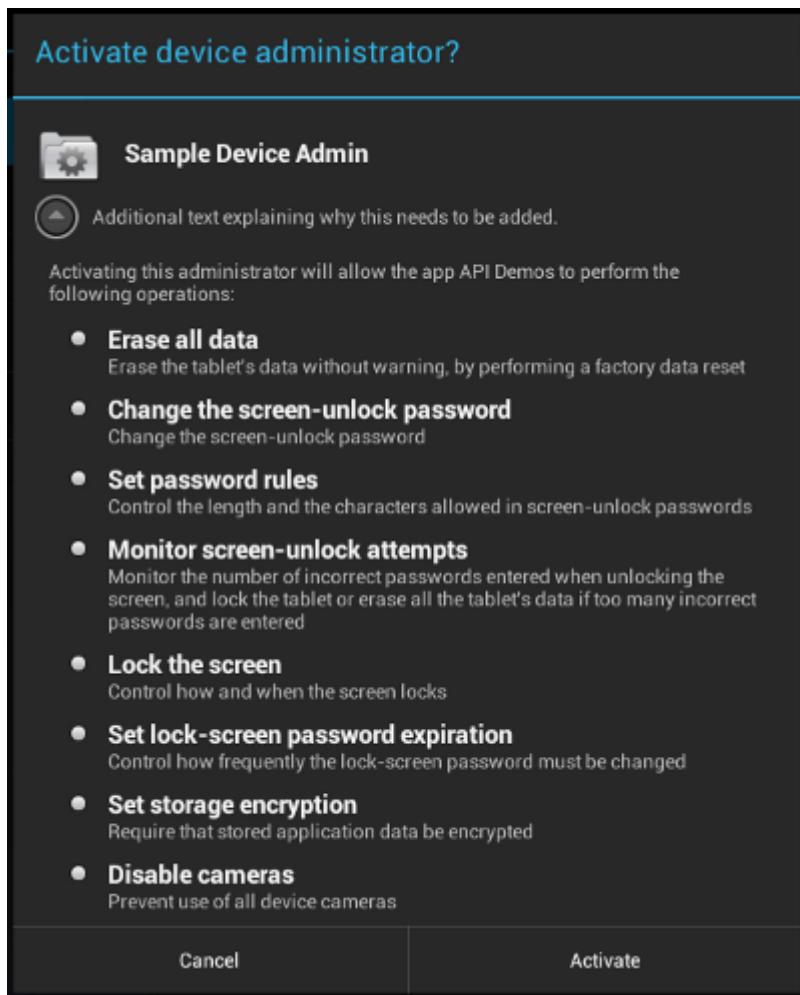
```

## Enabling the application

One of the major events a device admin application has to handle is the user enabling the application. The user must explicitly enable the application for the policies to be enforced. If the user chooses not to enable the application it will still be present on the device, but its policies will not be enforced, and the user will not get any of the application's benefits.

The process of enabling the application begins when the user performs an action that triggers the [ACTION\\_ADD\\_DEVICE\\_ADMIN](#) intent. In the sample application, this happens when the user clicks the **Enable Admin** checkbox.

When the user clicks the **Enable Admin** checkbox, the display changes to prompt the user to activate the device admin application, as shown in figure 2.



**Figure 2.** Sample Application: Activating the Application

Below is the code that gets executed when the user clicks the **Enable Admin** checkbox. This has the effect of triggering the `onPreferenceChange()` callback. This callback is invoked when the value of this `Pref.Preference` has been changed by the user and is about to be set and/or persisted. If the user is enabling the application, the display changes to prompt the user to activate the device admin application, as shown in figure 2. Otherwise, the device admin application is disabled.

```
@Override
public boolean onPreferenceChange(Preference preference, Object newValue) {
    if (super.onPreferenceChange(preference, newValue)) {
        return true;
    }
    boolean value = (Boolean) newValue;
    if (preference == mEnableCheckbox) {
        if (value != mAdminActive) {
            if (value) {
                // Launch the activity to have the user enable our admin
                Intent intent = new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
                intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, mDeviceAdminSample);
                intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
                               mActivity.getString(R.string.add_admin_extra_ap));
                startActivityForResult(intent, REQUEST_CODE_ENABLE_ADMIN);
                // return false - don't update checkbox until we're ready
                return false;
            } else {
                MDPM.removeActiveAdmin(mDeviceAdminSample);
            }
        }
    }
}
```

```

        enableDeviceCapabilitiesArea(false);
        mAdminActive = false;
    }
}
} else if (preference == mDisableCameraCheckbox) {
    mDPM.setCameraDisabled(mDeviceAdminSample, value);
    ...
}
return true;
}

```

The line `intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, mDeviceAdminSample)` states that `mDeviceAdminSample` (which is a [DeviceAdminReceiver](#) component) is the target policy. This line invokes the user interface shown in figure 2, which guides users through adding the device administrator to the system (or allows them to reject it).

When the application needs to perform an operation that is contingent on the device admin application being enabled, it confirms that the application is active. To do this it uses the [DevicePolicyManager](#) method `isAdminActive()`. Notice that the [DevicePolicyManager](#) method `isAdminActive()` takes a [DeviceAdminReceiver](#) component as its argument:

```

DevicePolicyManager mDPM;
...
private boolean isActiveAdmin() {
    return mDPM.isAdminActive(mDeviceAdminSample);
}

```

## Managing policies

[DevicePolicyManager](#) is a public class for managing policies enforced on a device. [DevicePolicyManager](#) manages policies for one or more [DeviceAdminReceiver](#) instances.

You get a handle to the [DevicePolicyManager](#) as follows:

```

DevicePolicyManager mDPM =
    (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);

```

This section describes how to use [DevicePolicyManager](#) to perform administrative tasks:

- [Set password policies](#)
- [Set device lock](#)
- [Perform data wipe](#)

### Set password policies

[DevicePolicyManager](#) includes APIs for setting and enforcing the device password policy. In the Device Administration API, the password only applies to screen lock. This section describes common password-related tasks.

#### Set a password for the device

This code displays a user interface prompting the user to set a password:

```
Intent intent = new Intent(DevicePolicyManager.ACTION_SET_NEW_PASSWORD);
startActivity(intent);
```

## Set the password quality

The password quality can be one of the following [DevicePolicyManager](#) constants:

### PASSWORD QUALITY ALPHABETIC

The user must enter a password containing at least alphabetic (or other symbol) characters.

### PASSWORD QUALITY ALPHANUMERIC

The user must enter a password containing at least *both* numeric *and* alphabetic (or other symbol) characters.

### PASSWORD QUALITY NUMERIC

The user must enter a password containing at least numeric characters.

### PASSWORD QUALITY COMPLEX

The user must have entered a password containing at least a letter, a numerical digit and a special symbol.

### PASSWORD QUALITY SOMETHING

The policy requires some kind of password, but doesn't care what it is.

### PASSWORD QUALITY UNSPECIFIED

The policy has no requirements for the password.

For example, this is how you would set the password policy to require an alphanumeric password:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
...
mDPM.setPasswordQuality(mDeviceAdminSample, DevicePolicyManager.PASSWORD_QUALITY_ALPHANUMERIC);
```

## Set password content requirements

Beginning with Android 3.0, the [DevicePolicyManager](#) class includes methods that let you fine-tune the contents of the password. For example, you could set a policy that states that passwords must contain at least *n* uppercase letters. Here are the methods for fine-tuning a password's contents:

- [setPasswordMinimumLetters\(\)](#)
- [setPasswordMinimumLowerCase\(\)](#)
- [setPasswordMinimumUpperCase\(\)](#)
- [setPasswordMinimumNonLetter\(\)](#)
- [setPasswordMinimumNumeric\(\)](#)
- [setPasswordMinimumSymbols\(\)](#)

For example, this snippet states that the password must have at least 2 uppercase letters:

```
DevicePolicyManager mDPM;
ComponentName mDeviceAdminSample;
int pwMinUppercase = 2;
...
mDPM.setPasswordMinimumUpperCase(mDeviceAdminSample, pwMinUppercase);
```

## **Set the minimum password length**

You can specify that a password must be at least the specified minimum length. For example:

```
DevicePolicyManager mDPM;  
ComponentName mDeviceAdminSample;  
int pwLength;  
...  
mDPM.setPasswordMinimumLength(mDeviceAdminSample, pwLength);
```

## **Set maximum failed password attempts**

You can set the maximum number of allowed failed password attempts before the device is wiped (that is, reset to factory settings). For example:

```
DevicePolicyManager mDPM;  
ComponentName mDeviceAdminSample;  
int maxFailedPw;  
...  
mDPM.setMaximumFailedPasswordsForWipe(mDeviceAdminSample, maxFailedPw);
```

## **Set password expiration timeout**

Beginning with Android 3.0, you can use the [setPasswordExpirationTimeout\(\)](#) method to set when a password will expire, expressed as a delta in milliseconds from when a device admin sets the expiration timeout. For example:

```
DevicePolicyManager mDPM;  
ComponentName mDeviceAdminSample;  
long pwExpiration;  
...  
mDPM.setPasswordExpirationTimeout(mDeviceAdminSample, pwExpiration);
```

## **Restrict password based on history**

Beginning with Android 3.0, you can use the [setPasswordHistoryLength\(\)](#) method to limit users' ability to reuse old passwords. This method takes a *length* parameter, which specifies how many old passwords are stored. When this policy is active, users cannot enter a new password that matches the last *n* passwords. This prevents users from using the same password over and over. This policy is typically used in conjunction with [setPasswordExpirationTimeout\(\)](#), which forces users to update their passwords after a specified amount of time has elapsed.

For example, this snippet prohibits users from reusing any of their last 5 passwords:

```
DevicePolicyManager mDPM;  
ComponentName mDeviceAdminSample;  
int pwHistoryLength = 5;  
...  
mDPM.setPasswordHistoryLength(mDeviceAdminSample, pwHistoryLength);
```

## **Set device lock**

You can set the maximum period of user inactivity that can occur before the device locks. For example:

```
DevicePolicyManager mDPM;  
ComponentName mDeviceAdminSample;  
...  
long timeMs = 1000L*Long.parseLong(mTimeout.getText().toString());  
mDPM.setMaximumTimeToLock(mDeviceAdminSample, timeMs);
```

You can also programmatically tell the device to lock immediately:

```
DevicePolicyManager mDPM;  
mDPM.lockNow();
```

## Perform data wipe

You can use the [DevicePolicyManager](#) method [wipeData\(\)](#) to reset the device to factory settings. This is useful if the device is lost or stolen. Often the decision to wipe the device is the result of certain conditions being met. For example, you can use [setMaximumFailedPasswordsForWipe\(\)](#) to state that a device should be wiped after a specific number of failed password attempts.

You wipe data as follows:

```
DevicePolicyManager mDPM;  
mDPM.wipeData(0);
```

The [wipeData\(\)](#) method takes as its parameter a bit mask of additional options. Currently the value must be 0.

## Disable camera

Beginning with Android 4.0, you can disable the camera. Note that this doesn't have to be a permanent disabling. The camera can be enabled/disabled dynamically based on context, time, and so on.

You control whether the camera is disabled by using the [setCameraDisabled\(\)](#) method. For example, this snippet sets the camera to be enabled or disabled based on a checkbox setting:

```
private CheckBoxPreference mDisableCameraCheckbox;  
DevicePolicyManager mDPM;  
ComponentName mDeviceAdminSample;  
...  
mDPM.setCameraDisabled(mDeviceAdminSample, mDisableCameraCheckbox.isChecked());
```

## Storage encryption

Beginning with Android 3.0, you can use the [setStorageEncryption\(\)](#) method to set a policy requiring encryption of the storage area, where supported.

For example:

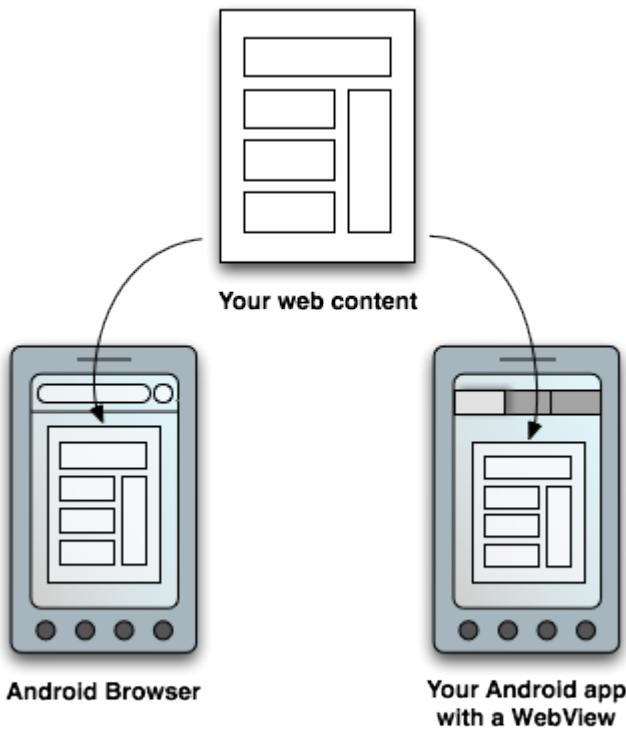
```
DevicePolicyManager mDPM;  
ComponentName mDeviceAdminSample;  
...  
mDPM.setStorageEncryption(mDeviceAdminSample, true);
```

See the [Device Administration API sample](#) for a complete example of how to enable storage encryption.

# Web Apps

Android has always been about connectivity and providing a great web browsing experience, so building your app with web technologies can be a great opportunity. Not only can you build an app on the web and still optimize your designs for Android's various screen sizes and densities, but you can also embed web-based content into your Android app using WebView.

# Web Apps Overview



**Figure 1.** You can make your web content available to users in two ways: in a traditional web browser and in an Android application, by including a `WebView` in the layout.

There are essentially two ways to deliver an application on Android: as a client-side application (developed using the Android SDK and installed on user devices as an `.apk`) or as a web application (developed using web standards and accessed through a web browser—there's nothing to install on user devices).

The approach you choose for your application could depend on several factors, but Android makes the decision to develop a web application easier by providing:

- Support for viewport properties that allow you to properly size your web application based on the screen size
- CSS and JavaScript features that allow you to provide different styles and images based on the screen's pixel density (screen resolution)

Thus, your decision to develop a web application for Android can exclude consideration for screen support, because it's already easy to make your web pages look good on all types of screens powered by Android.

Another great feature of Android is that you don't have to build your application purely on the client or purely on the web. You can mix the two together by developing a client-side Android application that embeds some web pages (using a [WebView](#) in your Android application layout). Figure 1 visualizes how you can provide access to your web pages from either a web browser or your Android application. However, you shouldn't develop an Android application simply as a means to launch your web site. Rather, the web pages you embed in your Android application should be designed especially for that environment. You can even define an interface between your Android application and your web pages that allows JavaScript in the web pages to call upon APIs in your Android application—providing Android APIs to your web-based application.

Since Android 1.0, [WebView](#) has been available for Android applications to embed web content in their layout and bind JavaScript to Android APIs. After Android added support for more screen densities (adding support for high and low-density screens), Android 2.0 added features to the WebKit framework to allow web pages to

specify viewport properties and query the screen density in order to modify styles and image assets, as mentioned above. Because these features are a part of Android's WebKit framework, both the Android Browser (the default web browser provided with the platform) and [WebView](#) support the same viewport and screen density features.

To develop a web application for Android-powered devices, you should read the following documents:

### [\*\*Targeting Screens from Web Apps\*\*](#)

How to properly size your web app on Android-powered devices and support multiple screen densities. The information in this document is important if you're building a web application that you at least expect to be available on Android-powered devices (which you should assume for anything you publish on the web), but especially if you're targeting mobile devices or using [WebView](#).

### [\*\*Building Web Apps in WebView\*\*](#)

How to embed web pages into your Android application using [WebView](#) and bind JavaScript to Android APIs.

### [\*\*Debugging Web Apps\*\*](#)

How to debug web apps using JavaScript Console APIs.

### [\*\*Best Practices for Web Apps\*\*](#)

A list of practices you should follow, in order to provide an effective web application on Android-powered devices.

# Targeting Screens from Web Apps

## Quickview

- You can target your web page for different screens using viewport metadata, CSS, and JavaScript
- Techniques in this document work for Android 2.0 and greater, and for web pages rendered in the default Android Browser and in a [WebView](#)

## In this document

1. [Using Viewport Metadata](#)
  1. [Defining the viewport size](#)
  2. [Defining the viewport scale](#)
  3. [Defining the viewport target density](#)
2. [Targeting Device Density with CSS](#)
3. [targeting Device Density with JavaScript](#)

If you're developing a web application for Android or redesigning one for mobile devices, you should carefully consider how your web pages appear on different kinds of screens. Because Android is available on devices with different types of screens, you should account for some factors that affect the way your web pages appear on Android devices.

**Note:** The features described in this document are supported by the Android Browser application (provided with the default Android platform) and [WebView](#) (the framework view widget for displaying web pages), on Android 2.0 and greater. Third-party web browsers running on Android might not support these features for controlling the viewport size and screen densities.

When targeting your web pages for Android devices, there are two fundamental factors that you should account for:

### The size of the viewport and scale of the web page

When the Android Browser loads a web page, the default behavior is to load the page in "overview mode," which provides a zoomed-out perspective of the web page. You can override this behavior for your web page by defining the default dimensions of the viewport or the initial scale of the viewport. You can also control how much the user can zoom in and out of your web page, if at all. The user can also disable overview mode in the Browser settings, so you should never assume that your page will load in overview mode. You should instead customize the viewport size and/or scale as appropriate for your page.

However, when your page is rendered in a [WebView](#), the page loads at full zoom (not in "overview mode"). That is, it appears at the default size for the page, instead of zoomed out. (This is also how the page appears if the user disables overview mode.)

### The device's screen density

The screen density (the number of pixels per inch) on an Android-powered device affects the resolution and size at which a web page is displayed. (There are three screen density categories: low, medium, and high.) The Android Browser and [WebView](#) compensate for variations in the screen density by scaling a web page so that all devices display the web page at the same perceivable size as a medium-density screen. If graphics are an important element of your web design, you should pay close attention to the scaling that occurs on different densities, because image scaling can produce artifacts (blurring and pixelation).

To provide the best visual representation on all screen densities, you should control how scaling occurs by providing viewport metadata about your web page's target screen density and providing alternative graphics for different screen densities, which you can apply to different screens using CSS or JavaScript.

The rest of this document describes how you can account for these effects and provide a good design on multiple types of screens.

## Using Viewport Metadata

The viewport is the area in which your web page is drawn. Although the viewport's visible area matches the size of the screen, the viewport has its own dimensions that determine the number of pixels available to a web page. That is, the number of pixels available to a web page before it exceeds the screen area is defined by the dimensions of the viewport, not the dimensions of the device screen. For example, although a device screen might have a width of 480 pixels, the viewport can have a width of 800 pixels, so that a web page designed to be 800 pixels wide is completely visible on the screen.

You can define properties of the viewport for your web page using the "viewport" property in an HTML `<meta>` tag (which must be placed in your document `<head>`). You can define multiple viewport properties in the `<meta>` tag's `content` attribute. For example, you can define the height and width of the viewport, the initial scale of the page, and the target screen density. Each viewport property in the `content` attribute must be separated by a comma.

For example, the following snippet from an HTML document specifies that the viewport width should exactly match the device screen width and that the ability to zoom should be disabled:

```
<head>
  <title>Example</title>
  <meta name="viewport" content="width=device-width, user-scalable=no" />
</head>
```

That's an example of just two viewport properties. The following syntax shows all of the supported viewport properties and the general types of values accepted by each one:

```
<meta name="viewport"
      content="
        height = [pixel_value | device-height] ,
        width = [pixel_value | device-width] ,
        initial-scale = float_value ,
        minimum-scale = float_value ,
        maximum-scale = float_value ,
        user-scalable = [yes | no] ,
        target-densitydpi = [dpi_value | device-dpi |
                               high-dpi | medium-dpi | low-dpi]
      " />
```

The following sections discuss how to use each of these viewport properties and exactly what the accepted values are.



**Figure 1.** A web page with an image that's 320 pixels wide, in the Android Browser when there is no viewport metadata set (with "overview mode" enabled, the viewport is 800 pixels wide, by default).



**Figure 2.** A web page with `viewport width=400` and "overview mode" enabled (the image in the web page is 320 pixels wide).

## Defining the viewport size

Viewport's `height` and `width` properties allow you to specify the size of the viewport (the number of pixels available to the web page before it goes off screen).

As mentioned in the introduction above, the Android Browser loads pages in "overview mode" by default (unless disabled by the user), which sets the minimum viewport width to 800 pixels. So, if your web page specifies its size to be 320 pixels wide, then your page appears smaller than the visible screen (even if the physical screen is 320 pixels wide, because the viewport simulates a drawable area that's 800 pixels wide), as shown in figure 1. To avoid this effect, you should explicitly define the `viewport width` to match the width for which you have designed your web page.

For example, if your web page is designed to be exactly 320 pixels wide, then you might want to specify that size for the `viewport width`:

```
<meta name="viewport" content="width=320" />
```

In this case, your web page exactly fits the screen width, because the web page width and viewport width are the same.

**Note:** Width values that are greater than 10,000 are ignored and values less than (or equal to) 320 result in a value equal to the device-width (discussed below). Height values that are greater than 10,000 or less than 200 are also ignored.

To demonstrate how this property affects the size of your web page, figure 2 shows a web page that contains an image that's 320 pixels wide, but with the viewport width set to 400.

**Note:** If you set the viewport width to match your web page width and the device's screen width does *not* match those dimensions, then the web page still fits the screen even if the device has a high or low-density screen, because the Android Browser and [WebView](#) scale web pages to match the perceived size on a medium-density screen, by default (as you can see in figure 2, when comparing the hdpi device to the mdpi device). Screen densities are discussed more in [Defining the viewport target density](#).

## Automatic sizing

As an alternative to specifying the viewport dimensions with exact pixels, you can set the viewport size to always match the dimensions of the device screen, by defining the viewport properties height and width with the values device-height and device-width, respectively. This is appropriate when you're developing a web application that has a fluid width (not fixed width), but you want it to appear as if it's fixed (to perfectly fit every screen as if the web page width is set to match each screen). For example:

```
<meta name="viewport" content="width=device-width" />
```

This results in the viewport width matching whatever the current screen width is, as shown in figure 3. It's important to notice that, this results in images being scaled to fit the screen when the current device does not match the [target density](#), which is medium-density if you don't specify otherwise. As a result, the image displayed on the high-density device in figure 3 is scaled up in order to match the width of a screen with a medium-density screen.



**Figure 3.** A web page with `viewport width=device-width or initial-scale=1.0`.

**Note:** If you instead want `device-width` and `device-height` to match the physical screen pixels for every device, instead of scaling your web page to match the target density, then you must also include the `target-densitydpi` property with a value of `device-dpi`. This is discussed more in the section about [Defining the viewport density](#). Otherwise, simply using `device-height` and `device-width` to define the viewport size makes your web page fit every device screen, but scaling occurs on your images in order to adjust for different screen densities.

## Defining the viewport scale

The scale of the viewport defines the level of zoom applied to the web page. Viewport properties allow you to specify the scale of your web page in the following ways:

### **initial-scale**

The initial scale of the page. The value is a float that indicates a multiplier for your web page size, relative to the screen size. For example, if you set the initial scale to "1.0" then the web page is displayed to match the resolution of the [target density](#) 1-to-1. If set to "2.0", then the page is enlarged (zoomed in) by a factor of 2.

The default initial scale is calculated to fit the web page in the viewport size. Because the default viewport width is 800 pixels, if the device screen resolution is less than 800 pixels wide, the initial scale is something less than 1.0, by default, in order to fit the 800-pixel-wide page on the screen.

### **minimum-scale**

The minimum scale to allow. The value is a float that indicates the minimum multiplier for your web page size, relative to the screen size. For example, if you set this to "1.0", then the page can't zoom out because the minimum size is 1-to-1 with the [target density](#).

### **maximum-scale**

The maximum scale to allow for the page. The value is a float that indicates the maximum multiplier for your web page size, relative to the screen size. For example, if you set this to "2.0", then the page can't zoom in more than 2 times the target size.

### **user-scalable**

Whether the user can change the scale of the page at all (zoom in and out). Set to `yes` to allow scaling and `no` to disallow scaling. The default is `yes`. If you set this to `no`, then the `minimum-scale` and `maximum-scale` are ignored, because scaling is not possible.

All scale values must be within the range 0.01–10.

For example:

```
<meta name="viewport" content="initial-scale=1.0" />
```

This metadata sets the initial scale to be full sized, relative to the viewport's target density.

## Defining the viewport target density

The density of a device's screen is based on the screen resolution, as defined by the number of dots per inch (dpi). There are three screen density categories supported by Android: low (ldpi), medium (mdpi), and high (hdpi). A screen with low density has fewer available pixels per inch, whereas a screen with high density has more pixels per inch (compared to a medium density screen). The Android Browser and [WebView](#) target a medium density screen by default.



**Figure 4.** A web page with `viewport width=device-width` and `target-densitydpi=device-dpi`.

Because the default target density is medium, when users have a device with a low or high density screen, the Android Browser and [WebView](#) scale web pages (effectively zoom the pages) so they display at a size that matches the perceived appearance on a medium density screen. More specifically, the Android Browser and [WebView](#) apply approximately 1.5x scaling to web pages on a high density screen (because its screen pixels are smaller) and approximately 0.75x scaling to pages on a low density screen (because its screen pixels are bigger).

Due to this default scaling, figures 1, 2, and 3 show the example web page at the same physical size on both the high and medium density device (the high-density device shows the web page with a default scale factor that is 1.5 times larger than the actual pixel resolution, to match the target density). This can introduce some undesirable artifacts in your images. For example, although an image appears the same size on a medium and high-density device, the image on the high-density device appears more blurry, because the image is designed to be 320 pixels wide, but is drawn with 480 pixels.

You can change the target screen density for your web page using the `target-densitydpi` `viewport` property. It accepts the following values:

- `device-dpi` - Use the device's native dpi as the target dpi. Default scaling never occurs.
- `high-dpi` - Use hdpi as the target dpi. Medium and low density screens scale down as appropriate.
- `medium-dpi` - Use mdpi as the target dpi. High density screens scale up and low density screens scale down. This is the default target density.
- `low-dpi` - Use ldpi as the target dpi. Medium and high density screens scale up as appropriate.
- `<value>` - Specify a dpi value to use as the target dpi. Values must be within the range 70–400.

For example, to prevent the Android Browser and [WebView](#) from scaling your web page for different screen densities, set the `target-densitydpi` `viewport` property to `device-dpi`. When you do, the page is not scaled. Instead, the page is displayed at a size that matches the current screen's density. In this case, you should also define the `viewport width` to match the device width, so your web page naturally fits the screen size. For example:

```
<meta name="viewport" content="target-densitydpi=device-dpi, width=device-width"/>
```

Figure 4 shows a web page using these `viewport` settings—the high-density device now displays the page smaller because its physical pixels are smaller than those on the medium-density device, so no scaling occurs and the 320-pixel-wide image is drawn using exactly 320 pixels on both screens. (This is how you should define your

viewport if you want to customize your web page based on screen density and provide different image assets for different densities, [with CSS](#) or [with JavaScript](#).)

## Targeting Device Density with CSS

The Android Browser and [WebView](#) support a CSS media feature that allows you to create styles for specific screen densities—the `-webkit-device-pixel-ratio` CSS media feature. The value you apply to this feature should be either "0.75", "1", or "1.5", to indicate that the styles are for devices with low density, medium density, or high density screens, respectively.

For example, you can create separate stylesheets for each density:

```
<link rel="stylesheet" media="screen and (-webkit-device-pixel-ratio: 1.5)" href="high-density.css"/>
<link rel="stylesheet" media="screen and (-webkit-device-pixel-ratio: 1.0)" href="medium-density.css"/>
<link rel="stylesheet" media="screen and (-webkit-device-pixel-ratio: 0.75)" href="low-density.css"/>
```



**Figure 5.** A web page with CSS that's targeted to specific screen densities using the `-webkit-device-pixel-ratio` media feature. Notice that the hdpi device shows a different image that's applied in CSS.

Or, specify the different styles in one stylesheet:

```
#header {
    background:url(medium-density-image.png);
}

@media screen and (-webkit-device-pixel-ratio: 1.5) {
    /* CSS for high-density screens */
    #header {
        background:url(high-density-image.png);
    }
}

@media screen and (-webkit-device-pixel-ratio: 0.75) {
    /* CSS for low-density screens */
    #header {
        background:url(low-density-image.png);
    }
}
```

```
    }  
}
```

**Note:** The default style for `#header` applies the image designed for medium-density devices in order to support devices running a version of Android less than 2.0, which do not support the `-webkit-device-pixel-ratio` media feature.

The types of styles you might want to adjust based on the screen density depend on how you've defined your viewport properties. To provide fully-customized styles that tailor your web page for each of the supported densities, you should set your viewport properties so the viewport width and density match the device. That is:

```
<meta name="viewport" content="target-densitydpi=device-dpi, width=device-width"
```

This way, the Android Browser and [WebView](#) do not perform scaling on your web page and the viewport width matches the screen width exactly. On their own, these viewport properties create results shown in figure 4. However, by adding some custom CSS using the `-webkit-device-pixel-ratio` media feature, you can apply different styles. For example, figure 5 shows a web page with these viewport properties and also some CSS added that applies a high-resolution image for high-density screens.

## Targeting Device Density with JavaScript

The Android Browser and [WebView](#) support a DOM property that allows you to query the density of the current device—the `window.devicePixelRatio` DOM property. The value of this property specifies the scaling factor used for the current device. For example, if the value of `window.devicePixelRatio` is "1.0", then the device is considered a medium density device and no scaling is applied by default; if the value is "1.5", then the device is considered a high density device and the page is scaled 1.5x by default; if the value is "0.75", then the device is considered a low density device and the page is scaled 0.75x by default. Of course, the scaling that the Android Browser and [WebView](#) apply is based on the web page's target density—as described in the section about [Defining the viewport target density](#), the default target is medium-density, but you can change the target to affect how your web page is scaled for different screen densities.

For example, here's how you can query the device density with JavaScript:

```
if (window.devicePixelRatio == 1.5) {  
  alert("This is a high-density screen");  
} else if (window.devicePixelRatio == 0.75) {  
  alert("This is a low-density screen");  
}
```

# Building Web Apps in WebView

## Quickview

- Use [WebView](#) to display web pages in your Android application layout
- You can create interfaces from your JavaScript to your client-side Android code

## In this document

1. [Adding a WebView to Your Application](#)
2. [Using JavaScript in WebView](#)
  1. [Enabling JavaScript](#)
  2. [Binding JavaScript code to Android code](#)
3. [Handling Page Navigation](#)
  1. [Navigating web page history](#)

## Key classes

1. [WebView](#)
2. [WebSettings](#)
3. [WebViewClient](#)

If you want to deliver a web application (or just a web page) as a part of a client application, you can do it using [WebView](#). The [WebView](#) class is an extension of Android's [View](#) class that allows you to display web pages as a part of your activity layout. It does *not* include any features of a fully developed web browser, such as navigation controls or an address bar. All that [WebView](#) does, by default, is show a web page.

A common scenario in which using [WebView](#) is helpful is when you want to provide information in your application that you might need to update, such as an end-user agreement or a user guide. Within your Android application, you can create an [Activity](#) that contains a [WebView](#), then use that to display your document that's hosted online.

Another scenario in which [WebView](#) can help is if your application provides data to the user that always requires an Internet connection to retrieve data, such as email. In this case, you might find that it's easier to build a [WebView](#) in your Android application that shows a web page with all the user data, rather than performing a network request, then parsing the data and rendering it in an Android layout. Instead, you can design a web page that's tailored for Android devices and then implement a [WebView](#) in your Android application that loads the web page.

This document shows you how to get started with [WebView](#) and how to do some additional things, such as handle page navigation and bind JavaScript from your web page to client-side code in your Android application.

## Adding a WebView to Your Application

To add a [WebView](#) to your Application, simply include the <WebView> element in your activity layout. For example, here's a layout file in which the [WebView](#) fills the screen:

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

```
    android:id="@+id/webview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
/>
```

To load a web page in the [WebView](#), use [loadUrl\(\)](#). For example:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.loadUrl("http://www.example.com");
```

Before this will work, however, your application must have access to the Internet. To get Internet access, request the [INTERNET](#) permission in your manifest file. For example:

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

That's all you need for a basic [WebView](#) that displays a web page.

## Using JavaScript in WebView

If the web page you plan to load in your [WebView](#) use JavaScript, you must enable JavaScript for your [WebView](#). Once JavaScript is enabled, you can also create interfaces between your application code and your JavaScript code.

### Enabling JavaScript

JavaScript is disabled in a [WebView](#) by default. You can enable it through the [WebSettings](#) attached to your [WebView](#). You can retrieve [WebSettings](#) with [getSettings\(\)](#), then enable JavaScript with [setJavaScriptEnabled\(\)](#).

For example:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
WebSettings webSettings = myWebView.getSettings();
webSettings.setJavaScriptEnabled(true);
```

[WebSettings](#) provides access to a variety of other settings that you might find useful. For example, if you're developing a web application that's designed specifically for the [WebView](#) in your Android application, then you can define a custom user agent string with [setUserAgentString\(\)](#), then query the custom user agent in your web page to verify that the client requesting your web page is actually your Android application.

from your Android SDK tools/ directory

### Binding JavaScript code to Android code

When developing a web application that's designed specifically for the [WebView](#) in your Android application, you can create interfaces between your JavaScript code and client-side Android code. For example, your JavaScript code can call a method in your Android code to display a [Dialog](#), instead of using JavaScript's [alert\(\)](#) function.

To bind a new interface between your JavaScript and Android code, call [addJavascriptInterface\(\)](#), passing it a class instance to bind to your JavaScript and an interface name that your JavaScript can call to access the class.

For example, you can include the following class in your Android application:

```
public class WebAppInterface {
    Context mContext;

    /** Instantiate the interface and set the context */
    WebAppInterface(Context c) {
        mContext = c;
    }

    /** Show a toast from the web page */
    @JavascriptInterface
    public void showToast(String toast) {
        Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show();
    }
}
```

**Caution:** If you've set your [targetSdkVersion](#) to 17 or higher, **you must add the `@JavascriptInterface` annotation** to any method that you want available to your JavaScript (the method must also be public). If you do not provide the annotation, the method is not accessible by your web page when running on Android 4.2 or higher.

In this example, the `WebAppInterface` class allows the web page to create a [Toast](#) message, using the `showToast()` method.

You can bind this class to the JavaScript that runs in your [WebView](#) with [addJavascriptInterface\(\)](#) and name the interface `Android`. For example:

```
WebView webView = (WebView) findViewById(R.id.webview);
webView.addJavascriptInterface(new WebAppInterface(this), "Android");
```

This creates an interface called `Android` for JavaScript running in the [WebView](#). At this point, your web application has access to the `WebAppInterface` class. For example, here's some HTML and JavaScript that creates a toast message using the new interface when the user clicks a button:

```
<input type="button" value="Say hello" onClick="showAndroidToast('Hello Android')>

<script type="text/javascript">
    function showAndroidToast(toast) {
        Android.showToast(toast);
    }
</script>
```

There's no need to initialize the `Android` interface from JavaScript. The [WebView](#) automatically makes it available to your web page. So, at the click of the button, the `showAndroidToast()` function uses the `Android` interface to call the `WebAppInterface.showToast()` method.

**Note:** The object that is bound to your JavaScript runs in another thread and not in the thread in which it was constructed.

**Caution:** Using `addJavascriptInterface()` allows JavaScript to control your Android application. This can be a very useful feature or a dangerous security issue. When the HTML in the `WebView` is untrustworthy (for example, part or all of the HTML is provided by an unknown person or process), then an attacker can include HTML that executes your client-side code and possibly any code of the attacker's choosing. As such, you should not use `addJavascriptInterface()` unless you wrote all of the HTML and JavaScript that appears in your `WebView`. You should also not allow the user to navigate to other web pages that are not your own, within your `WebView` (instead, allow the user's default browser application to open foreign links—by default, the user's web browser opens all URL links, so be careful only if you handle page navigation as described in the following section).

## Handling Page Navigation

When the user clicks a link from a web page in your `WebView`, the default behavior is for Android to launch an application that handles URLs. Usually, the default web browser opens and loads the destination URL. However, you can override this behavior for your `WebView`, so links open within your `WebView`. You can then allow the user to navigate backward and forward through their web page history that's maintained by your `WebView`.

To open links clicked by the user, simply provide a `WebViewClient` for your `WebView`, using `setWebViewClient()`. For example:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient());
```

That's it. Now all links the user clicks load in your `WebView`.

If you want more control over where a clicked link load, create your own `WebViewClient` that overrides the `shouldOverrideUrlLoading()` method. For example:

```
private class MyWebViewClient extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        if (Uri.parse(url).getHost().equals("www.example.com")) {
            // This is my web site, so do not override; let my WebView load the
            return false;
        }
        // Otherwise, the link is not for a page on my site, so launch another
        Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
        startActivity(intent);
        return true;
    }
}
```

Then create an instance of this new `WebViewClient` for the `WebView`:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new MyWebViewClient());
```

Now when the user clicks a link, the system calls `shouldOverrideUrlLoading()`, which checks whether the URL host matches a specific domain (as defined above). If it does match, then the method returns `false` in order to *not* override the URL loading (it allows the `WebView` to load the URL as usual). If the URL host does not match, then an `Intent` is created to launch the default Activity for handling URLs (which resolves to the user's default web browser).

## Navigating web page history

When your [WebView](#) overrides URL loading, it automatically accumulates a history of visited web pages. You can navigate backward and forward through the history with [goBack\(\)](#) and [goForward\(\)](#).

For example, here's how your [Activity](#) can use the device *Back* button to navigate backward:

```
@Override  
public boolean onKeyDown(int keyCode, KeyEvent event) {  
    // Check if the key event was the Back button and if there's history  
    if ((keyCode == KeyEvent.KEYCODE_BACK) && myWebView.canGoBack()) {  
        myWebView.goBack();  
        return true;  
    }  
    // If it wasn't the Back key or there's no web page history, bubble up to the  
    // system behavior (probably exit the activity)  
    return super.onKeyDown(keyCode, event);  
}
```

The [canGoBack\(\)](#) method returns true if there is actually web page history for the user to visit. Likewise, you can use [canGoForward\(\)](#) to check whether there is a forward history. If you don't perform this check, then once the user reaches the end of the history, [goBack\(\)](#) or [goForward\(\)](#) does nothing.

# Debugging Web Apps

## Quickview

- You can debug your web app using console methods in JavaScript
- If debugging in a custom WebView, you need to implement a callback method to handle debug messages

## In this document

1. [Using Console APIs in the Android Browser](#)
2. [Using Console APIs in WebView](#)

## See also

1. [Debugging](#)

If you're developing a web application for Android, you can debug your JavaScript using the `console` JavaScript APIs, which output messages to logcat. If you're familiar with debugging web pages with Firebug or Web Inspector, then you're probably familiar with using `console` (such as `console.log()`). Android's WebKit framework supports most of the same APIs, so you can receive logs from your web page when debugging in Android's Browser or in your own [WebView](#).

## Using Console APIs in the Android Browser

### Logcat

Logcat is a tool that dumps a log of system messages. The messages include a stack trace when the device throws an error, as well as log messages written from your application and those written using JavaScript `console` APIs.

To run logcat and view messages, execute `adb logcat` from your Android SDK tools/ directory, or, from DDMS, select **Device > Run logcat**. When using the [ADT plugin for Eclipse](#), you can also view logcat messages by opening the Logcat view, available from **Window > Show View > Other > Android > Logcat**.

See [Debugging](#) for more information about .

When you call a `console` function (in the DOM's `window.console` object), the output appears in logcat. For example, if your web page executes the following JavaScript:

```
console.log("Hello World");
```

Then the logcat message looks something like this:

```
Console: Hello World http://www.example.com/hello.html :82
```

The format of the message might appear different depending on which version of Android you're using. On Android 2.1 and higher, console messages from the Android Browser are tagged with the name "browser". On Android 1.6 and lower, Android Browser messages are tagged with the name "WebCore".

Android's WebKit does not implement all of the console APIs available in other desktop browsers. You can, however, use the basic text logging functions:

- `console.log(String)`
- `console.info(String)`
- `console.warn(String)`
- `console.error(String)`

Other console functions don't raise errors, but might not behave the same as what you expect from other web browsers.

## Using Console APIs in WebView

If you've implemented a custom [WebView](#) in your application, all the same console APIs are supported when debugging your web page in WebView. On Android 1.6 and lower, console messages are automatically sent to logcat with the "WebCore" logging tag. If you're targeting Android 2.1 (API Level 7) or higher, then you must provide a [WebChromeClient](#) that implements the [onConsoleMessage\(\)](#) callback method, in order for console messages to appear in logcat.

Additionally, the [onConsoleMessage\(String, int, String\)](#) method introduced in API Level 7 has been deprecated in favor of [onConsoleMessage\(ConsoleMessage\)](#) in API Level 8.

Whether you're developing for Android 2.1 (API Level 7) or Android 2.2 (API Level 8 or greater), you must implement [WebChromeClient](#) and override the appropriate [onConsoleMessage\(\)](#) callback method. Then, apply the [WebChromeClient](#) to your [WebView](#) with [setWebChromeClient\(\)](#).

Using API Level 7, this is how your code for [onConsoleMessage\(String, int, String\)](#) might look:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebChromeClient(new WebChromeClient() {
    public void onConsoleMessage(String message, int lineNumber, String sourceID) {
        Log.d("MyApplication", message + " -- From line "
              + lineNumber + " of "
              + sourceID);
    }
});
```

With API Level 8 or greater, your code for [onConsoleMessage\(ConsoleMessage\)](#) might look like this:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebChromeClient(new WebChromeClient() {
    public boolean onConsoleMessage(ConsoleMessage cm) {
        Log.d("MyApplication", cm.message() + " -- From line "
              + cm.lineNumber() + " of "
              + cm.sourceId());
        return true;
    }
});
```

The [ConsoleMessage](#) also includes a [MessageLevel](#) to indicate the type of console message being delivered. You can query the message level with [messageLevel\(\)](#) to determine the severity of the message, then use the appropriate [Log](#) method or take other appropriate actions.

Whether you're using `onConsoleMessage(String, int, String)` or `onConsoleMessage(ConsoleMessage)`, when you execute a console method in your web page, Android calls the appropriate `onConsoleMessage()` method so you can report the error. For example, with the example code above, a logcat message is printed that looks like this:

```
Hello World -- From line 82 of http://www.example.com/hello.html
```

# Best Practices for Web Apps

Developing web pages and web applications for mobile devices presents a different set of challenges compared to developing a web page for the typical desktop web browser. To help you get started, the following is a list of practices you should follow in order to provide the most effective web application for Android and other mobile devices.

## 1. Redirect mobile devices to a dedicated mobile version of your web site

There are several ways you can redirect requests to the mobile version of your web site, using server-side redirects. Most often, this is done by "sniffing" the User Agent string provided by the web browser. To determine whether to serve a mobile version of your site, you should simply look for the "mobile" string in the User Agent, which matches a wide variety of mobile devices. If necessary, you can also identify the specific operating system in the User Agent string (such as "Android 2.1").

**Note:** Large screen Android-powered devices that should be served full-size web sites (such as tablets) do *not* include the "mobile" string in the user agent, while the rest of the user agent string is mostly the same. As such, it's important that you deliver the mobile version of your web site based on whether the "mobile" string exists in the user agent.

## 2. Use a valid markup DOCTYPE that's appropriate for mobile devices

The most common markup language used for mobile web sites is [XHTML Basic](#). This standard ensures specific markup for your web site that works best on mobile devices. For instance, it does not allow HTML frames or nested tables, which perform poorly on mobile devices. Along with the DOCTYPE, be sure to declare the appropriate character encoding for the document (such as UTF-8).

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.1//EN"
  "http://www.w3.org/TR/xhtml-basic/xhtml-basic11.dtd">
```

Also be sure that your web page markup is valid against the declared DOCTYPE. Use a validator, such as the one available at <http://validator.w3.org>.

## 3. Use viewport meta data to properly resize your web page

In your document <head>, you should provide meta data that specifies how you want the browser's viewport to render your web page. For example, your viewport meta data can specify the height and width for the browser's viewport, the initial web page scale and even the target screen density.

For example:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0, use
```

For more information about how to use viewport meta data for Android-powered devices, read [Targeting Screens from Web Apps](#).

## 4. Avoid multiple file requests

Because mobile devices typically have a connection speed far slower than a desktop computer, you should make your web pages load as fast as possible. One way to speed it up is to avoid loading extra files such as stylesheets and script files in the <head>. Instead, provide your CSS and JavaScript di-

rectly in the <head> (or at the end of the <body>, for scripts that you don't need until the page is loaded). Alternatively, you should optimize the size and speed of your files by compressing them with tools like [Minify](#).

##### 5. Use a vertical linear layout

Avoid the need for the user to scroll left and right while navigating your web page. Scrolling up and down is easier for the user and makes your web page simpler.

For a more thorough guide to creating great mobile web applications, see the W3C's [Mobile Web Best Practices](#). For other guidance on improving the speed of your web site (for mobile and desktop), see Yahoo!'s guide to [Exceptional Performance](#) and Google's speed tutorials in [Let's make the web faster](#).

# Best Practices

Design and build apps the right way. Learn how to create apps that look great and perform well on as many devices as possible, from phones to tablets and more.

## Blog Articles

### Improving App Quality

One way of improving your app's visibility in the ecosystem is by deploying well-targeted mobile advertising campaigns and cross-app promotions. However, there's another time-tested method of fueling the impression-install-ranking cycle: improve the product!

### Say Goodbye to the Menu Button

As Ice Cream Sandwich rolls out to more devices, it's important that you begin to migrate your designs to the action bar in order to promote a consistent Android user experience.

### New Tools For Managing Screen Sizes

Android 3.2 includes new tools for supporting devices with a wide range of screen sizes. One important result is better support for a new size of screen; what is typically called a "7-inch" tablet. This release also offers several new APIs to simplify developers' work in adjusting to different screen sizes.

### Identifying App Installations

It is very common, and perfectly reasonable, for a developer to want to track individual installations of their apps. It sounds plausible just to call `TelephonyManager.getDeviceId()` and use that value to identify the installation. There are problems with this

### Making Android Games that Play Nice

Making a game on Android is easy. Making a *great* game for a mobile, multitasking, often multi-core, multi-purpose system like Android is trickier. Even the best developers frequently make mistakes in the way they interact with the Android system and with other applications

# Android Compatibility

## See also

1. [Filtering on Google Play](#)
2. [Providing Alternative Resources](#)
3. [Supporting Multiple Screens](#)
4. [`<supports-screens>`](#)
5. [`<uses-configuration>`](#)
6. [`<uses-feature>`](#)
7. [`<uses-library>`](#)
8. [`<uses-permission>`](#)
9. [`<uses-sdk>`](#)

Android is designed to run on many different types of devices. For developers, the range and number of devices means a huge potential audience: the more devices that run Android apps, the more users who can access your app. In exchange, however, it also means that your apps will have to cope with that same variety of hardware.

Fortunately, Android has built-in tools and support that make it easy for your apps to do that, while at the same time letting you maintain control of what types of devices your app is available to. With a bit of forethought and some minor changes in your app's manifest file, you can ensure that users whose devices can't run your app will never see it on Google Play, and will not get in trouble by downloading it. This page explains how you can control which devices have access to your apps, and how to prepare your apps to make sure they reach the right audience.

## What does “compatibility” mean?

A device is “Android compatible” if it can correctly run apps written for the *Android execution environment*. The exact details of the Android execution environment are defined by the Android Compatibility Definition Document, but the single most important characteristic of a compatible device is the ability to install and correctly run an Android .apk file.

There is exactly one Android API for each [API level](#), and it's the same API no matter what kind of device it's installed on. No parts of the API are optional, and you never have to worry about parts of the API missing on some devices. Every compatible Android device your app will land on will include every class and every API for that API level.

Of course, some APIs won't work correctly if a particular device lacks the corresponding hardware or feature. But that's not a problem: we also designed Android to prevent apps from being visible to devices which don't have features the apps require. We've built support for this right into the SDK tools, and it's part of the Android platform itself, as well as part of Google Play.

As a developer, you have complete control of how and where your apps are available. Android provides tools as a first-class part of the platform that let you manage this. You control the availability of your apps, so that they reach only the devices capable of running them.

## How does it work?

You manage your app's availability through a simple three-step process:

1. You state the features your app requires by declaring [`<uses-feature>`](#) elements its manifest file.
2. Devices are required to declare the features they include to Google Play.

3. Google Play uses your app's stated requirements to filter it from devices that don't meet those requirements.

This way, users never even see apps that won't work properly on their devices. As long as you accurately describe your app's requirements, you don't need to worry about users blaming you for compatibility problems.

If you're familiar with web development, you may recognize this model as "capability detection". Web developers typically prefer this approach to "browser detection", because it's very difficult to keep up as new browsers and new versions of current browsers are released. By checking for support for specific required capabilities instead of the current browser, web developers get better fine-grained control. That's the same approach Android uses: since it's impossible to keep up with all the Android devices being released, you instead use the fine-grained controls Android provides.

## Filtering for technical reasons

---



### Filtering on Google Play

Google Play filters the applications that are visible to users, so that users can see and download only those applications that are compatible with their devices.

One of the ways Google Play filters applications is by feature compatibility. To do this, Google Play checks the `<uses-feature>` elements in each application's manifest, to establish the app's feature needs. Google Play then shows or hides the application to each user, based on a comparison with the features available on the user's device.

For information about other filters that you can use to control the availability of your apps, see the [Filters on Google Play](#) document.

Android includes support for a lot of features, some hardware and some software. Examples include compass and accelerometer sensors, cameras, and Live Wallpapers. However, not every device will support every feature. For instance, some devices don't have the hardware horsepower to display Live Wallpapers well.

To manage this, Android defines *feature IDs*. Every capability has a corresponding feature ID defined by the Android platform. For instance, the feature ID for compass is `"android.hardware.sensor.compass"`, while the feature ID for Live Wallpapers is `"android.software.live_wallpapers"`. Each of these IDs also has a corresponding Java-language constant on the [PackageManager](#) class that you can use to query whether feature is supported at runtime. As Android adds support for new features in future versions, new feature IDs will be added as well.

When you write your application, you specify which features your app requires by listing their feature IDs in `<uses-feature>` elements in the `AndroidManifest.xml` file. This is the information that Google Play uses to match your app to devices that can run it. For instance, if you state that your app requires `android.software.live_wallpapers`, it won't be shown to devices that don't support Live Wallpapers.

This puts you in total control of your app — because you don't have to declare these features. Consider an example involving cameras.

If you're building a really impressive next-generation augmented-reality app, your app won't function at all without a camera. However, if you're building a shopping app that only uses the camera for barcode scanning, users without cameras might still find it useful even if they can't scan barcodes. While both apps need to ac-

quire the permission to access the camera, only the first app needs to state that it requires a camera. (The shopping app can simply check at runtime and disable the camera-related features if there's no camera present.)

Since only you can say what the best approach is for your app, Android provides the tools and lets you make your own tradeoff between maximizing audience size and minimizing development costs.

## Filtering for business reasons

It's possible that you may need to restrict your app's availability for business or legal reasons. For instance, an app that displays train schedules for the London Underground is unlikely to be useful to users outside the United Kingdom. Other apps might not be permitted in certain countries for business or legal reasons. For cases such as these, Google Play itself provides developers with filtering options that allow them control their app's availability for non-technical reasons.

The help information for Google Play provides full details, but in a nutshell, developers can use the Google Play publisher UI to:

- List the countries an app is available in.
- Select which carrier's users are able to access the app.

Filtering for technical compatibility (such as required hardware components) is always based on information contained within your .apk file. But filtering for non-technical reasons (such as geographic restrictions) is always handled in the Google Play user interface.

## Future-proofing

There's one additional quirk that we haven't yet addressed: protecting apps from changes made to future versions of Android. If the Android platform introduces a new feature or changes how existing features are handled, what happens to existing apps that were written without any knowledge of the new behavior?

Simply put, Android commits to not making existing apps available to devices where they won't work properly, even when the platform changes. The best way to explain this is through examples, so here are two:

- Android 1.0 through 1.5 required a 2 megapixel camera with auto-focus. However, with version 1.6, Android devices were permitted to omit the auto-focus capability, though a (fixed-focus) camera was still required. Some apps such as barcode scanners do not function as well with cameras that do not auto-focus. To prevent users from having a bad experience with those apps, existing apps that obtain permission to use the Camera were assumed by default to require auto-focus. This allowed Google Play to filter those apps from devices that lack auto-focus.
- Android 2.2, meanwhile, allowed the microphone to be optional on some devices, such as set-top boxes. Android 2.2 included a new feature ID for the microphone which allows developers to filter their apps if necessary, but — as with camera — apps that obtain permission to record audio are assumed to require the microphone feature by default. If your app can use a microphone but doesn't strictly need it, you can explicitly state that you don't require it; but unless you do that, your app won't be shown to devices without microphones.

In other words, whenever Android introduces new features or changes existing ones, we will always take steps to protect existing applications so that they don't end up being available to devices where they won't work.

This is implemented, in part, using the aapt tool in the SDK. To see which features your app explicitly requires or is implicitly assumed to require, you can use the command `aapt dump badging`.

## Conclusion

The goal of Android is to create a huge installed base for developers to take advantage of. One of the ways we will achieve this is through different kinds of hardware running the same software environment. But we also recognize that only developers know which kinds of devices their apps make sense on. We've built in tools to the SDK and set up policies and requirements to ensure that developers remain in control of their apps, today and in the future. With the information you just read, and the resources listed in the sidebar of this document, you can publish your app with the confidence that only users who can run it will see it.

For more information about Android device compatibility, please visit:

<http://source.android.com/compatibility/index.html>

# Supporting Multiple Screens

## Quickview

- Android runs on devices that have different screen sizes and densities.
- The screen on which your application is displayed can affect its user interface.
- The system handles most of the work of adapting your app to the current screen.
- You should create screen-specific resources for precise control of your UI.

## In this document

1. [Overview of Screen Support](#)
  1. [Terms and concepts](#)
  2. [Range of screens supported](#)
  3. [Density independence](#)
2. [How to Support Multiple Screens](#)
  1. [Using configuration qualifiers](#)
  2. [Designing alternative layouts and drawables](#)
3. [Declaring Tablet Layouts for Android 3.2](#) new!
  1. [Using new size qualifiers](#)
  2. [Configuration examples](#)
  3. [Declaring screen size support](#)
4. [Best Practices](#)
5. [Additional Density Considerations](#)
  1. [Scaling Bitmap objects created at runtime](#)
  2. [Converting dp units to pixel units](#)
6. [How to Test Your Application on Multiple Screens](#)

## Related samples

1. [Multiple Resolutions](#)

## See also

1. [Thinking Like a Web Designer](#)
2. [Providing Alternative Resources](#)
3. [Icon Design Guidelines](#)
4. [Managing Virtual Devices](#)

Android runs on a variety of devices that offer different screen sizes and densities. For applications, the Android system provides a consistent development environment across devices and handles most of the work to adjust each application's user interface to the screen on which it is displayed. At the same time, the system provides APIs that allow you to control your application's UI for specific screen sizes and densities, in order to optimize your UI design for different screen configurations. For example, you might want a UI for tablets that's different from the UI for handsets.

Although the system performs scaling and resizing to make your application work on different screens, you should make the effort to optimize your application for different screen sizes and densities. In doing so, you maximize the user experience for all devices and your users believe that your application was actually designed for *their* devices—rather than simply stretched to fit the screen on their devices.

By following the practices described in this document, you can create an application that displays properly and provides an optimized user experience on all supported screen configurations, using a single .apk file.

**Note:** The information in this document assumes that your application is designed for Android 1.6 (API Level 4) or higher. If your application supports Android 1.5 or lower, please first read [Strategies for Android 1.5](#).

Also, be aware that **Android 3.2 has introduced new APIs** that allow you to more precisely control the layout resources your application uses for different screen sizes. These new features are especially important if you're developing an application that's optimized for tablets. For details, see the section about [Declaring Tablet Layouts for Android 3.2](#).

## Overview of Screens Support

This section provides an overview of Android's support for multiple screens, including: an introduction to the terms and concepts used in this document and in the API, a summary of the screen configurations that the system supports, and an overview of the API and underlying screen-compatibility features.

### Terms and concepts

#### *Screen size*

Actual physical size, measured as the screen's diagonal.

For simplicity, Android groups all actual screen sizes into four generalized sizes: small, normal, large, and extra large.

#### *Screen density*

The quantity of pixels within a physical area of the screen; usually referred to as dpi (dots per inch). For example, a "low" density screen has fewer pixels within a given physical area, compared to a "normal" or "high" density screen.

For simplicity, Android groups all actual screen densities into four generalized densities: low, medium, high, and extra high.

#### *Orientation*

The orientation of the screen from the user's point of view. This is either landscape or portrait, meaning that the screen's aspect ratio is either wide or tall, respectively. Be aware that not only do different devices operate in different orientations by default, but the orientation can change at runtime when the user rotates the device.

#### *Resolution*

The total number of physical pixels on a screen. When adding support for multiple screens, applications do not work directly with resolution; applications should be concerned only with screen size and density, as specified by the generalized size and density groups.

#### *Density-independent pixel (dp)*

A virtual pixel unit that you should use when defining UI layout, to express layout dimensions or position in a density-independent way.

The density-independent pixel is equivalent to one physical pixel on a 160 dpi screen, which is the baseline density assumed by the system for a "medium" density screen. At runtime, the system transparently handles any scaling of the dp units, as necessary, based on the actual density of the screen in use. The conversion of dp units to screen pixels is simple:  $\text{px} = \text{dp} * (\text{dpi} / 160)$ . For example, on a 240 dpi screen, 1 dp equals 1.5 physical pixels. You should always use dp units when defining your application's UI, to ensure proper display of your UI on screens with different densities.

## Range of screens supported

Starting with Android 1.6 (API Level 4), Android provides support for multiple screen sizes and densities, reflecting the many different screen configurations that a device may have. You can use features of the Android system to optimize your application's user interface for each screen configuration and ensure that your application not only renders properly, but provides the best user experience possible on each screen.

To simplify the way that you design your user interfaces for multiple screens, Android divides the range of actual screen sizes and densities into:

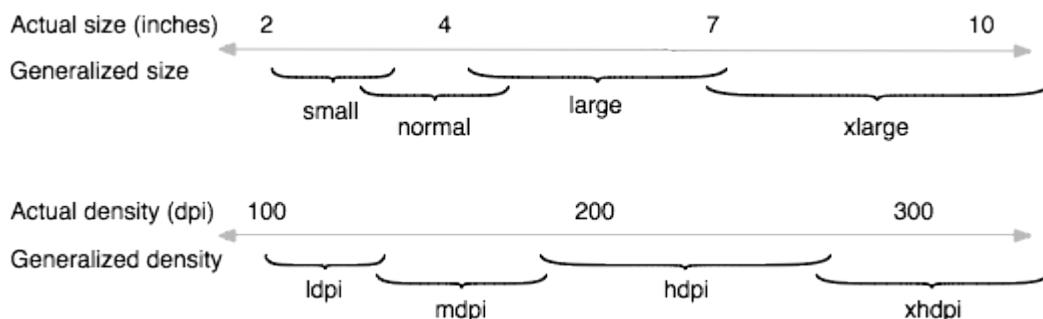
- A set of four generalized **sizes**: *small*, *normal*, *large*, and *xlarge*

**Note:** Beginning with Android 3.2 (API level 13), these size groups are deprecated in favor of a new technique for managing screen sizes based on the available screen width. If you're developing for Android 3.2 and greater, see [Declaring Tablet Layouts for Android 3.2](#) for more information.

- A set of four generalized **densities**: *ldpi* (low), *mdpi* (medium), *hdpi* (high), and *xhdpi* (extra high)

The generalized sizes and densities are arranged around a baseline configuration that is a *normal* size and *mdpi* (medium) density. This baseline is based upon the screen configuration for the first Android-powered device, the T-Mobile G1, which has an HVGA screen (until Android 1.6, this was the only screen configuration that Android supported).

Each generalized size and density spans a range of actual screen sizes and densities. For example, two devices that both report a screen size of *normal* might have actual screen sizes and aspect ratios that are slightly different when measured by hand. Similarly, two devices that report a screen density of *hdpi* might have real pixel densities that are slightly different. Android makes these differences abstract to applications, so you can provide UI designed for the generalized sizes and densities and let the system handle any final adjustments as necessary. Figure 1 illustrates how different sizes and densities are roughly categorized into the different size and density groups.



**Figure 1.** Illustration of how Android roughly maps actual sizes and densities to generalized sizes and densities (figures are not exact).

As you design your UI for different screen sizes, you'll discover that each design requires a minimum amount of space. So, each generalized screen size above has an associated minimum resolution that's defined by the system. These minimum sizes are in "dp" units—the same units you should use when defining your layouts—which allows the system to avoid worrying about changes in screen density.

- *xlarge* screens are at least 960dp x 720dp
- *large* screens are at least 640dp x 480dp
- *normal* screens are at least 470dp x 320dp
- *small* screens are at least 426dp x 320dp

**Note:** These minimum screen sizes were not as well defined prior to Android 3.0, so you may encounter some devices that are mis-classified between normal and large. These are also based on the physical resolution of the screen, so may vary across devices—for example a 1024x720 tablet with a system bar actually has a bit less space available to the application due to it being used by the system bar.

To optimize your application's UI for the different screen sizes and densities, you can provide [alternative resources](#) for any of the generalized sizes and densities. Typically, you should provide alternative layouts for some of the different screen sizes and alternative bitmap images for different screen densities. At runtime, the system uses the appropriate resources for your application, based on the generalized size or density of the current device screen.

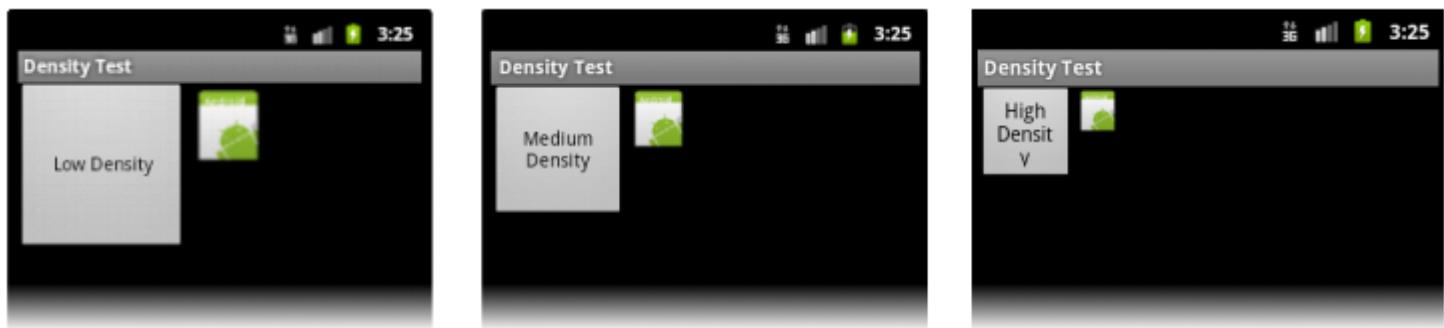
You do not need to provide alternative resources for every combination of screen size and density. The system provides robust compatibility features that can handle most of the work of rendering your application on any device screen, provided that you've implemented your UI using techniques that allow it to gracefully resize (as described in the [Best Practices](#), below).

**Note:** The characteristics that define a device's generalized screen size and density are independent from each other. For example, a WVGA high-density screen is considered a normal size screen because its physical size is about the same as the T-Mobile G1 (Android's first device and baseline screen configuration). On the other hand, a WVGA medium-density screen is considered a large size screen. Although it offers the same resolution (the same number of pixels), the WVGA medium-density screen has a lower screen density, meaning that each pixel is physically larger and, thus, the entire screen is larger than the baseline (normal size) screen.

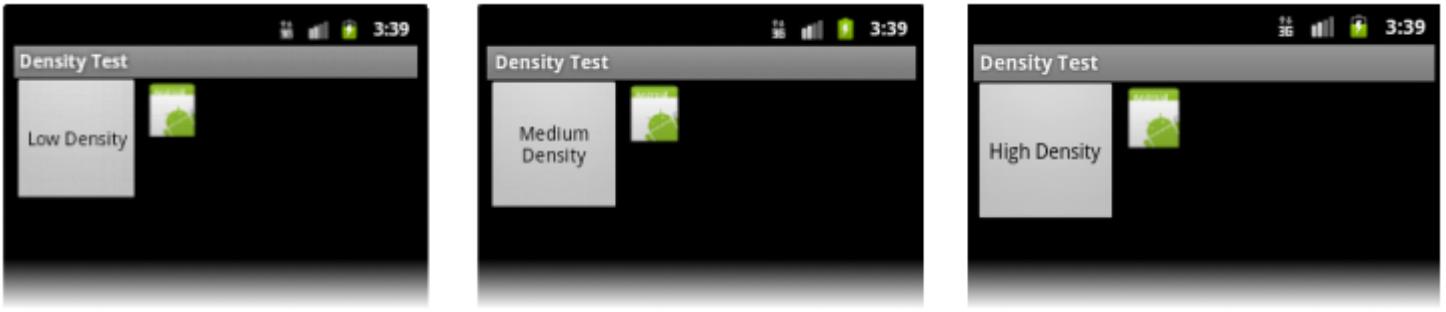
## Density independence

Your application achieves "density independence" when it preserves the physical size (from the user's point of view) of user interface elements when displayed on screens with different densities.

Maintaining density independence is important because, without it, a UI element (such as a button) appears physically larger on a low density screen and smaller on a high density screen. Such density-related size changes can cause problems in your application layout and usability. Figures 2 and 3 show the difference between an application when it does not provide density independence and when it does, respectively.



**Figure 2.** Example application without support for different densities, as shown on low, medium, and high density screens.



**Figure 3.** Example application with good support for different densities (it's density independent), as shown on low, medium, and high density screens.

The Android system helps your application achieve density independence in two ways:

- The system scales dp units as appropriate for the current screen density
- The system scales drawable resources to the appropriate size, based on the current screen density, if necessary

In figure 2, the text view and bitmap drawable have dimensions specified in pixels (px units), so the views are physically larger on a low density screen and smaller on a high density screen. This is because although the actual screen sizes may be the same, the high density screen has more pixels per inch (the same amount of pixels fit in a smaller area). In figure 3, the layout dimensions are specified in density-independent pixels (dp units). Because the baseline for density-independent pixels is a medium-density screen, the device with a medium-density screen looks the same as it does in figure 2. For the low-density and high-density screens, however, the system scales the density-independent pixel values down and up, respectively, to fit the screen as appropriate.

In most cases, you can ensure density independence in your application simply by specifying all layout dimension values in density-independent pixels (dp units) or with "wrap\_content", as appropriate. The system then scales bitmap drawables as appropriate in order to display at the appropriate size, based on the appropriate scaling factor for the current screen's density.

However, bitmap scaling can result in blurry or pixelated bitmaps, which you might notice in the above screenshots. To avoid these artifacts, you should provide alternative bitmap resources for different densities. For example, you should provide higher-resolution bitmaps for high-density screens and the system will use those instead of resizing the bitmap designed for medium-density screens. The following section describes more about how to supply alternative resources for different screen configurations.

## How to Support Multiple Screens

The foundation of Android's support for multiple screens is its ability to manage the rendering of an application's layout and bitmap drawables in an appropriate way for the current screen configuration. The system handles most of the work to render your application properly on each screen configuration by scaling layouts to fit the screen size/density and scaling bitmap drawables for the screen density, as appropriate. To more gracefully handle different screen configurations, however, you should also:

- **Explicitly declare in the manifest which screen sizes your application supports**

By declaring which screen sizes your application supports, you can ensure that only devices with the screens you support can download your application. Declaring support for different screen sizes can also affect how the system draws your application on larger screens—specifically, whether your application runs in [screen compatibility mode](#).

To declare the screen sizes your application supports, you should include the [`<supports-screens>`](#) element in your manifest file.

- **Provide different layouts for different screen sizes**

By default, Android resizes your application layout to fit the current device screen. In most cases, this works fine. In other cases, your UI might not look as good and might need adjustments for different screen sizes. For example, on a larger screen, you might want to adjust the position and size of some elements to take advantage of the additional screen space, or on a smaller screen, you might need to adjust sizes so that everything can fit on the screen.

The configuration qualifiers you can use to provide size-specific resources are `small`, `normal`, `large`, and `xlarge`. For example, layouts for an extra large screen should go in `layout-xlarge/`.

Beginning with Android 3.2 (API level 13), the above size groups are deprecated and you should instead use the `sw<N>dp` configuration qualifier to define the smallest available width required by your layout resources. For example, if your multi-pane tablet layout requires at least 600dp of screen width, you should place it in `layout-sw600dp/`. Using the new techniques for declaring layout resources is discussed further in the section about [Declaring Tablet Layouts for Android 3.2](#).

- **Provide different bitmap drawables for different screen densities**

By default, Android scales your bitmap drawables (`.png`, `.jpg`, and `.gif` files) and Nine-Patch drawables (`.9.png` files) so that they render at the appropriate physical size on each device. For example, if your application provides bitmap drawables only for the baseline, medium screen density (`mdpi`), then the system scales them up when on a high-density screen, and scales them down when on a low-density screen. This scaling can cause artifacts in the bitmaps. To ensure your bitmaps look their best, you should include alternative versions at different resolutions for different screen densities.

The configuration qualifiers you can use for density-specific resources are `ldpi` (low), `mdpi` (medium), `hdpi` (high), and `xhdpi` (extra high). For example, bitmaps for high-density screens should go in `drawable-hdpi/`.

The size and density configuration qualifiers correspond to the generalized sizes and densities described in [Range of screens supported](#), above.

**Note:** If you're not familiar with configuration qualifiers and how the system uses them to apply alternative resources, read [Providing Alternative Resources](#) for more information.

At runtime, the system ensures the best possible display on the current screen with the following procedure for any given resource:

1. The system uses the appropriate alternative resource

Based on the size and density of the current screen, the system uses any size- and density-specific resource provided in your application. For example, if the device has a high-density screen and the application requests a drawable resource, the system looks for a drawable resource directory that best matches the device configuration. Depending on the other alternative resources available, a resource directory with the `hdpi` qualifier (such as `drawable-hdpi/`) might be the best match, so the system uses the drawable resource from this directory.

2. If no matching resource is available, the system uses the default resource and scales it up or down as needed to match the current screen size and density

The "default" resources are those that are not tagged with a configuration qualifier. For example, the resources in `drawable/` are the default drawable resources. The system assumes that default resources are designed for the baseline screen size and density, which is a normal screen size and a medium density. As such, the system scales default density resources up for high-density screens and down for low-density screens, as appropriate.

However, when the system is looking for a density-specific resource and does not find it in the density-specific directory, it won't always use the default resources. The system may instead use one of the other density-specific resources in order to provide better results when scaling. For example, when looking for a low-density resource and it is not available, the system prefers to scale-down the high-density version of the resource, because the system can easily scale a high-density resource down to low-density by a factor of 0.5, with fewer artifacts, compared to scaling a medium-density resource by a factor of 0.75.

For more information about how Android selects alternative resources by matching configuration qualifiers to the device configuration, read [How Android Finds the Best-matching Resource](#).

## Using configuration qualifiers

Android supports several configuration qualifiers that allow you to control how the system selects your alternative resources based on the characteristics of the current device screen. A configuration qualifier is a string that you can append to a resource directory in your Android project and specifies the configuration for which the resources inside are designed.

To use a configuration qualifier:

1. Create a new directory in your project's `res/` directory and name it using the format:

```
<resources_name>-<qualifier>
  • <resources_name> is the standard resource name (such as drawable or layout).
  • <qualifier> is a configuration qualifier from table 1, below, specifying the screen configuration for which these resources are to be used (such as hdpi or xlarge).
```

You can use more than one `<qualifier>` at a time—simply separate each qualifier with a dash.

2. Save the appropriate configuration-specific resources in this new directory. The resource files must be named exactly the same as the default resource files.

For example, `xlarge` is a configuration qualifier for extra large screens. When you append this string to a resource directory name (such as `layout-xlarge`), it indicates to the system that these resources are to be used on devices that have an extra large screen.

**Table 1.** Configuration qualifiers that allow you to provide special resources for different screen configurations.

Screen characteristic	Qualifier	Description
Size	small	Resources for <i>small</i> size screens.
	normal	Resources for <i>normal</i> size screens. (This is the baseline size.)
	large	Resources for <i>large</i> size screens.
	xlarge	Resources for <i>extra large</i> size screens.
Density	ldpi	Resources for low-density ( <i>ldpi</i> ) screens (~120dpi).
	mdpi	Resources for medium-density ( <i>mdpi</i> ) screens (~160dpi). (This is the baseline density)
	hdpi	Resources for high-density ( <i>hdpi</i> ) screens (~240dpi).

	xhdpi	Resources for extra high-density ( <i>xhdpi</i> ) screens (~320dpi).
	nodpi	Resources for all densities. These are density-independent resources. The system does not scale resources tagged with this qualifier, regardless of the current screen's density.
	tvdpi	Resources for screens somewhere between mdpi and hdpi; approximately 213dpi. This is not considered a "primary" density group. It is mostly intended for televisions and most apps shouldn't need it—providing mdpi and hdpi resources is sufficient for most apps and the system will scale them as appropriate. If you find it necessary to provide tvdpi resources, you should size them at a factor of 1.33*mdpi. For example, a 100px x 100px image for mdpi screens should be 133px x 133px for tvdpi.
Orientation	land	Resources for screens in the landscape orientation (wide aspect ratio).
	port	Resources for screens in the portrait orientation (tall aspect ratio).
Aspect ratio	long	Resources for screens that have a significantly taller or wider aspect ratio (when in portrait or landscape orientation, respectively) than the baseline screen configuration.
	notlong	Resources for use screens that have an aspect ratio that is similar to the baseline screen configuration.

**Note:** If you're developing your application for Android 3.2 and higher, see the section about [Declaring Tablet Layouts for Android 3.2](#) for information about new configuration qualifiers that you should use when declaring layout resources for specific screen sizes (instead of using the size qualifiers in table 1).

For more information about how these qualifiers roughly correspond to real screen sizes and densities, see [Range of Screens Supported](#), earlier in this document.

For example, the following is a list of resource directories in an application that provides different layout designs for different screen sizes and different bitmap drawables for medium, high, and extra high density screens.

```
res/layout/my_layout.xml                      // layout for normal screen size ("default")
res/layout-small/my_layout.xml                 // layout for small screen size
res/layout-large/my_layout.xml                // layout for large screen size
res/layout-xlarge/my_layout.xml               // layout for extra large screen size
res/layout-xlarge-land/my_layout.xml          // layout for extra large in landscape orientation

res/drawable-mdpi/my_icon.png                // bitmap for medium density
res/drawable-hdpi/my_icon.png                // bitmap for high density
res/drawable-xhdpi/my_icon.png               // bitmap for extra high density
```

For more information about how to use alternative resources and a complete list of configuration qualifiers (not just for screen configurations), see [Providing Alternative Resources](#).

Be aware that, when the Android system picks which resources to use at runtime, it uses certain logic to determine the "best matching" resources. That is, the qualifiers you use don't have to exactly match the current screen configuration in all cases in order for the system to use them. Specifically, when selecting resources based on the size qualifiers, the system will use resources designed for a screen smaller than the current screen if there are no resources that better match (for example, a large-size screen will use normal-size screen resources if necessary). However, if the only available resources are *larger* than the current screen, the system will not use them and your application will crash if no other resources match the device configuration (for example, if all layout resources are tagged with the *xlarge* qualifier, but the device is a normal-size screen). For more information about how the system selects resources, read [How Android Finds the Best-matching Resource](#).

**Tip:** If you have some drawable resources that the system should never scale (perhaps because you perform some adjustments to the image yourself at runtime), you should place them in a directory with the *nodpi* con-

figuration qualifier. Resources with this qualifier are considered density-agnostic and the system will not scale them.

## Designing alternative layouts and drawables

The types of alternative resources you should create depends on your application's needs. Usually, you should use the size and orientation qualifiers to provide alternative layout resources and use the density qualifiers to provide alternative bitmap drawable resources.

The following sections summarize how you might want to use the size and density qualifiers to provide alternative layouts and drawables, respectively.

### Alternative layouts

Generally, you'll know whether you need alternative layouts for different screen sizes once you test your application on different screen configurations. For example:

- When testing on a small screen, you might discover that your layout doesn't quite fit on the screen. For example, a row of buttons might not fit within the width of the screen on a small screen device. In this case you should provide an alternative layout for small screens that adjusts the size or position of the buttons.
- When testing on an extra large screen, you might realize that your layout doesn't make efficient use of the big screen and is obviously stretched to fill it. In this case, you should provide an alternative layout for extra large screens that provides a redesigned UI that is optimized for bigger screens such as tablets.

Although your application should work fine without an alternative layout on big screens, it's quite important to users that your application looks as though it's designed specifically for their devices. If the UI is obviously stretched, users are more likely to be unsatisfied with the application experience.

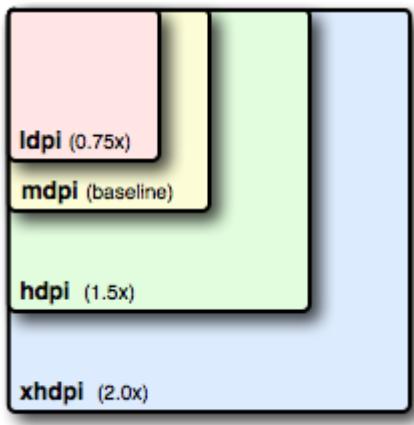
- And, when testing in the landscape orientation compared to the portrait orientation, you might notice that UI elements placed at the bottom of the screen for the portrait orientation should instead be on the right side of the screen in landscape orientation.

To summarize, you should be sure that your application layout:

- Fits on small screens (so users can actually use your application)
- Is optimized for bigger screens to take advantage of the additional screen space
- Is optimized for both landscape and portrait orientations

If your UI uses bitmaps that need to fit the size of a view even after the system scales the layout (such as the background image for a button), you should use [Nine-Patch](#) bitmap files. A Nine-Patch file is basically a PNG file in which you specific two-dimensional regions that are stretchable. When the system needs to scale the view in which the bitmap is used, the system stretches the Nine-Patch bitmap, but stretches only the specified regions. As such, you don't need to provide different drawables for different screen sizes, because the Nine-Patch bitmap can adjust to any size. You should, however, provide alternate versions of your Nine-Patch files for different screen densities.

## Alternative drawables



**Figure 4.** Relative sizes for bitmap drawables that support each density.

Almost every application should have alternative drawable resources for different screen densities, because almost every application has a launcher icon and that icon should look good on all screen densities. Likewise, if you include other bitmap drawables in your application (such as for menu icons or other graphics in your application), you should provide alternative versions of each one, for different densities.

**Note:** You only need to provide density-specific drawables for bitmap files (`.png`, `.jpg`, or `.gif`) and Nine-Path files (`.9.png`). If you use XML files to define shapes, colors, or other [drawable resources](#), you should put one copy in the default drawable directory (`drawable/`).

To create alternative bitmap drawables for different densities, you should follow the **3:4:6:8 scaling ratio** between the four generalized densities. For example, if you have a bitmap drawable that's 48x48 pixels for medium-density screen (the size for a launcher icon), all the different sizes should be:

- 36x36 for low-density
- 48x48 for medium-density
- 72x72 for high-density
- 96x96 for extra high-density

For more information about designing icons, see the [Icon Design Guidelines](#), which includes size information for various bitmap drawables, such as launcher icons, menu icons, status bar icons, tab icons, and more.

## Declaring Tablet Layouts for Android 3.2

For the first generation of tablets running Android 3.0, the proper way to declare tablet layouts was to put them in a directory with the `xlarge` configuration qualifier (for example, `res/layout-xlarge/`). In order to accommodate other types of tablets and screen sizes—in particular, 7" tablets—Android 3.2 introduces a new way to specify resources for more discrete screen sizes. The new technique is based on the amount of space your layout needs (such as 600dp of width), rather than trying to make your layout fit the generalized size groups (such as `large` or `xlarge`).

The reason designing for 7" tablets is tricky when using the generalized size groups is that a 7" tablet is technically in the same group as a 5" handset (the `large` group). While these two devices are seemingly close to each other in size, the amount of space for an application's UI is significantly different, as is the style of user interaction. Thus, a 7" and 5" screen should not always use the same layout. To make it possible for you to provide different layouts for these two kinds of screens, Android now allows you to specify your layout resources based on the width and/or height that's actually available for your application's layout, specified in dp units.

For example, after you've designed the layout you want to use for tablet-style devices, you might determine that the layout stops working well when the screen is less than 600dp wide. This threshold thus becomes the minimum size that you require for your tablet layout. As such, you can now specify that these layout resources should be used only when there is at least 600dp of width available for your application's UI.

You should either pick a width and design to it as your minimum size, or test what is the smallest width your layout supports once it's complete.

**Note:** Remember that all the figures used with these new size APIs are density-independent pixel (dp) values and your layout dimensions should also always be defined using dp units, because what you care about is the amount of screen space available after the system accounts for screen density (as opposed to using raw pixel resolution). For more information about density-independent pixels, read [Terms and concepts](#), earlier in this document.

## Using new size qualifiers

The different resource configurations that you can specify based on the space available for your layout are summarized in table 2. These new qualifiers offer you more control over the specific screen sizes your application supports, compared to the traditional screen size groups (small, normal, large, and xlarge).

**Note:** The sizes that you specify using these qualifiers are **not the actual screen sizes**. Rather, the sizes are for the width or height in dp units that are **available to your activity's window**. The Android system might use some of the screen for system UI (such as the system bar at the bottom of the screen or the status bar at the top), so some of the screen might not be available for your layout. Thus, the sizes you declare should be specifically about the sizes needed by your activity—the system accounts for any space used by system UI when declaring how much space it provides for your layout. Also beware that the [Action Bar](#) is considered a part of your application's window space, although your layout does not declare it, so it reduces the space available for your layout and you must account for it in your design.

**Table 2.** New configuration qualifiers for screen size (introduced in Android 3.2).

Screen configuration	Qualifier values	Description
smallestWidth Examples:	sw<N>dp sw600dp sw720dp	The fundamental size of a screen, as indicated by the shortest dimension of the available screen area. Specifically, the device's smallestWidth is the shortest of the screen's available height and width (you may also think of it as the "smallest possible width" for the screen). You can use this qualifier to ensure that, regardless of the screen's current orientation, your application's has at least <N> dps of width available for its UI.  For example, if your layout requires that its smallest dimension of screen area be at least 600 dp at all times, then you can use this qualifier to create the layout resources, res/layout-sw600dp/. The system will use these resources only when the smallest dimension of available screen is at least 600dp, regardless of whether the 600dp side is the user-perceived height or width. The smallestWidth is a fixed screen size characteristic of the device; <b>the device's smallestWidth does not change when the screen's orientation changes</b> .
		The smallestWidth of a device takes into account screen decorations and system UI. For example, if the device has some persistent UI elements on the screen that account for space along the axis of the smallestWidth, the system declares the smallestWidth

to be smaller than the actual screen size, because those are screen pixels not available for your UI.

This is an alternative to the generalized screen size qualifiers (small, normal, large, xlarge) that allows you to define a discrete number for the effective size available for your UI. Using `smallestWidth` to determine the general screen size is useful because width is often the driving factor in designing a layout. A UI will often scroll vertically, but have fairly hard constraints on the minimum space it needs horizontally. The available width is also the key factor in determining whether to use a one-pane layout for handsets or multi-pane layout for tablets. Thus, you likely care most about what the smallest possible width will be on each device.

		Specifies a minimum available width in dp units at which the resources should be used—defined by the <N> value. The system's corresponding value for the width changes when the screen's orientation switches between landscape and portrait to reflect the current actual width that's available for your UI.
Available screen width	w<N>dp Examples: w720dp w1024dp	This is often useful to determine whether to use a multi-pane layout, because even on a tablet device, you often won't want the same multi-pane layout for portrait orientation as you do for landscape. Thus, you can use this to specify the minimum width required for the layout, instead of using both the screen size and orientation qualifiers together.
Available screen height	h<N>dp Examples: h720dp h1024dp etc.	Specifies a minimum screen height in dp units at which the resources should be used—defined by the <N> value. The system's corresponding value for the height changes when the screen's orientation switches between landscape and portrait to reflect the current actual height that's available for your UI.  Using this to define the height required by your layout is useful in the same way as <code>w&lt;N&gt;dp</code> is for defining the required width, instead of using both the screen size and orientation qualifiers. However, most apps won't need this qualifier, considering that UIs often scroll vertically and are thus more flexible with how much height is available, whereas the width is more rigid.

While using these qualifiers might seem more complicated than using screen size groups, it should actually be simpler once you determine the requirements for your UI. When you design your UI, the main thing you probably care about is the actual size at which your application switches between a handset-style UI and a tablet-style UI that uses multiple panes. The exact point of this switch will depend on your particular design—maybe you need a 720dp width for your tablet layout, maybe 600dp is enough, or 480dp, or some number between these. Using these qualifiers in table 2, you are in control of the precise size at which your layout changes.

For more discussion about these size configuration qualifiers, see the [Providing Resources](#) document.

## Configuration examples

To help you target some of your designs for different types of devices, here are some numbers for typical screen widths:

- 320dp: a typical phone screen (240x320 ldpi, 320x480 mdpi, 480x800 hdpi, etc).
- 480dp: a tweener tablet like the Streak (480x800 mdpi).

- 600dp: a 7" tablet (600x1024 mdpi).
- 720dp: a 10" tablet (720x1280 mdpi, 800x1280 mdpi, etc).

Using the size qualifiers from table 2, your application can switch between your different layout resources for handsets and tablets using any number you want for width and/or height. For example, if 600dp is the smallest available width supported by your tablet layout, you can provide these two sets of layouts:

```
res/layout/main_activity.xml          # For handsets
res/layout-sw600dp/main_activity.xml # For tablets
```

In this case, the smallest width of the available screen space must be 600dp in order for the tablet layout to be applied.

For other cases in which you want to further customize your UI to differentiate between sizes such as 7" and 10" tablets, you can define additional smallest width layouts:

```
res/layout/main_activity.xml          # For handsets (smaller than 600dp available)
res/layout-sw600dp/main_activity.xml  # For 7" tablets (600dp wide and bigger)
res/layout-sw720dp/main_activity.xml  # For 10" tablets (720dp wide and bigger)
```

Notice that the previous two sets of example resources use the "smallest width" qualifer, `sw<N>dp`, which specifies the smallest of the screen's two sides, regardless of the device's current orientation. Thus, using `sw<N>dp` is a simple way to specify the overall screen size available for your layout by ignoring the screen's orientation.

However, in some cases, what might be important for your layout is exactly how much width or height is *currently* available. For example, if you have a two-pane layout with two fragments side by side, you might want to use it whenever the screen provides at least 600dp of width, whether the device is in landscape or portrait orientation. In this case, your resources might look like this:

```
res/layout/main_activity.xml          # For handsets (smaller than 600dp available)
res/layout-w600dp/main_activity.xml   # Multi-pane (any screen with 600dp available)
```

Notice that the second set is using the "available width" qualifier, `w<N>dp`. This way, one device may actually use both layouts, depending on the orientation of the screen (if the available width is at least 600dp in one orientation and less than 600dp in the other orientation).

If the available height is a concern for you, then you can do the same using the `h<N>dp` qualifier. Or, even combine the `w<N>dp` and `h<N>dp` qualifiers if you need to be really specific.

## Declaring screen size support

Once you've implemented your layouts for different screen sizes, it's equally important that you declare in your manifest file which screens your application supports.

Along with the new configuration qualifiers for screen size, Android 3.2 introduces new attributes for the [`<supports-screens>`](#) manifest element:

### [`android:requiresSmallestWidthDp`](#)

Specifies the minimum `smallestWidth` required. The `smallestWidth` is the shortest dimension of the screen space (in dp units) that must be available to your application UI—that is, the shortest of the available screen's two dimensions. So, in order for a device to be considered compatible with your application, the device's `smallestWidth` must be equal to or greater than this value. (Usually, the value you supply for this is the "smallest width" that your layout supports, regardless of the screen's current orientation.)

For example, if your application is only for tablet-style devices with a 600dp smallest available width:

```
<manifest ... >
    <supports-screens android:requiresSmallestWidthDp="600" />
    ...
</manifest>
```

However, if your application supports all screen sizes supported by Android (as small as 426dp x 320dp), then you don't need to declare this attribute, because the smallest width your application requires is the smallest possible on any device.

**Caution:** The Android system does not pay attention to this attribute, so it does not affect how your application behaves at runtime. Instead, it is used to enable filtering for your application on services such as Google Play. However, **Google Play currently does not support this attribute for filtering** (on Android 3.2), so you should continue using the other size attributes if your application does not support small screens.

#### [android:compatibleWidthLimitDp](#)

This attribute allows you to enable [screen compatibility mode](#) as a user-optional feature by specifying the maximum "smallest width" that your application supports. If the smallest side of a device's available screen is greater than your value here, users can still install your application, but are offered to run it in screen compatibility mode. By default, screen compatibility mode is disabled and your layout is resized to fit the screen as usual, but a button is available in the system bar that allows users to toggle screen compatibility mode on and off.

**Note:** If your application's layout properly resizes for large screens, you do not need to use this attribute. We recommend that you avoid using this attribute and instead ensure your layout resizes for larger screens by following the recommendations in this document.

#### [android:largestWidthLimitDp](#)

This attribute allows you to force-enable [screen compatibility mode](#) by specifying the maximum "smallest width" that your application supports. If the smallest side of a device's available screen is greater than your value here, the application runs in screen compatibility mode with no way for the user to disable it.

**Note:** If your application's layout properly resizes for large screens, you do not need to use this attribute. We recommend that you avoid using this attribute and instead ensure your layout resizes for larger screens by following the recommendations in this document.

**Caution:** When developing for Android 3.2 and higher, you should not use the older screen size attributes in combination with the attributes listed above. Using both the new attributes and the older size attributes might cause unexpected behavior.

For more information about each of these attributes, follow the respective links above.

## Best Practices

The objective of supporting multiple screens is to create an application that can function properly and look good on any of the generalized screen configurations supported by Android. The previous sections of this document provide information about how Android adapts your application to screen configurations and how you can customize the look of your application on different screen configurations. This section provides some additional tips and an overview of techniques that help ensure that your application scales properly for different screen configurations.

Here is a quick checklist about how you can ensure that your application displays properly on different screens:

1. Use `wrap_content`, `fill_parent`, or `dp` units when specifying dimensions in an XML layout file
2. Do not use hard coded pixel values in your application code
3. Do not use `AbsoluteLayout` (it's deprecated)
4. Supply alternative bitmap drawables for different screen densities

The following sections provide more details.

## 1. Use `wrap_content`, `fill_parent`, or the `dp` unit for layout dimensions

When defining the `android:layout_width` and `android:layout_height` for views in an XML layout file, using "`wrap_content`", "`fill_parent`" or `dp` units guarantees that the view is given an appropriate size on the current device screen.

For instance, a view with a `layout_width="100dp"` measures 100 pixels wide on medium-density screen and the system scales it up to 150 pixels wide on high-density screen, so that the view occupies approximately the same physical space on the screen.

Similarly, you should prefer the `sp` (scale-independent pixel) to define text sizes. The `sp` scale factor depends on a user setting and the system scales the size the same as it does for `dp`.

## 2. Do not use hard-coded pixel values in your application code

For performance reasons and to keep the code simpler, the Android system uses pixels as the standard unit for expressing dimension or coordinate values. That means that the dimensions of a view are always expressed in the code using pixels, but always based on the current screen density. For instance, if `myView.getWidth()` returns 10, the view is 10 pixels wide on the current screen, but on a device with a higher density screen, the value returned might be 15. If you use pixel values in your application code to work with bitmaps that are not pre-scaled for the current screen density, you might need to scale the pixel values that you use in your code to match the un-scaled bitmap source.

If your application manipulates bitmaps or deals with pixel values at runtime, see the section below about [Additional Density Considerations](#).

## 3. Do not use `AbsoluteLayout`

Unlike the other layouts widgets, `AbsoluteLayout` enforces the use of fixed positions to lay out its child views, which can easily lead to user interfaces that do not work well on different displays. Because of this, [AbsoluteLayout](#) was deprecated in Android 1.5 (API Level 3).

You should instead use `RelativeLayout`, which uses relative positioning to lay out its child views. For instance, you can specify that a button widget should appear "to the right of" a text widget.

## 4. Use size and density-specific resources

Although the system scales your layout and drawable resources based on the current screen configuration, you may want to make adjustments to the UI on different screen sizes and provide bitmap drawables that are optimized for different densities. This essentially reiterates the information from earlier in this document.

If you need to control exactly how your application will look on various screen configurations, adjust your layouts and bitmap drawables in configuration-specific resource directories. For example, consider an icon that you want to display on medium and high density screens. Simply create your icon at two different sizes (for in-

stance 100x100 for medium density and 150x150 for high density) and put the two variations in the appropriate directories, using the proper qualifiers:

```
res/drawable-mdpi/icon.png      //for medium-density screens  
res/drawable-hdpi/icon.png     //for high-density screens
```

**Note:** If a density qualifier is not defined in a directory name, the system assumes that the resources in that directory are designed for the baseline medium density and will scale for other densities as appropriate.

For more information about valid configuration qualifiers, see [Using configuration qualifiers](#), earlier in this document.

## Additional Density Considerations

This section describes more about how Android performs scaling for bitmap drawables on different screen densities and how you can further control how bitmaps are drawn on different densities. The information in this section shouldn't be important to most applications, unless you have encountered problems in your application when running on different screen densities or your application manipulates graphics.

To better understand how you can support multiple densities when manipulating graphics at runtime, you should understand that the system helps ensure the proper scale for bitmaps in the following ways:

### 1. *Pre-scaling of resources (such as bitmap drawables)*

Based on the density of the current screen, the system uses any size- or density-specific resources from your application and displays them without scaling. If resources are not available in the correct density, the system loads the default resources and scales them up or down as needed to match the current screen's density. The system assumes that default resources (those from a directory without configuration qualifiers) are designed for the baseline screen density (mdpi), unless they are loaded from a density-specific resource directory. Pre-scaling is, thus, what the system does when resizing a bitmap to the appropriate size for the current screen density.

If you request the dimensions of a pre-scaled resource, the system returns values representing the dimensions *after* scaling. For example, a bitmap designed at 50x50 pixels for an mdpi screen is scaled to 75x75 pixels on an hdpi screen (if there is no alternative resource for hdpi) and the system reports the size as such.

There are some situations in which you might not want Android to pre-scale a resource. The easiest way to avoid pre-scaling is to put the resource in a resource directory with the `nodpi` configuration qualifier. For example:

```
res/drawable-nodpi/icon.png
```

When the system uses the `icon.png` bitmap from this folder, it does not scale it based on the current device density.

### 2. *Auto-scaling of pixel dimensions and coordinates*

An application can disable pre-scaling by setting `android:anyDensity` to "false" in the manifest or programmatically for a [Bitmap](#) by setting `inScaled` to "false". In this case, the system auto-scales any absolute pixel coordinates and pixel dimension values at draw time. It does this to ensure that pixel-defined screen elements are still displayed at approximately the same physical size as they would be at the baseline screen density (mdpi). The system handles this scaling transparently to

the application and reports the scaled pixel dimensions to the application, rather than physical pixel dimensions.

For instance, suppose a device has a WVGA high-density screen, which is 480x800 and about the same size as a traditional HVGA screen, but it's running an application that has disabled pre-scaling. In this case, the system will "lie" to the application when it queries for screen dimensions, and report 320x533 (the approximate mdpi translation for the screen density). Then, when the application does drawing operations, such as invalidating the rectangle from (10,10) to (100, 100), the system transforms the coordinates by scaling them the appropriate amount, and actually invalidate the region (15,15) to (150, 150). This discrepancy may cause unexpected behavior if your application directly manipulates the scaled bitmap, but this is considered a reasonable trade-off to keep the performance of applications as good as possible. If you encounter this situation, read the following section about [Converting dp units to pixel units](#).

Usually, **you should not disable pre-scaling**. The best way to support multiple screens is to follow the basic techniques described above in [How to Support Multiple Screens](#).

If your application manipulates bitmaps or directly interacts with pixels on the screen in some other way, you might need to take additional steps to support different screen densities. For example, if you respond to touch gestures by counting the number of pixels that a finger crosses, you need to use the appropriate density-independent pixel values, instead of actual pixels.

## Scaling Bitmap objects created at runtime



**Figure 5.** Comparison of pre-scaled and auto-scaled bitmaps, from [ApiDemos](#).

If your application creates an in-memory bitmap (a [Bitmap](#) object), the system assumes that the bitmap is designed for the baseline medium-density screen, by default, and auto-scales the bitmap at draw time. The system applies "auto-scaling" to a [Bitmap](#) when the bitmap has unspecified density properties. If you don't properly account for the current device's screen density and specify the bitmap's density properties, the auto-scaling can result in scaling artifacts the same as when you don't provide alternative resources.

To control whether a [Bitmap](#) created at runtime is scaled or not, you can specify the density of the bitmap with [setDensity\(\)](#), passing a density constant from [DisplayMetrics](#), such as [DENSITY\\_HIGH](#) or [DENSITY\\_LOW](#).

If you're creating a [Bitmap](#) using [BitmapFactory](#), such as from a file or a stream, you can use [BitmapFactory.Options](#) to define properties of the bitmap as it already exists, which determine if or how the system will scale it. For example, you can use the [inDensity](#) field to define the density for which the bitmap is designed and the [inScaled](#) field to specify whether the bitmap should scale to match the current device's screen density.

If you set the [inScaled](#) field to `false`, then you disable any pre-scaling that the system may apply to the bitmap and the system will then auto-scale it at draw time. Using auto-scaling instead of pre-scaling can be more CPU expensive, but uses less memory.

Figure 5 demonstrates the results of the pre-scale and auto-scale mechanisms when loading low (120), medium (160) and high (240) density bitmaps on a high-density screen. The differences are subtle, because all of the bitmaps are being scaled to match the current screen density, however the scaled bitmaps have slightly different appearances depending on whether they are pre-scaled or auto-scaled at draw time. You can find the source code for this sample application, which demonstrates using pre-scaled and auto-scaled bitmaps, in [ApiDemos](#).

**Note:** In Android 3.0 and above, there should be no perceivable difference between pre-scaled and auto-scaled bitmaps, due to improvements in the graphics framework.

## Converting dp units to pixel units

In some cases, you will need to express dimensions in dp and then convert them to pixels. Imagine an application in which a scroll or fling gesture is recognized after the user's finger has moved by at least 16 pixels. On a baseline screen, a user's must move by 16 pixels / 160 dpi, which equals 1/10th of an inch (or 2.5 mm) before the gesture is recognized. On a device with a high density display (240dpi), the user's must move by 16 pixels / 240 dpi, which equals 1/15th of an inch (or 1.7 mm). The distance is much shorter and the application thus appears more sensitive to the user.

To fix this issue, the gesture threshold must be expressed in code in dp and then converted to actual pixels. For example:

```
// The gesture threshold expressed in dp
private static final float GESTURE_THRESHOLD_DP = 16.0f;

// Get the screen's density scale
final float scale = getResources().getDisplayMetrics().density;
// Convert the dps to pixels, based on density scale
mGestureThreshold = (int) (GESTURE_THRESHOLD_DP * scale + 0.5f);

// Use mGestureThreshold as a distance in pixels...
```

The [DisplayMetrics.density](#) field specifies the scale factor you must use to convert dp units to pixels, according to the current screen density. On a medium-density screen, [DisplayMetrics.density](#) equals

1.0; on a high-density screen it equals 1.5; on an extra high-density screen, it equals 2.0; and on a low-density screen, it equals 0.75. This figure is the factor by which you should multiply the dp units on order to get the actual pixel count for the current screen. (Then add 0.5f to round the figure up to the nearest whole number, when converting to an integer.) For more information, refer to the [DisplayMetrics](#) class.

However, instead of defining an arbitrary threshold for this kind of event, you should use pre-scaled configuration values that are available from [ViewConfiguration](#).

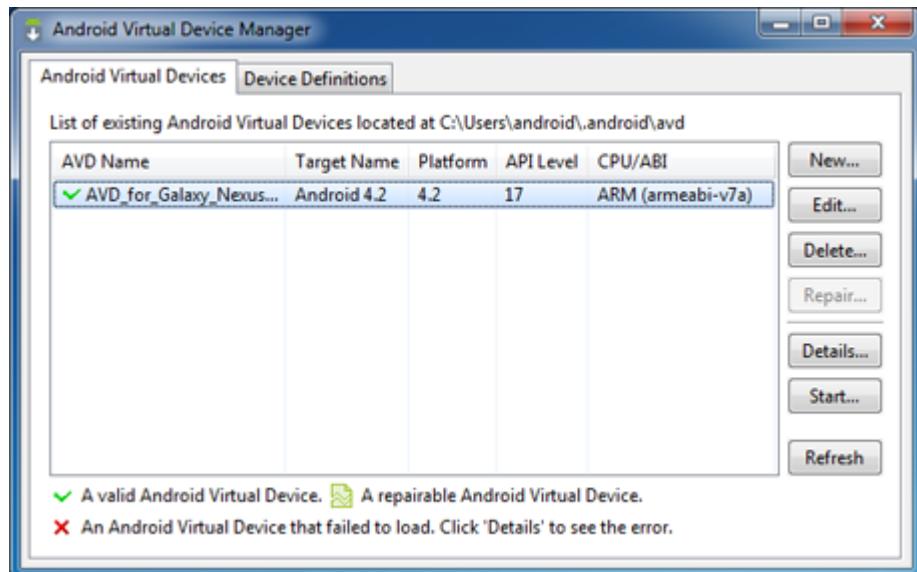
## Using pre-scaled configuration values

You can use the [ViewConfiguration](#) class to access common distances, speeds, and times used by the Android system. For instance, the distance in pixels used by the framework as the scroll threshold can be obtained with [getScaledTouchSlop\(\)](#):

```
private static final int GESTURE_THRESHOLD_DP = ViewConfiguration.get(myContext)
```

Methods in [ViewConfiguration](#) starting with the `getScaled` prefix are guaranteed to return a value in pixels that will display properly regardless of the current screen density.

## How to Test Your Application on Multiple Screens



**Figure 6.** A set of AVDs for testing screens support.

Before publishing your application, you should thoroughly test it in all of the supported screen sizes and densities. The Android SDK includes emulator skins you can use, which replicate the sizes and densities of common screen configurations on which your application is likely to run. You can also modify the default size, density, and resolution of the emulator skins to replicate the characteristics of any specific screen. Using the emulator skins and additional custom configurations allows you to test any possible screen configuration, so you don't have to buy various devices just to test your application's screen support.

To set up an environment for testing your application's screen support, you should create a series of AVDs (Android Virtual Devices), using emulator skins and screen configurations that emulate the screen sizes and densities you want your application to support. To do so, you can use the AVD Manager to create the AVDs and launch them with a graphical interface.

To launch the Android SDK Manager, execute the `SDK Manager.exe` from your Android SDK directory (on Windows only) or execute `android` from the `<sdk>/tools/` directory (on all platforms). Figure 6 shows the AVD Manager with a selection of AVDs, for testing various screen configurations.

Table 3 shows the various emulator skins that are available in the Android SDK, which you can use to emulate some of the most common screen configurations.

For more information about creating and using AVDs to test your application, see [Managing AVDs with AVD Manager](#).

**Table 3.** Various screen configurations available from emulator skins in the Android SDK (indicated in bold) and other representative resolutions.

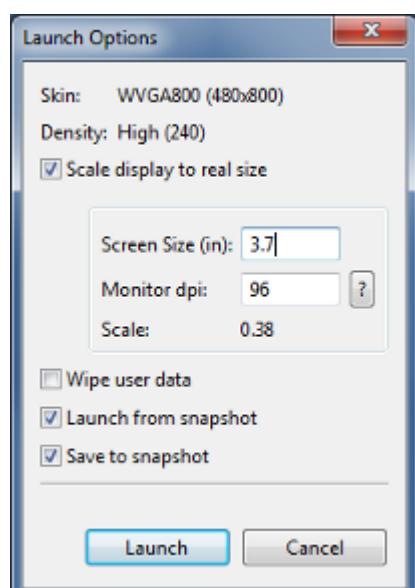
	<b>Low density (120), <i>ldpi</i></b>	<b>Medium density (160), <i>mdpi</i></b>	<b>High density (240), <i>hdpi</i></b>	<b>Extra high density (320), <i>xhdpi</i></b>
<b>Small screen</b>	<b>QVGA (240x320)</b>		480x640	
	<b>WQVGA400 (240x400)</b>		<b>WVGA800 (480x800)</b>	
<b>Normal screen</b>	<b>WQVGA432 (240x432)</b>	<b>HVGA (320x480)</b>	<b>WVGA854 (480x854)</b>	640x960
			600x1024	
	<b>WVGA800** (480x800)</b>	<b>WVGA800* (480x800)</b>		
<b>Large screen</b>	<b>WVGA854** (480x854)</b>	<b>WVGA854* (480x854)</b>		
		600x1024		
<b>Extra Large screen</b>	1024x600	<b>WXGA (1280x800)<sup>†</sup></b>	1536x1152	2048x1536
		1024x768	1920x1152	2560x1536
		1280x768	1920x1200	2560x1600

\* To emulate this configuration, specify a custom density of 160 when creating an AVD that uses a WVGA800 or WVGA854 skin.

\*\* To emulate this configuration, specify a custom density of 120 when creating an AVD that uses a WVGA800 or WVGA854 skin.

† This skin is available with the Android 3.0 platform

To see the relative numbers of active devices that support any given screen configuration, see the [Screen Sizes and Densities](#) dashboard.



**Figure 7.** Size and density options you can set, when starting an AVD from the AVD Manager.

We also recommend that you test your application in an emulator that is set up to run at a physical size that closely matches an actual device. This makes it a lot easier to compare the results at various sizes and densities. To do so you need to know the approximate density, in dpi, of your computer monitor (for instance, a 30" Dell monitor has a density of about 96 dpi). When you launch an AVD from the AVD Manager, you can specify the screen size for the emulator and your monitor dpi in the Launch Options, as shown in figure 7.

If you would like to test your application on a screen that uses a resolution or density not supported by the built-in skins, you can create an AVD that uses a custom resolution or density. When creating the AVD from the AVD Manager, specify the Resolution, instead of selecting a Built-in Skin.

If you are launching your AVD from the command line, you can specify the scale for the emulator with the `-scale` option. For example:

```
emulator -avd <avd_name> -scale 96dpi
```

To refine the size of the emulator, you can instead pass the `-scale` option a number between 0.1 and 3 that represents the desired scaling factor.

For more information about creating AVDs from the command line, see [Managing AVDs from the Command Line](#)

# Distributing to Specific Screens

## Quickview

- If necessary, you can control distribution of your application based on the device screen configuration

## In this document

1. [Declaring an App is Only for Handsets](#)
2. [Declaring an App is Only for Tablets](#)
3. [Publishing Multiple APKs for Different Screens](#)

## See also

1. [Supporting Multiple Screens](#)
2. [Optimizing Apps for Android 3.0](#)

Although we recommend that you design your application to function properly on multiple configurations of screen size and density, you can instead choose to limit the distribution of your application to certain types of screens, such as only tablets and other large devices or only handsets and similar-sized devices. To do so, you can enable filtering by external services such as Google Play by adding elements to your manifest file that specify the screen configurations your application supports.

However, before you decide to restrict your application to certain screen configurations, you should understand the techniques for [supporting multiple screens](#) and implement them to the best of your ability. By supporting multiple screens, your application can be made available to the greatest number of users with different devices, using a single APK.

## Declaring an App is Only for Handsets

Because the system generally scales applications to fit larger screens well, you shouldn't need to filter your application from larger screens. As long as you follow the [Best Practices for Screen Independence](#), your application should work well on larger screens such as tablets. However, you might discover that your application can't scale up well or perhaps you've decided to publish two versions of your application for different screen configurations. In such a case, you can use the `<compatible-screens>` element to manage the distribution of your application based on combinations of screen size and density. External services such as Google Play use this information to apply filtering to your application, so that only devices that have a screen configuration with which you declare compatibility can download your application.

The `<compatible-screens>` element must contain one or more `<screen>` elements. Each `<screen>` element specifies a screen configuration with which your application is compatible, using both the `android:screenSize` and `android:screenDensity` attributes. Each `<screen>` element **must include both attributes** to specify an individual screen configuration—if either attribute is missing, then the element is invalid (external services such as Google Play will ignore it).

For example, if your application is compatible with only small and normal size screens, regardless of screen density, you must specify eight different `<screen>` elements, because each screen size has four density configurations. You must declare each one of these; any combination of size and density that you do *not* specify is considered a screen configuration with which your application is *not* compatible. Here's what the manifest entry looks like if your application is compatible with only small and normal screen sizes:

```

<manifest ... >
    <compatible-screens>
        <!-- all small size screens -->
        <screen android:screenSize="small" android:screenDensity="ldpi" />
        <screen android:screenSize="small" android:screenDensity="mdpi" />
        <screen android:screenSize="small" android:screenDensity="hdpi" />
        <screen android:screenSize="small" android:screenDensity="xhdpi" />
        <!-- all normal size screens -->
        <screen android:screenSize="normal" android:screenDensity="ldpi" />
        <screen android:screenSize="normal" android:screenDensity="mdpi" />
        <screen android:screenSize="normal" android:screenDensity="hdpi" />
        <screen android:screenSize="normal" android:screenDensity="xhdpi" />
    </compatible-screens>
    ...
    <application ... >
        ...
    </application>
</manifest>

```

**Note:** Although you can also use the [`<compatible-screens>`](#) element for the reverse scenario (when your application is not compatible with smaller screens), it's easier if you instead use the [`<supports-screens>`](#) as discussed in the next section, because it doesn't require you to specify each screen density your application supports.

## Declaring an App is Only for Tablets

If you don't want your app to be used on handsets (perhaps your app truly makes sense only on a large screen) or you need time to optimize it for smaller screens, you can prevent small-screen devices from downloading your app by using the [`<supports-screens>`](#) manifest element.

For example, if you want your application to be available only to tablet devices, you can declare the element in your manifest like this:

```

<manifest ... >
    <supports-screens android:smallScreens="false"
                      android:normalScreens="false"
                      android:largeScreens="true"
                      android:xlargeScreens="true"
                      android:requiresSmallestWidthDp="600" />
    ...
    <application ... >
        ...
    </application>
</manifest>

```

This describes your app's screen-size support in two different ways:

- It declares that the app does *not* support the screen sizes "small" and "normal", which are traditionally not tablets.
- It declares that the app requires a screen size with a minimum usable area that is at least 600dp wide.

The first technique is for devices that are running Android 3.1 or older, because those devices declare their size based on generalized screen sizes. The [`requiresSmallestWidthDp`](#) attribute is for devices running Android 3.2 and newer, which includes the capability for apps to specify size requirements based on a minimum

number of density-independent pixels available. In this example, the app declares a minimum width requirement of 600dp, which generally implies a 7"-or-greater screen.

Your size choice might be different, of course, based on how well your design works on different screen sizes; for example, if your design works well only on screens that are 9" or larger, you might require a minimum width of 720dp.

The catch is that you must compile your application against Android 3.2 or higher in order to use the `requiresSmallestWidthDp` attribute. Older versions don't understand this attribute and will raise a compile-time error. The safest thing to do is develop your app against the platform that matches the API level you've set for [minSdkVersion](#). When you're making final preparations to build your release candidate, change the build target to Android 3.2 and add the `requiresSmallestWidthDp` attribute. Android versions older than 3.2 simply ignore that XML attribute, so there's no risk of a runtime failure.

For more information about why the "smallest width" screen size is important for supporting different screen sizes, read [New Tools for Managing Screen Sizes](#).

**Caution:** If you use the `<supports-screens>` element for the reverse scenario (when your application is not compatible with *larger* screens) and set the larger screen size attributes to "`false`", then external services such as Google Play **do not** apply filtering. Your application will still be available to larger screens, but when it runs, it will not resize to fit the screen. Instead, the system will emulate a handset screen size (about 320dp x 480dp; see [Screen Compatibility Mode](#) for more information). If you want to prevent your application from being downloaded on larger screens, use `<compatible-screens>`, as discussed in the previous section about [Declaring an App is Only for Handsets](#).

Remember, you should strive to make your application available to as many devices as possible by applying all necessary techniques for [supporting multiple screens](#). You should use `<compatible-screens>` or `<supports-screens>` only when you cannot provide compatibility on all screen configurations or you have decided to provide different versions of your application for different sets of screen configurations.

## Publishing Multiple APKs for Different Screens

Although we recommend that you publish one APK for your application, Google Play allows you to publish multiple APKs for the same application when each APK supports a different set of screen configurations (as declared in the manifest file). For example, if you want to publish both a handset version and a tablet version of your application, but you're unable to make the same APK work for both screen sizes, you can actually publish two APKs for the same application listing. Depending on each device's screen configuration, Google Play will deliver it the APK that you've declared to support that device's screen.

Beware, however, that publishing multiple APKs for the same application is considered an advanced feature and **most applications should publish only one APK that can support a wide range of device configurations**. Supporting multiple screen sizes, especially, is within reason using a single APK, as long as you follow the guide to [Supporting Multiple Screens](#).

If you need more information about how to publish multiple APKs on Google Play, read [Multiple APK Support](#).

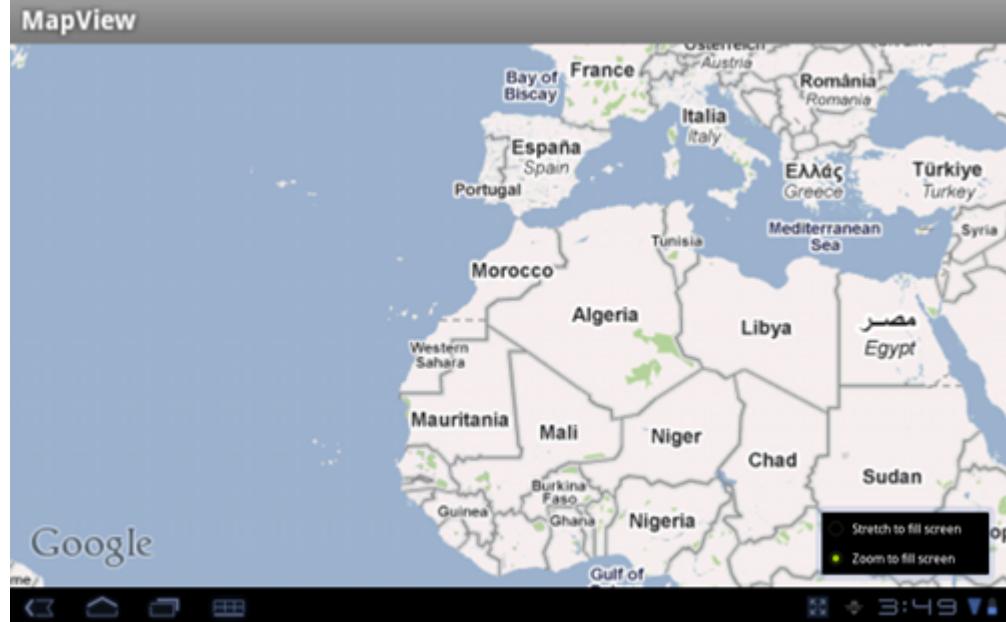
# Screen Compatibility Mode

## In this document

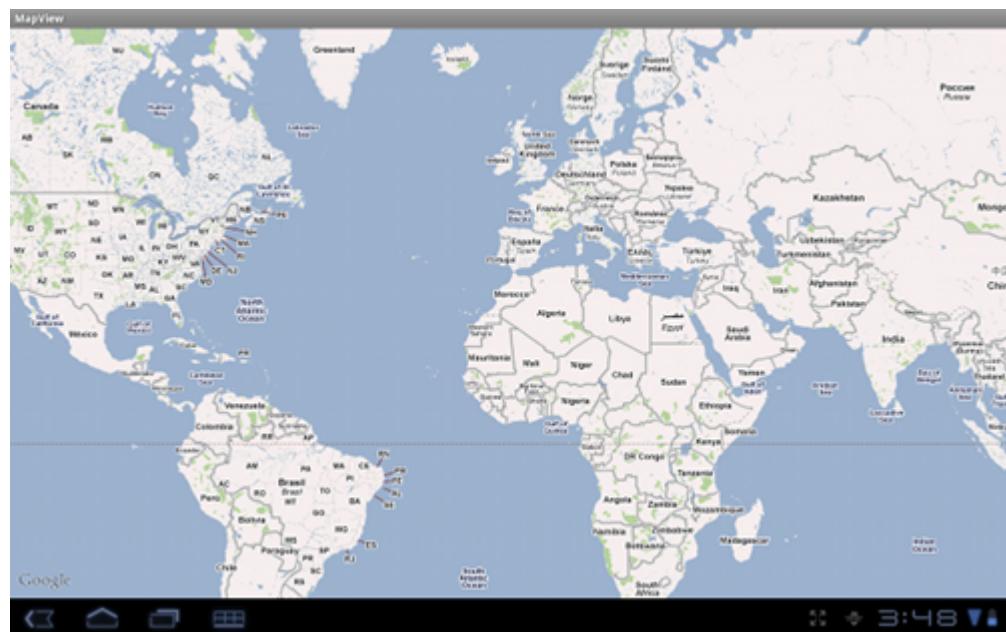
1. [Disabling Screen Compatibility Mode](#)
2. [Enabling Screen Compatibility Mode](#)

## See also

1. [Supporting Multiple Screens](#)
2. [`<supports-screens>`](#)



**Figure 1.** An application running in compatibility mode on an Android 3.2 tablet.



**Figure 2.** The same application from figure 1, with compatibility mode disabled.

**Notice:** If you've developed an application for a version of Android lower than Android 3.0, but it does resize properly for larger screens such as tablets, you should disable screen compatibility mode in order to maintain

the best user experience. To learn how to quickly disable the user option, jump to [Disabling Screen Compatibility Mode](#).

Screen compatibility mode is an escape hatch for applications that are not properly designed to resize for larger screens such as tablets. Since Android 1.6, Android has supported a variety of screen sizes and does most of the work to resize application layouts so that they properly fit each screen. However, if your application does not successfully follow the guide to [Supporting Multiple Screens](#), then it might encounter some rendering issues on larger screens. For applications with this problem, screen compatibility mode can make the application a little more usable on larger screens.

There are two versions of screen compatibility mode with slightly different behaviors:

### Version 1 (Android 1.6 - 3.1)

The system draws the application's UI in a "postage stamp" window. That is, the system draws the application's layout the same as it would on a normal size handset (emulating a 320dp x 480dp screen), with a black border that fills the remaining area of the screen.

This was introduced with Android 1.6 to handle apps that were designed only for the original screen size of 320dp x 480dp. Because there are so few active devices remaining that run Android 1.5, almost all applications should be developed against Android 1.6 or greater and should not have version 1 of screen compatibility mode enabled for larger screens. This version is considered obsolete.

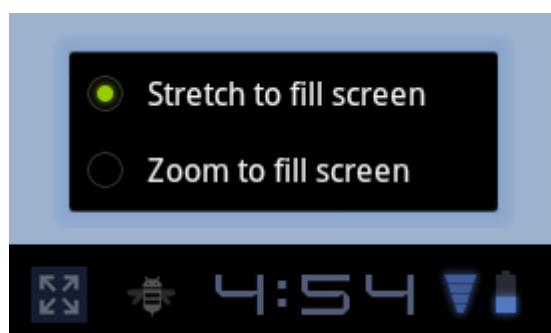
To disable this version of screen compatibility mode, you simply need to set [android:minSdkVersion](#) or [android:targetSdkVersion](#) to "4" or higher, or set [android:resizeable](#) to "true".

### Version 2 (Android 3.2 and greater)

The system draws the application's layout the same as it would on a normal size handset (approximately emulating a 320dp x 480dp screen), then scales it up to fill the screen. This essentially "zooms" in on your layout to make it bigger, which will usually cause artifacts such as blurring and pixelation in your UI.

This was introduced with Android 3.2 to further assist applications on the latest tablet devices when the applications have not yet implemented techniques for [Supporting Multiple Screens](#).

In general, large screen devices running Android 3.2 or higher allow users to enable screen compatibility mode when the application does not **explicitly declare that it supports large screens** in the manifest file. When this is the case, an icon (with outward-pointing arrows) appears next to the clock in the system bar, which allows the user to toggle screen compatibility mode on and off (figure 3). An application can also explicitly declare that it *does not* support large screens such that screen compatibility mode is always enabled and the user cannot disable it. (How to declare your application's support for large screens is discussed in the following sections.)



**Figure 3.** The pop up menu to toggle screen compatibility mode (currently disabled, so normal resizing occurs).

As a developer, you have control over when your application uses screen compatibility mode. The following sections describe how you can choose to disable or enable screen compatibility mode for larger screens when running Android 3.2 or higher.

## Disabling Screen Compatibility Mode

If you've developed your application primarily for versions of Android lower than 3.0, but **your application does resize properly** for larger screens such as tablets, **you should disable screen compatibility mode** in order to maintain the best user experience. Otherwise, users may enable screen compatibility mode and experience your application in a less-than-ideal format.

By default, screen compatibility mode for devices running Android 3.2 and higher is offered to users as an optional feature when one of the following is true:

- Your application has set both `android:minSdkVersion` and `android:targetSdkVersion` to "10" or lower and **does not explicitly declare support** for large screens using the `<supports-screens>` element.
- Your application has set either `android:minSdkVersion` or `android:targetSdkVersion` to "11" or higher and **explicitly declares that it does not support** large screens, using the `<supports-screens>` element.

To completely disable the user option for screen compatibility mode and remove the icon in the system bar, you can do one of the following:

- **Easiest:**

In your manifest file, add the `<supports-screens>` element and specify the `android:xlargeScreens` attribute to "true":

```
<supports-screens android:xlargeScreens="true" />
```

That's it. This declares that your application supports all larger screen sizes, so the system will always resize your layout to fit the screen. This works regardless of what values you've set in the `<uses-sdk>` attributes.

- **Easy but has other effects:**

In your manifest's `<uses-sdk>` element, set `android:targetSdkVersion` to "11" or higher:

```
<uses-sdk android:minSdkVersion="4" android:targetSdkVersion="11" />
```

This declares that your application supports Android 3.0 and, thus, is designed to work on larger screens such as tablets.

**Caution:** When running on Android 3.0 and greater, this also has the effect of enabling the Holo theme for your UI, adding the [Action Bar](#) to your activities, and removing the Options Menu button in the system bar.

If screen compatibility mode is still enabled after you change this, check your manifest's `<supports-screens>` and be sure that there are no attributes set "false". The best practice is to always explicitly declare your support for different screen sizes using the `<supports-screens>` element, so you should use this element anyway.

For more information about updating your application to target Android 3.0 devices, read [Optimizing Apps for Android 3.0](#).

## Enabling Screen Compatibility Mode

When your application is targeting Android 3.2 (API level 13) or higher, you can affect whether compatibility mode is enabled for certain screens by using the [`<supports-screens>`](#) element.

**Note:** Screen compatibility mode is **not** a mode in which you should want your application to run—it causes pixelation and blurring in your UI, due to zooming. The proper way to make your application work well on large screens is to follow the guide to [Supporting Multiple Screens](#) and provide alternative layouts for different screen sizes.

By default, when you've set either [`android:minSdkVersion`](#) or [`android:targetSdkVersion`](#) to "11" or higher, screen compatibility mode is **not** available to users. If either of these are true and your application does not resize properly for larger screens, you can choose to enable screen compatibility mode in one of the following ways:

- In your manifest file, add the [`<supports-screens>`](#) element and specify the [`an-droid:compatibleWidthLimitDp`](#) attribute to "320":

```
<supports-screens android:compatibleWidthLimitDp="320" />
```

This indicates that the maximum "smallest screen width" for which your application is designed is 320dp. This way, any devices with their smallest side being larger than this value will offer screen compatibility mode as a user-optional feature.

**Note:** Currently, screen compatibility mode only emulates handset screens with a 320dp width, so screen compatibility mode is not applied to any device if your value for [`an-droid:compatibleWidthLimitDp`](#) is larger than 320.

- If your application is functionally broken when resized for large screens and you want to force users into screen compatibility mode (rather than simply providing the option), you can use the [`an-droid:largestWidthLimitDp`](#) attribute:

```
<supports-screens android:largestWidthLimitDp="320" />
```

This works the same as [`android:compatibleWidthLimitDp`](#) except it force-enables screen compatibility mode and does not allow users to disable it.

# Supporting Tablets and Handsets

## In this document

1. [Basic Guidelines](#)
2. [Creating Single-pane and Multi-pane Layouts](#)
3. [Using the Action Bar](#)
  1. [Using split action bar](#)
  2. [Using "up" navigation](#)
4. [Other Design Tips](#)

## Related samples

1. [Honeycomb Gallery](#)

## See also

1. [Fragments](#)
2. [Action Bar](#)
3. [Supporting Multiple Screens](#)

The Android platform runs on a variety of screen sizes and the system gracefully resizes your application's UI to fit each one. Typically, all you need to do is design your UI to be flexible and optimize some elements for different sizes by providing [alternative resources](#) (such as alternative layouts that reposition some views or alternative dimension values for views). However, sometimes you might want to go a step further to optimize the overall user experience for different screen sizes. For example, tablets offer more space in which your application can present multiple sets of information at once, while a handset device usually requires that you split those sets apart and display them separately. So even though a UI designed for handsets will properly resize to fit a tablet, it does not fully leverage the potential of the tablet's screen to enhance the user experience.

With Android 3.0 (API level 11), Android introduced a new set of framework APIs that allow you to more effectively design activities that take advantage of large screens: the [Fragment](#) APIs. Fragments allow you to separate distinct behavioral components of your UI into separate parts, which you can then combine to create multi-pane layouts when running on a tablet or place in separate activities when running on a handset. Android 3.0 also introduced [ActionBar](#), which provides a dedicated UI at the top of the screen to identify the app and provide user actions and navigation.

This document provides guidance that can help you create an application that offers a unique and optimized user experience on both handsets and tablets, using fragments and the action bar.

Before you continue with this guide, it's important that you first read the guide to [Supporting Multiple Screens](#). That document describes the fundamental design principles for developing a UI that supports different screen sizes and densities with flexible layouts and alternative bitmaps, respectively.

## Basic Guidelines

Here are a few guidelines that will help you create an app that provides an optimized user experience on both tablets and handsets:

- **Build your activity designs based on fragments** that you can reuse in different combinations—in multi-pane layouts on tablets and single-pane layouts on handsets.

A [Fragment](#) represents a behavior or a portion of user interface in an activity. You can think of a fragment as a modular section of an activity (a "fragment" of an activity), which has its own lifecycle and which you can add or remove while the activity is running.

If you haven't used fragments yet, start by reading the [Fragments](#) developer guide.

- **Use the action bar**, but follow best practices and ensure your design is flexible enough for the system to adjust the action bar layout based on the screen size.

The [ActionBar](#) is a UI component for activities that replaces the traditional title bar at the top of the screen. By default, the action bar includes the application logo on the left side, followed by the activity title, and access to items from the options menu on the right side.

You can enable items from the options menu to appear directly in the action bar as "action items". You can also add navigation features to the action bar, such as tabs or a drop-down list, and use the application icon to supplement the system's *Back* button behavior with the option to navigate to your application's "home" activity or "up" the application's structural hierarchy.

This guide provides some tips for using the action bar in ways that support both tablets and handsets. For a detailed discussion of the action bar APIs, read the [Action Bar](#) developer guide.

- **Implement flexible layouts**, as discussed in the [Best Practices](#) for supporting multiple screens and the blog post, [Thinking Like a Web Designer](#).

A flexible layout design allows your application to adapt to variations in screen sizes. Not all tablets are the same size, nor are all handsets the same size. While you might provide different fragment combinations for "tablets" and "handsets", it's still necessary that each design be flexible to resize to variations in size and aspect ratio.

The following sections discuss the first two recommendations in more detail. For more information about creating flexible layouts, refer to the links provided above.

**Note:** Aside from one feature in the action bar, all the APIs needed to accomplish the recommendations in this document are available in Android 3.0. Additionally, you can even implement the fragment design patterns and remain backward-compatible with Android 1.6, by using the support library—discussed in the sidebar below.

## Creating Single-pane and Multi-pane Layouts

### Remaining backward-compatible

If you want to use fragments in your application *and* remain compatible with versions of Android older than 3.0, you can do so by using the Android [Support Library](#) (downloadable from the SDK Manager).

The support library includes APIs for [fragments](#), [loaders](#), and other APIs added in newer versions of Android. By simply adding this library to your Android project, you can use backward-compatible versions of these APIs in your application and remain compatible with Android 1.6 (your `android:minSdkVersion` value can be as low as "4"). For information about how to get the library and start using it, see the [Support Library](#) document.

The support library *does not* provide APIs for the action bar, but you can use code from the sample app, [Action Bar Compatibility](#), to create an action bar that supports all devices.

The most effective way to create a distinct user experience for tablets and handsets is to create layouts with different combinations of fragments, such that you can design "multi-pane" layouts for tablets and "single-pane"

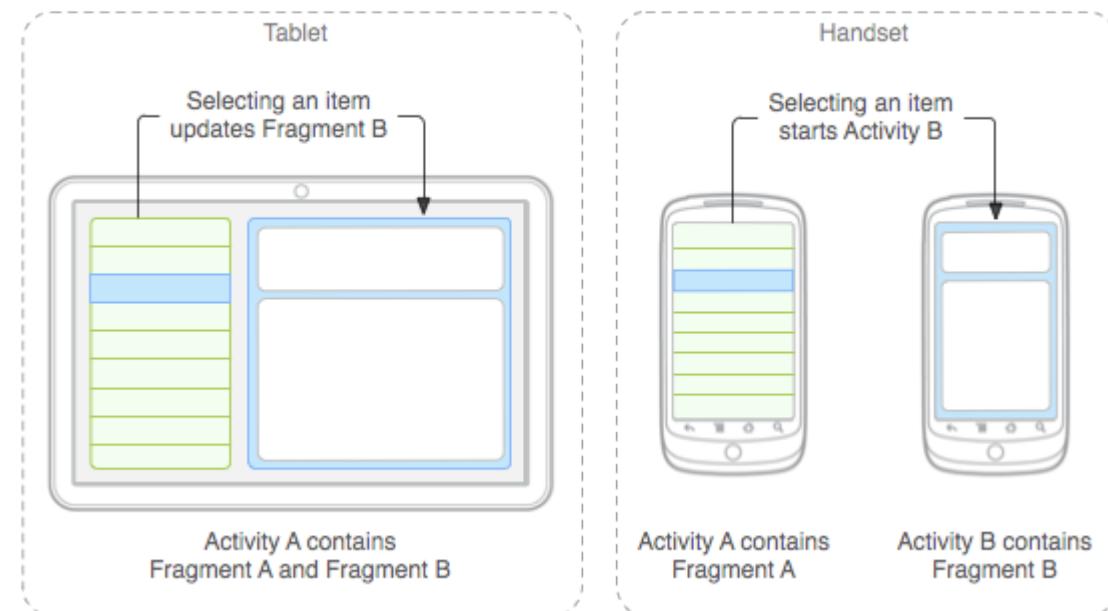
layouts for handsets. For example, a news application on a tablet might show a list of articles on the left side and a full article on the right side—selecting an article on the left updates the article view on the right. On a handset, however, these two components should appear on separate screens—selecting an article from a list changes the entire screen to show that article. There are two techniques to accomplish this design with fragments:

- *Multiple fragments, one activity*: Use one activity regardless of the device size, but decide at runtime whether to combine fragments in the layout (to create a multiple-pane design) or swap fragments (to create a single-pane design). Or...
- *Multiple fragments, multiple activities*: On a tablet, place multiple fragments in one activity; on a handset, use separate activities to host each fragment. For example, when the tablet design uses two fragments in an activity, use the same activity for handsets, but supply an alternative layout that includes just the first fragment. When running on a handset and you need to switch fragments (such as when the user selects an item), start another activity that hosts the second fragment.

The approach you choose depends on your design and personal preferences. The first option (one activity; swapping fragments) requires that you determine the screen size at runtime and dynamically add each fragment as appropriate—rather than declare the fragments in your activity's XML layout—because you *cannot* remove a fragment from an activity if it's been declared in the XML layout. When using the first technique, you might also need to update the action bar each time the fragments change, depending on what actions or navigation modes are available for each fragment. In some cases, these factors might not affect your design, so using one activity and swapping fragments might work well (especially if your tablet design requires that you add fragments dynamically anyway). Other times, however, dynamically swapping fragments for your handset design can make your code more complicated, because you must manage all the fragment combinations in the activity's code (rather than use alternative layout resources to define fragment combinations) and manage the back stack of fragments yourself (rather than allow the normal activity stack to handle back-navigation).

This guide focuses on the second option, in which you display each fragment in a separate activity when on a smaller screen. Using this technique means that you can use alternative layout files that define different fragment combinations for different screen sizes, keep fragment code modular, simplify action bar management, and let the system handle all the back stack work on handsets.

Figure 1 illustrates how an application with two fragments might be arranged for both handsets and tablets when using separate activities for the handset design:



**Figure 1.** Different design patterns for tablets and handsets when selecting an item to view its details.

In the application shown in figure 1, Activity A is the "main activity" and uses different layouts to display either one or two fragments at a time, depending on the size of the screen:

- On a tablet-sized screen, the Activity A layout contains both Fragment A and Fragment B.
- On a handset-sized screen, the Activity A layout contains only Fragment A (the list view). In order to show the details in Fragment B, Activity B must open.

**Note:** Activity B is never used on a tablet. It is simply a container to present Fragment B, so is only used on handset devices when the two fragments must be displayed separately.

Depending on the screen size, the system applies a different `main.xml` layout file:

`res/layout/main.xml` for handsets:

```
<?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">
    <!-- "Fragment A" -->
    <fragment class="com.example.android.TitlesFragment"
        android:id="@+id/list_frag"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</FrameLayout>
```

`res/layout-large/main.xml` for tablets:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/frags">
    <!-- "Fragment A" -->
    <fragment class="com.example.android.TitlesFragment"
        android:id="@+id/list_frag"
        android:layout_width="@dimen/titles_size"
        android:layout_height="match_parent"/>
    <!-- "Fragment B" -->
    <fragment class="com.example.android.DetailsFragment"
        android:id="@+id/details_frag"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

## Supporting sizes based on screen width

Android 3.2 (API level 13) adds new APIs that provide more fine-grain control over what screen sizes your app supports and what resources it uses, by declaring screen sizes based on the minimum width your layouts require. For example, both a 5" and 7" device qualify as a "large" screen, so your "large" layout resources are used on both devices. With API level 13, you can distinguish between these two sizes based on the screen width, as measured in density-independent pixels.

For details, read the blog post about [New Tools for Managing Screen Sizes](#).

**Note:** Although the above sample layout for tablets is based on the "large" screen configuration qualifier, you should also use the new "minimum width" size qualifiers in order to more precisely control the screen size at which the system applies your handset or tablet layout. See the sidebar for more information.

How the application responds when a user selects an item from the list depends on whether Fragment B is available in the layout:

- If Fragment B is in the layout, Activity A notifies Fragment B to update itself.
- If Fragment B is *not* in the layout, Activity A starts Activity B (which hosts Fragment B).

To implement this pattern for your application, it's important that you develop your fragments to be highly compartmentalized. Specifically, you should follow two guidelines:

- Do not manipulate one fragment directly from another.
- Keep all code that concerns content in a fragment inside that fragment, rather than putting it in the host activity's code.

To avoid directly calling one fragment from another, **define a callback interface in each fragment class** that it can use to deliver events to its host activity, which implements the callback interface. When the activity receives a callback due to an event (such as the user selecting a list item), the activity responds appropriately based on the current fragment configuration.

For example, Activity A from above can handle item selections depending on whether it's using the tablet or handset layout like this:

```
public class MainActivity extends Activity implements TitlesFragment.OnItemSelectedListener {  
    ...  
  
    /** This is a callback that the list fragment (Fragment A)  
     * calls when a list item is selected */  
    public void onItemSelected(int position) {  
        DisplayFragment displayFrag = (DisplayFragment) getSupportFragmentManager()  
            .findFragmentById(R.id.display_frag);  
        if (displayFrag == null) {  
            // DisplayFragment (Fragment B) is not in the layout (handset layout)  
            // so start DisplayActivity (Activity B)  
            // and pass it the info about the selected item  
            Intent intent = new Intent(this, DisplayActivity.class);  
            intent.putExtra("position", position);  
            startActivity(intent);  
        } else {  
            // DisplayFragment (Fragment B) is in the layout (tablet layout),  
            // so tell the fragment to update  
            displayFrag.updateContent(position);  
        }  
    }  
}
```

When `DisplayActivity` (Activity B) starts, it reads the data delivered by the [Intent](#) and passes it to the `DisplayFragment` (Fragment B).

If Fragment B needs to deliver a result back to Fragment A (because Activity B was started with [startActivityForResult\(\)](#)), then the process works similarly with a callback interface between Fragment B and Activity B. That is, Activity B implements a different callback interface defined by Fragment B. When Activity

B receives the callback with a result from the fragment, it sets the result for the activity (with [setResult\(\)](#)) and finishes itself. Activity A then receives the result and delivers it to Fragment A.

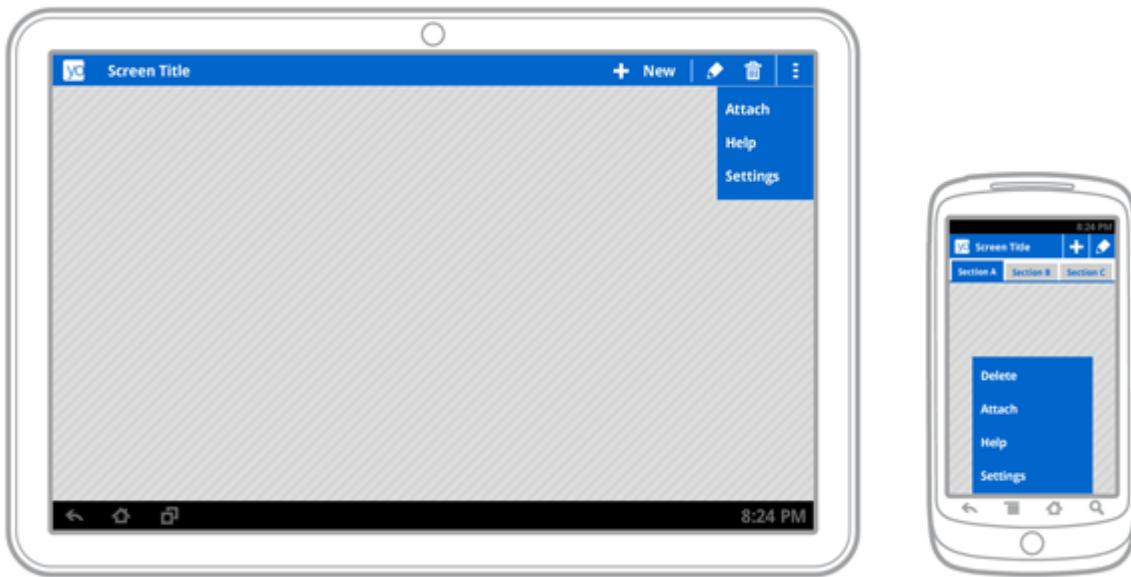
For a demonstration of this technique for creating different fragment combinations for tablets and handsets, see the updated version of the [Honeycomb Gallery](#) sample.

## Using the Action Bar

The [Action Bar](#) is an important UI component for Android apps on both tablets and handsets. To ensure that the action bar behaves appropriately on all screen sizes, it's important that you use the [ActionBar](#) APIs without adding complex customizations. By using the standard [ActionBar](#) APIs to design your action bar, the Android system does all the work to gracefully adapt the action bar for different screen sizes. Here are some important tips to follow when creating your action bar:

- When setting a menu item to be an action item, **avoid using the "always" value**. In your [menu resource](#), use "ifRoom" for the `android:showAsAction` attribute if you'd like the menu item to appear in the action bar. However, you might need "always" when an action view does not provide a default action for the overflow menu (that is, it must appear as an action view). However, you should not use "always" more than once or twice. In almost all other cases, use "ifRoom" as the value for "`android:showAsAction`" when you want the item to appear as an action item. Forcing too many action items into the action bar can create a cluttered UI and action items may overlap with other action bar elements such as the title or navigation items.
- When adding action items to the action bar with a text title, also **provide an icon**, when appropriate, and declare `showAsAction="ifRoom|withText"`. This way, if there's not enough room for the title, but there is enough room for the icon, then only the icon may be used.
- Always **provide a title** for your action items, even if you don't enable "withText", because users can view the title as a "tool-tip" by performing a "long click" on the item—the title text appears momentarily in a toast message. Providing a title is also critical for accessibility, because screen readers read aloud the item title even when not visible.
- **Avoid using custom navigation modes when possible**. Use the built-in tab and drop-down navigation modes when possible—they're designed so the system can adapt their presentation to different screen sizes. For example, when the width is too narrow for both tabs and other action items (such as a handset in portrait orientation), the tabs appear below the action bar (this is known as the "stacked action bar"). If you must build a custom navigation mode or other custom views in the action bar, thoroughly test them on smaller screens and make any necessary adjustments to support a narrow action bar.

For example, the mock-ups below demonstrate how the system may adapt an action bar based on the available screen space. On the handset, only two action items fit, so the remaining menu items appear in the overflow menu (because `android:showAsAction` was set to "ifRoom") and the tabs appear in a separate row (the stacked action bar). On the tablet, more action items can fit in the action bar and so do the tabs.



**Figure 2.** Mock-up showing how the system re-configures action bar components based on the available screen space.

## Using split action bar

When your application is running on Android 4.0 (API level 14) and higher, there's an extra mode available for the action bar called "split action bar." When you enable split action bar, a separate bar appears at the bottom of the screen to display all action items when the activity is running on a narrow screen (such as a portrait handset). Splitting the action bar ensures that a reasonable amount of space is available to display action items on a narrow screen and also leave room for navigation and title elements at the top.

To enable split action bar, simply add `uiOptions="splitActionBarWhenNarrow"` to your [`<activity>`](#) or [`<application>`](#) manifest element.



**Figure 3.** Split action bar with navigation tabs on the left; with the app icon and title disabled on the right.

If you'd like to hide the main action bar at the top, because you're using the built-in navigation tabs along with the split action bar, call [`setDisplayShowHomeEnabled\(false\)`](#) to disable the application icon in the action bar. In this case, there's now nothing left in the main action bar, so it disappears and all that's left are the navigation tabs at the top and the action items at the bottom, as shown by the second device in figure 3.

**Note:** Although the `uiOptions` attribute was added in Android 4.0 (API level 14), you can safely include it in your application even if your `minSdkVersion` is set to a value lower than "14" to remain compatible with older versions of Android. When running on older versions, the system simply ignores the attribute because it doesn't understand it. The only condition to adding it to your manifest is that you must compile your application against a platform version that supports API level 14 or higher. Just be sure that you don't openly use other APIs in your application code that aren't supported by the version declared by your `minSdkVersion` attribute.

## Using "up" navigation

As discussed in the [Action Bar](#) developer guide, you can use the application icon in the action bar to facilitate user navigation when appropriate—either as a method to get back to the "home" activity (similar to clicking the logo on a web site) or as a way to navigate up the application's structural hierarchy. Although it might seem similar to the standard *Back* navigation in some cases, the up navigation option provides a more predictable navigation method for situations in which the user may have entered from an external location, such as a notification, app widget, or a different application.

When using fragments in different combinations for different devices, it's important to give extra consideration to how your up navigation behaves in each configuration. For example, when on a handset and your application shows just one fragment at a time, it might be appropriate to enable up navigation to go up to the parent screen, whereas it's not necessary when showing the same fragment in a multi-pane configuration.

For more information about enabling up navigation, see the [Action Bar](#) developer guide.

## Other Design Tips

- When working with a [Listview](#), consider how you might provide more or less information in each list item based on the available space. That is, you can create alternative layouts to be used by the items in your list adapter such that a large screen might display more detail for each item.
- Create [alternative resource files](#) for values such as integers, dimensions, and even booleans. Using size qualifiers for these resources, you can easily apply different layout sizes, font sizes, or enable/disable features based on the current screen size.

Welcome to **Android Design**, your place for learning how to design exceptional Android apps.

[Creative Vision](#)



# Iconography

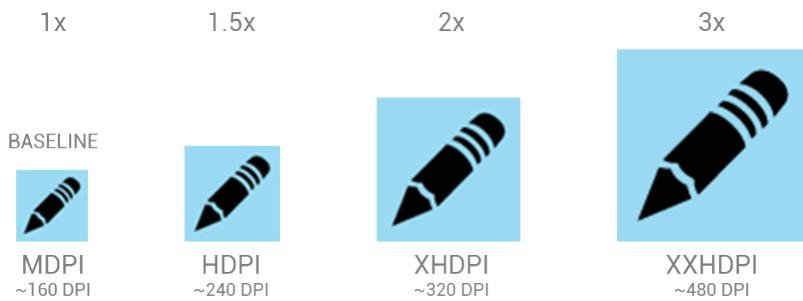
[Previous](#) [Next](#)



An icon is a graphic that takes up a small portion of screen real estate and provides a quick, intuitive representation of an action, a status, or an app.

When you design icons for your app, it's important to keep in mind that your app may be installed on a variety of devices that offer a range of pixel densities, as mentioned in [Devices and Displays](#). But you can make your icons look great on all devices by providing each icon in multiple sizes. When your app runs, Android checks the characteristics of the device screen and loads the appropriate density-specific assets for your app.

Because you will deliver each icon in multiple sizes to support different densities, the design guidelines below refer to the icon dimensions in `dp` units, which are based on the pixel dimensions of a medium-density (MDPI) screen.



So, to create an icon for different densities, you should follow the **2:3:4:6 scaling ratio** between the four primary densities (medium, high, x-high, and xx-high, respectively). For example, consider that the size for a launcher icon is specified to be 48x48 dp. This means the baseline (MDPI) asset is 48x48 px, and the high density (HDPI) asset should be 1.5x the baseline at 72x72 px, and the x-high density (XHDPI) asset should be 2x the baseline at 96x96 px, and so on.

**Note:** Android also supports low-density (LDPI) screens, but you normally don't need to create custom assets at this size because Android effectively down-scales your HDPI assets by 1/2 to match the expected size.

## Launcher

The launcher icon is the visual representation of your app on the Home or All Apps screen. Since the user can change the Home screen's wallpaper, make sure that your launcher icon is clearly visible on any type of background.



## Sizes & scale

- Launcher icons on a mobile device must be **48x48 dp**.
- Launcher icons for display on Google Play must be **512x512 pixels**.

## Proportions

- Full asset, **48x48**

## Style

Use a distinct silhouette. Three-dimensional, front view, with a slight perspective as if viewed from above, so that users perceive some depth.

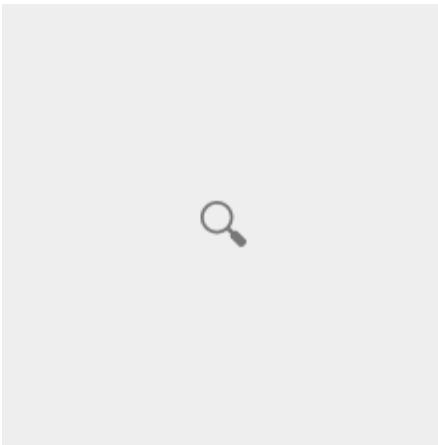


## Action Bar

Action bar icons are graphic buttons that represent the most important actions people can take within your app. Each one should employ a simple metaphor representing a single concept that most people can grasp at a glance.

Pre-defined glyphs should be used for certain common actions such as "refresh" and "share." The download link below provides a package with icons that are scaled for various screen densities and are suitable for use with the Holo Light and Holo Dark themes. The package also includes unstyled icons that you can modify to match your theme, in addition to Adobe® Illustrator® source files for further customization.

[Download the Action Bar Icon Pack](#)



## Sizes & scale

- Action bar icons for phones should be **32x32 dp**.

## Focal area & proportions

- Full asset, **32x32 dp**

Optical square, **24x24 dp**

## Style

Pictographic, flat, not too detailed, with smooth curves or sharp shapes. If the graphic is thin, rotate it 45° left or right to fill the focal space. The thickness of the strokes and negative spaces should be a minimum of 2 dp.

## Colors

Colors: #333333

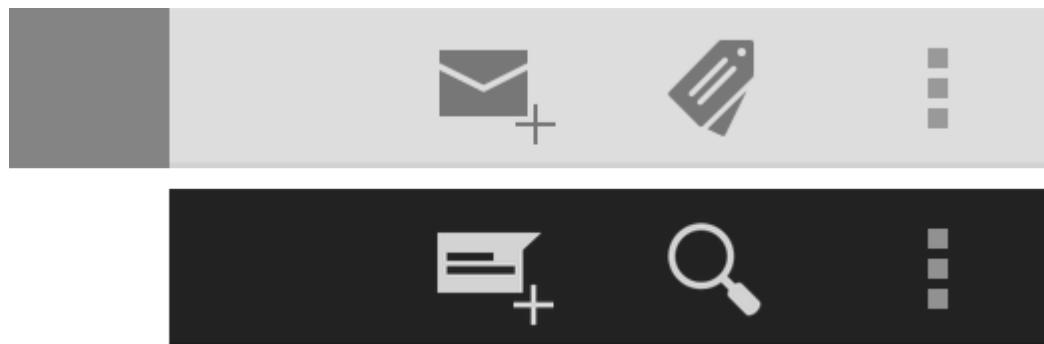
Enabled: **60%** opacity

Disabled: **30%** opacity

Colors: #FFFFFF

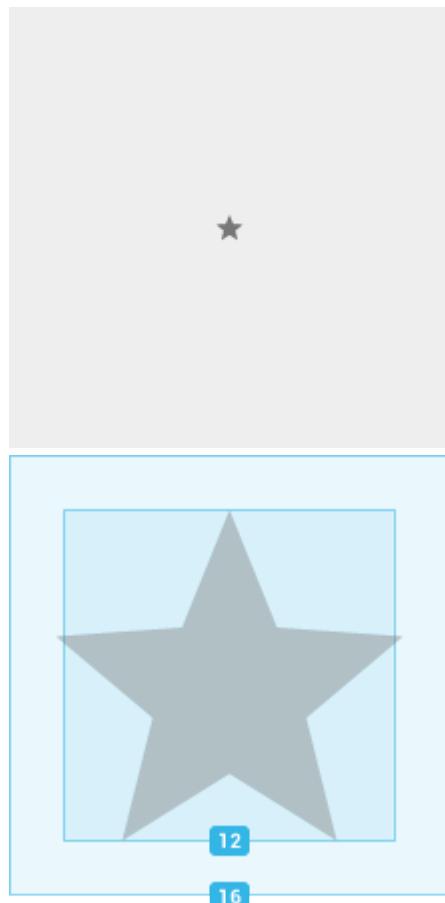
Enabled: **80%** opacity

Disabled: **30%** opacity



## Small / Contextual Icons

Within the body of your app, use small icons to surface actions and/or provide status for specific items. For example, in the Gmail app, each message has a star icon that marks the message as important.





## Sizes & scale

- Small icons should be **16x16 dp**.

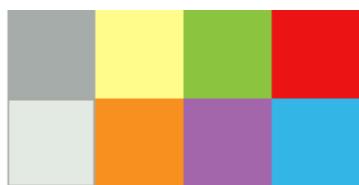
## Focal area & proportions

- Full asset, **16x16 dp**

Optical square, **12x12 dp**

## Style

Neutral, flat, and simple. Filled shapes are easier to see than thin strokes. Use a single visual metaphor so that a user can easily recognize and understand its purpose.



## Colors

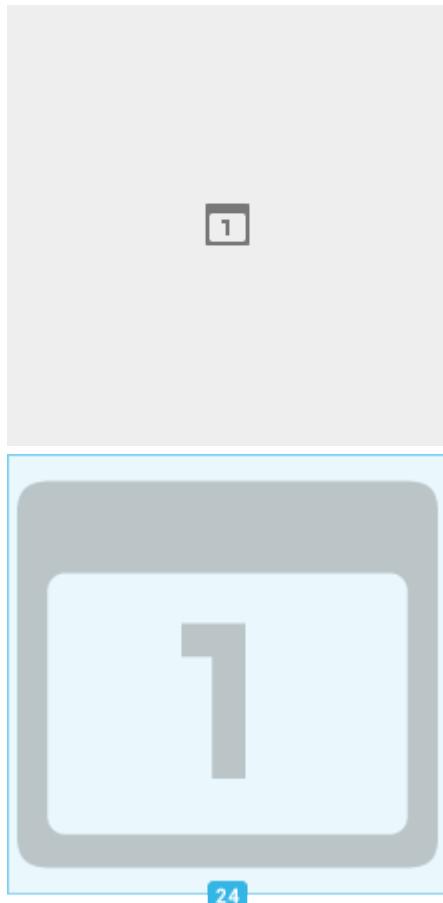
Use non-neutral colors sparingly and with purpose. For example, Gmail uses yellow in the star icon to indicate a bookmarked message. If an icon is actionable, choose a color that contrasts well with the background.

Dec 16  
san biodiesel, commodo        
helvetica aliquip photo

Dec 16  
eimer        
art party, you probably  
hem et officia mustache lo-

## Notification Icons

If your app generates notifications, provide an icon that the system can display in the status bar whenever a new notification is available.





## Sizes & scale

- Notification icons must be **24x24 dp**.

## Focal area & proportions

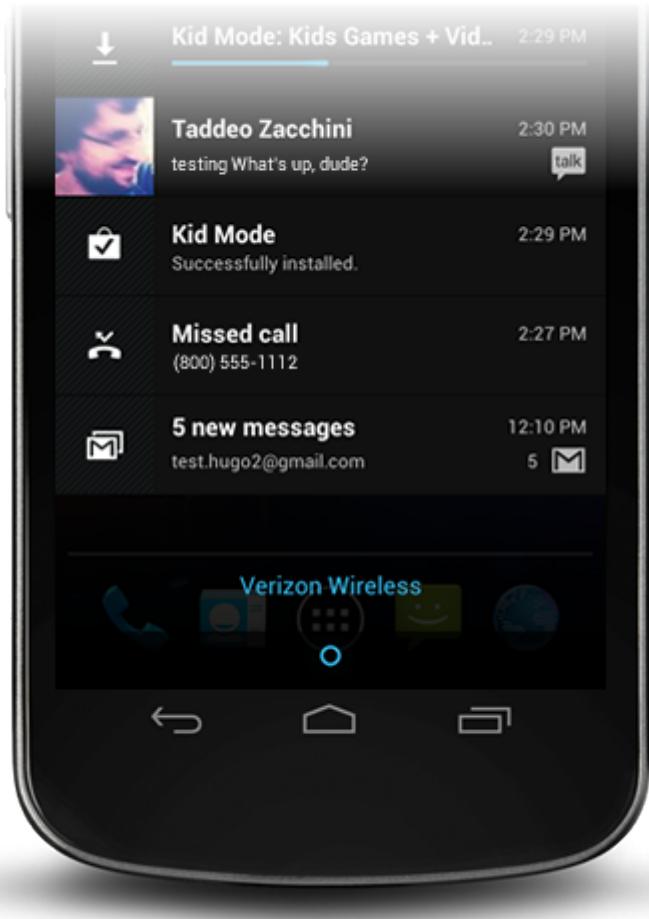
- Full asset, **24x24 dp**  
Optical square, **22x22 dp**

## Style

Keep the style flat and simple, using the same single, visual metaphor as your launcher icon.

## Colors

Notification icons must be entirely white. Also, the system may scale down and/or darken the icons.



## Design Tips

Here are some tips you might find useful as you create icons or other drawable assets for your application. These tips assume you are using Adobe® Photoshop® or a similar raster and vector image-editing program.

### Use vector shapes where possible

Many image-editing programs such as Adobe® Photoshop® allow you to use a combination of vector shapes and raster layers and effects. When possible, use vector shapes so that if the need arises, assets can be scaled up without loss of detail and edge crispness.

Using vectors also makes it easy to align edges and corners to pixel boundaries at smaller resolutions.

### Start with large artboards

Because you will need to create assets for different screen densities, it is best to start your icon designs on large artboards with dimensions that are multiples of the target icon sizes. For example, launcher icons are 48, 72, 96, or 144 pixels wide, depending on screen density (mdpi, hdpi, xhdpi, and xxhdpi, respectively). If you initially draw launcher icons on an 864x864 artboard, it will be easier and cleaner to adjust the icons when you scale the artboard down to the target sizes for final asset creation.

### When scaling, redraw bitmap layers as needed

If you scaled an image up from a bitmap layer, rather than from a vector layer, those layers will need to be redrawn manually to appear crisp at higher densities. For example if a 60x60 circle was painted as a bitmap for mdpi it will need to be repainted as a 90x90 circle for hdpi.

## Use common naming conventions for icon assets

Try to name files so that related assets will group together inside a directory when they are sorted alphabetically. In particular, it helps to use a common prefix for each icon type. For example:

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Note that you are not required to use a shared prefix of any type—doing so is for your convenience only.

## Set up a working space that organizes files by density

Supporting multiple screen densities means you must create multiple versions of the same icon. To help keep the multiple copies of files safe and easier to find, we recommend creating a directory structure in your working space that organizes asset files based on the target density. For example:

```
art/...
    mdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    hdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    xhdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png

xxhdpi/... _pre_production/... working_file.psd finished_asset.png
```

Because the structure in your working space is similar to that of the application, you can quickly determine which assets should be copied to each resources directory. Separating assets by density also helps you detect any variances in filenames across densities, which is important because corresponding assets for different densities must share the same filename.

For comparison, here's the resources directory structure of a typical application:

```
res/...
    drawable-ldpi/...
        finished_asset.png
    drawable-mdpi/...
        finished_asset.png
    drawable-hdpi/...
        finished_asset.png
    drawable-xhdpi/...
        finished_asset.png
```

For more information about how to save resources in the application project, see [Providing Resources](#).

## Remove unnecessary metadata from final assets

Although the Android SDK tools will automatically compress PNGs when packaging application resources into the application binary, a good practice is to remove unnecessary headers and metadata from your PNG assets. Tools such as [OptiPNG](#) or [Pngcrush](#) can ensure that this metadata is removed and that your image asset file sizes are optimized.

# Iconography

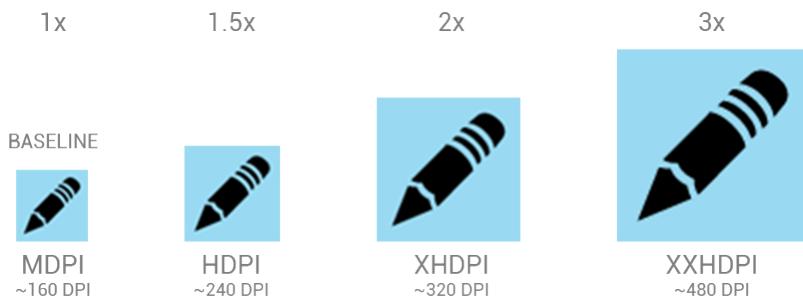
[Previous](#) [Next](#)



An icon is a graphic that takes up a small portion of screen real estate and provides a quick, intuitive representation of an action, a status, or an app.

When you design icons for your app, it's important to keep in mind that your app may be installed on a variety of devices that offer a range of pixel densities, as mentioned in [Devices and Displays](#). But you can make your icons look great on all devices by providing each icon in multiple sizes. When your app runs, Android checks the characteristics of the device screen and loads the appropriate density-specific assets for your app.

Because you will deliver each icon in multiple sizes to support different densities, the design guidelines below refer to the icon dimensions in `dp` units, which are based on the pixel dimensions of a medium-density (MDPI) screen.



So, to create an icon for different densities, you should follow the **2:3:4:6 scaling ratio** between the four primary densities (medium, high, x-high, and xx-high, respectively). For example, consider that the size for a launcher icon is specified to be 48x48 dp. This means the baseline (MDPI) asset is 48x48 px, and the high density (HDPI) asset should be 1.5x the baseline at 72x72 px, and the x-high density (XHDPI) asset should be 2x the baseline at 96x96 px, and so on.

**Note:** Android also supports low-density (LDPI) screens, but you normally don't need to create custom assets at this size because Android effectively down-scales your HDPI assets by 1/2 to match the expected size.

## Launcher

The launcher icon is the visual representation of your app on the Home or All Apps screen. Since the user can change the Home screen's wallpaper, make sure that your launcher icon is clearly visible on any type of background.



## Sizes & scale

- Launcher icons on a mobile device must be **48x48 dp**.
- Launcher icons for display on Google Play must be **512x512 pixels**.

## Proportions

- Full asset, **48x48**

## Style

Use a distinct silhouette. Three-dimensional, front view, with a slight perspective as if viewed from above, so that users perceive some depth.

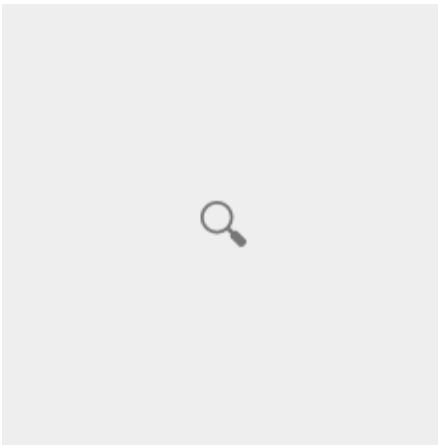


## Action Bar

Action bar icons are graphic buttons that represent the most important actions people can take within your app. Each one should employ a simple metaphor representing a single concept that most people can grasp at a glance.

Pre-defined glyphs should be used for certain common actions such as "refresh" and "share." The download link below provides a package with icons that are scaled for various screen densities and are suitable for use with the Holo Light and Holo Dark themes. The package also includes unstyled icons that you can modify to match your theme, in addition to Adobe® Illustrator® source files for further customization.

[Download the Action Bar Icon Pack](#)



## Sizes & scale

- Action bar icons for phones should be **32x32 dp**.

## Focal area & proportions

- Full asset, **32x32 dp**

Optical square, **24x24 dp**

## Style

Pictographic, flat, not too detailed, with smooth curves or sharp shapes. If the graphic is thin, rotate it 45° left or right to fill the focal space. The thickness of the strokes and negative spaces should be a minimum of 2 dp.

## Colors

Colors: #333333

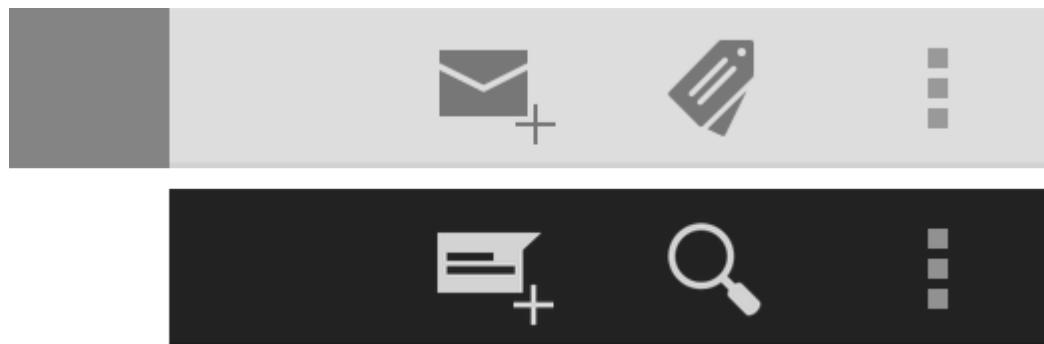
Enabled: **60%** opacity

Disabled: **30%** opacity

Colors: #FFFFFF

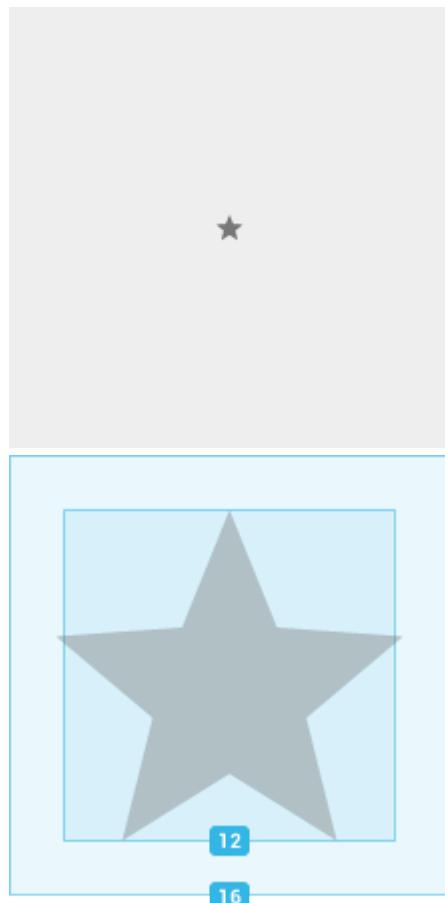
Enabled: **80%** opacity

Disabled: **30%** opacity



## Small / Contextual Icons

Within the body of your app, use small icons to surface actions and/or provide status for specific items. For example, in the Gmail app, each message has a star icon that marks the message as important.





## Sizes & scale

- Small icons should be **16x16 dp**.

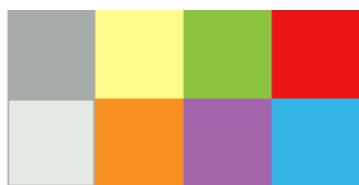
## Focal area & proportions

- Full asset, **16x16 dp**

Optical square, **12x12 dp**

## Style

Neutral, flat, and simple. Filled shapes are easier to see than thin strokes. Use a single visual metaphor so that a user can easily recognize and understand its purpose.



## Colors

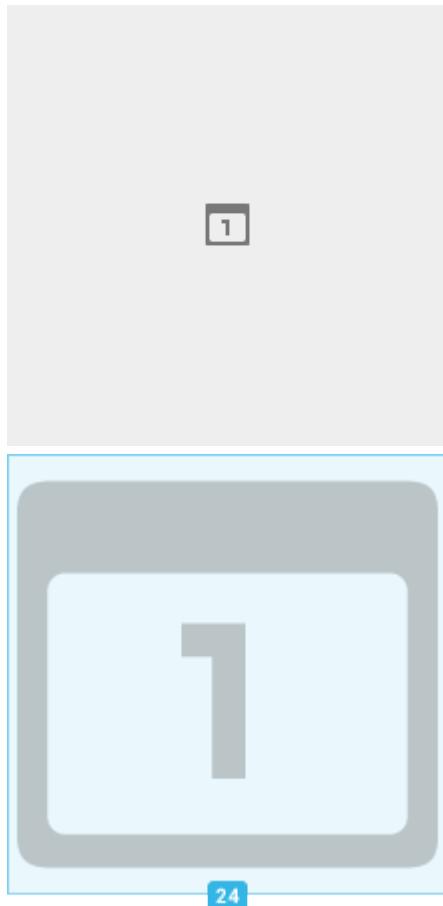
Use non-neutral colors sparingly and with purpose. For example, Gmail uses yellow in the star icon to indicate a bookmarked message. If an icon is actionable, choose a color that contrasts well with the background.

Dec 16  
san biodiesel, commodo        
helvetica aliquip photo

Dec 16  
eimer        
art party, you probably  
hem et officia mustache lo-

## Notification Icons

If your app generates notifications, provide an icon that the system can display in the status bar whenever a new notification is available.





## Sizes & scale

- Notification icons must be **24x24 dp**.

## Focal area & proportions

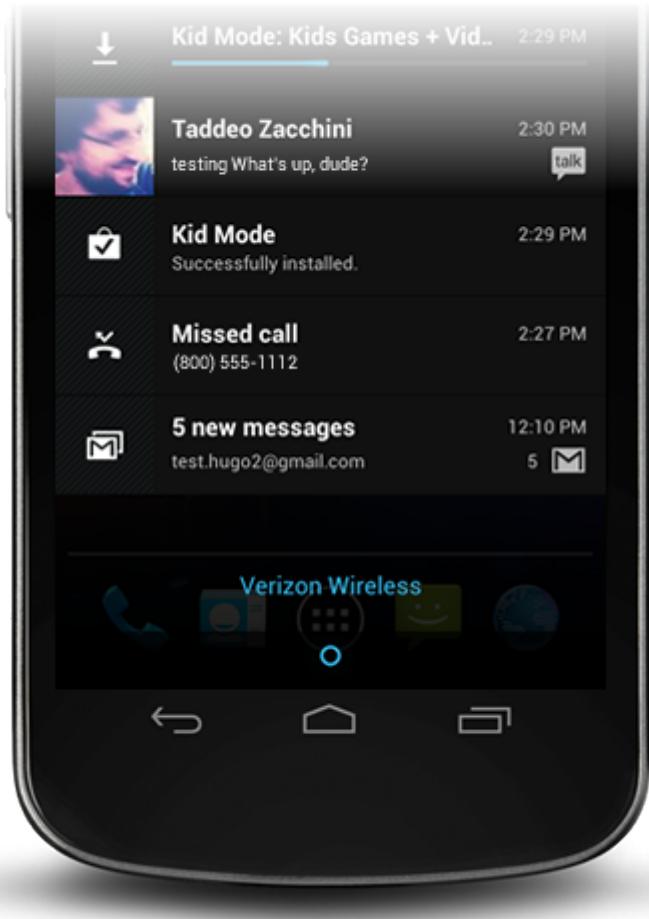
- Full asset, **24x24 dp**  
Optical square, **22x22 dp**

## Style

Keep the style flat and simple, using the same single, visual metaphor as your launcher icon.

## Colors

Notification icons must be entirely white. Also, the system may scale down and/or darken the icons.



## Design Tips

Here are some tips you might find useful as you create icons or other drawable assets for your application. These tips assume you are using Adobe® Photoshop® or a similar raster and vector image-editing program.

### Use vector shapes where possible

Many image-editing programs such as Adobe® Photoshop® allow you to use a combination of vector shapes and raster layers and effects. When possible, use vector shapes so that if the need arises, assets can be scaled up without loss of detail and edge crispness.

Using vectors also makes it easy to align edges and corners to pixel boundaries at smaller resolutions.

### Start with large artboards

Because you will need to create assets for different screen densities, it is best to start your icon designs on large artboards with dimensions that are multiples of the target icon sizes. For example, launcher icons are 48, 72, 96, or 144 pixels wide, depending on screen density (mdpi, hdpi, xhdpi, and xxhdpi, respectively). If you initially draw launcher icons on an 864x864 artboard, it will be easier and cleaner to adjust the icons when you scale the artboard down to the target sizes for final asset creation.

### When scaling, redraw bitmap layers as needed

If you scaled an image up from a bitmap layer, rather than from a vector layer, those layers will need to be redrawn manually to appear crisp at higher densities. For example if a 60x60 circle was painted as a bitmap for mdpi it will need to be repainted as a 90x90 circle for hdpi.

## Use common naming conventions for icon assets

Try to name files so that related assets will group together inside a directory when they are sorted alphabetically. In particular, it helps to use a common prefix for each icon type. For example:

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Note that you are not required to use a shared prefix of any type—doing so is for your convenience only.

## Set up a working space that organizes files by density

Supporting multiple screen densities means you must create multiple versions of the same icon. To help keep the multiple copies of files safe and easier to find, we recommend creating a directory structure in your working space that organizes asset files based on the target density. For example:

```
art/...
    mdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    hdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    xhdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png

xxhdpi/... _pre_production/... working_file.psd finished_asset.png
```

Because the structure in your working space is similar to that of the application, you can quickly determine which assets should be copied to each resources directory. Separating assets by density also helps you detect any variances in filenames across densities, which is important because corresponding assets for different densities must share the same filename.

For comparison, here's the resources directory structure of a typical application:

```
res/...
    drawable-ldpi/...
        finished_asset.png
    drawable-mdpi/...
        finished_asset.png
    drawable-hdpi/...
        finished_asset.png
    drawable-xhdpi/...
        finished_asset.png
```

For more information about how to save resources in the application project, see [Providing Resources](#).

## Remove unnecessary metadata from final assets

Although the Android SDK tools will automatically compress PNGs when packaging application resources into the application binary, a good practice is to remove unnecessary headers and metadata from your PNG assets. Tools such as [OptiPNG](#) or [Pngcrush](#) can ensure that this metadata is removed and that your image asset file sizes are optimized.

# Iconography

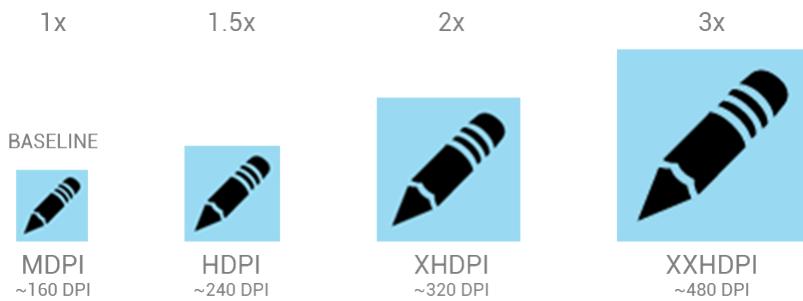
[Previous](#) [Next](#)



An icon is a graphic that takes up a small portion of screen real estate and provides a quick, intuitive representation of an action, a status, or an app.

When you design icons for your app, it's important to keep in mind that your app may be installed on a variety of devices that offer a range of pixel densities, as mentioned in [Devices and Displays](#). But you can make your icons look great on all devices by providing each icon in multiple sizes. When your app runs, Android checks the characteristics of the device screen and loads the appropriate density-specific assets for your app.

Because you will deliver each icon in multiple sizes to support different densities, the design guidelines below refer to the icon dimensions in `dp` units, which are based on the pixel dimensions of a medium-density (MDPI) screen.



So, to create an icon for different densities, you should follow the **2:3:4:6 scaling ratio** between the four primary densities (medium, high, x-high, and xx-high, respectively). For example, consider that the size for a launcher icon is specified to be 48x48 dp. This means the baseline (MDPI) asset is 48x48 px, and the high density (HDPI) asset should be 1.5x the baseline at 72x72 px, and the x-high density (XHDPI) asset should be 2x the baseline at 96x96 px, and so on.

**Note:** Android also supports low-density (LDPI) screens, but you normally don't need to create custom assets at this size because Android effectively down-scales your HDPI assets by 1/2 to match the expected size.

## Launcher

The launcher icon is the visual representation of your app on the Home or All Apps screen. Since the user can change the Home screen's wallpaper, make sure that your launcher icon is clearly visible on any type of background.



## Sizes & scale

- Launcher icons on a mobile device must be **48x48 dp**.
- Launcher icons for display on Google Play must be **512x512 pixels**.

## Proportions

- Full asset, **48x48**

## Style

Use a distinct silhouette. Three-dimensional, front view, with a slight perspective as if viewed from above, so that users perceive some depth.

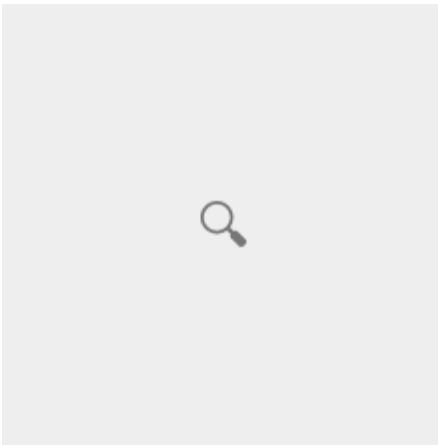


## Action Bar

Action bar icons are graphic buttons that represent the most important actions people can take within your app. Each one should employ a simple metaphor representing a single concept that most people can grasp at a glance.

Pre-defined glyphs should be used for certain common actions such as "refresh" and "share." The download link below provides a package with icons that are scaled for various screen densities and are suitable for use with the Holo Light and Holo Dark themes. The package also includes unstyled icons that you can modify to match your theme, in addition to Adobe® Illustrator® source files for further customization.

[Download the Action Bar Icon Pack](#)



## Sizes & scale

- Action bar icons for phones should be **32x32 dp**.

## Focal area & proportions

- Full asset, **32x32 dp**

Optical square, **24x24 dp**

## Style

Pictographic, flat, not too detailed, with smooth curves or sharp shapes. If the graphic is thin, rotate it 45° left or right to fill the focal space. The thickness of the strokes and negative spaces should be a minimum of 2 dp.

## Colors

Colors: #333333

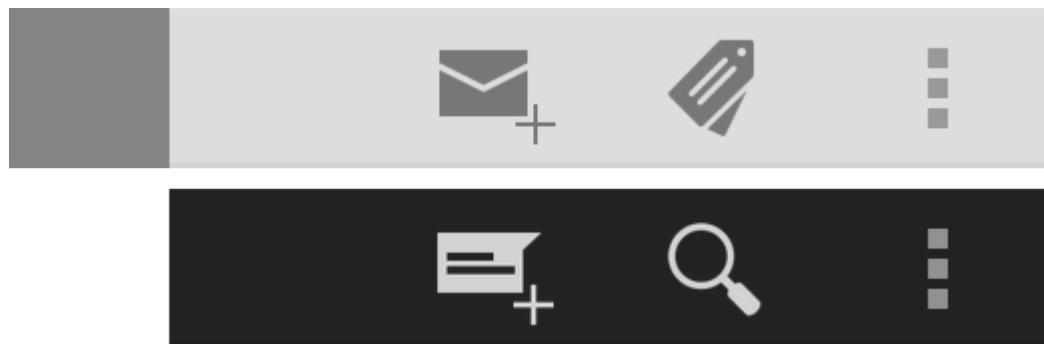
Enabled: **60%** opacity

Disabled: **30%** opacity

Colors: #FFFFFF

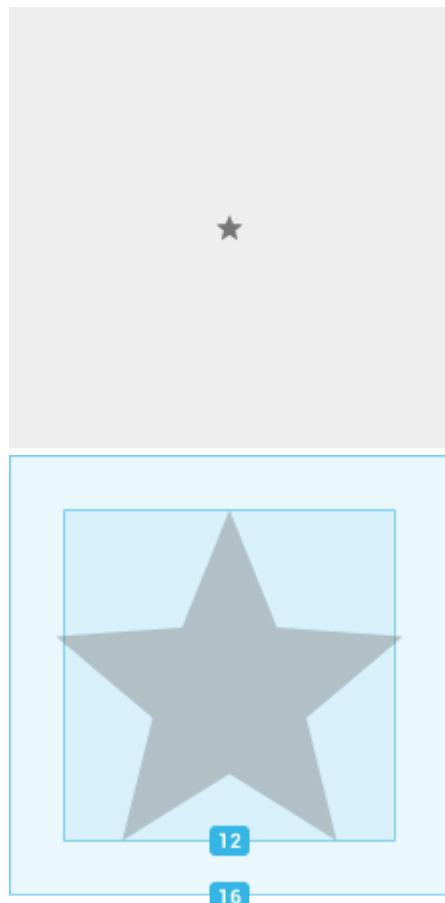
Enabled: **80%** opacity

Disabled: **30%** opacity



## Small / Contextual Icons

Within the body of your app, use small icons to surface actions and/or provide status for specific items. For example, in the Gmail app, each message has a star icon that marks the message as important.





## Sizes & scale

- Small icons should be **16x16 dp**.

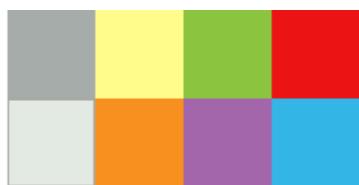
## Focal area & proportions

- Full asset, **16x16 dp**

Optical square, **12x12 dp**

## Style

Neutral, flat, and simple. Filled shapes are easier to see than thin strokes. Use a single visual metaphor so that a user can easily recognize and understand its purpose.



## Colors

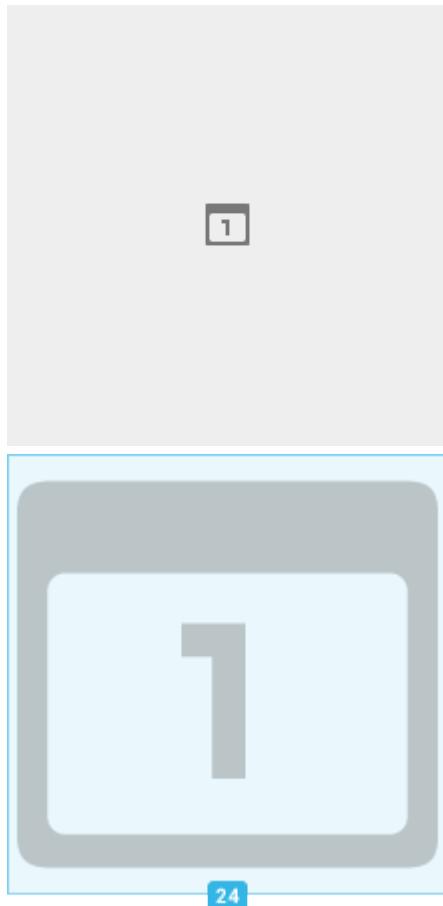
Use non-neutral colors sparingly and with purpose. For example, Gmail uses yellow in the star icon to indicate a bookmarked message. If an icon is actionable, choose a color that contrasts well with the background.

Dec 16  
san biodiesel, commodo        
helvetica aliquip photo

Dec 16  
eimer        
art party, you probably  
hem et officia mustache lo-

## Notification Icons

If your app generates notifications, provide an icon that the system can display in the status bar whenever a new notification is available.





## Sizes & scale

- Notification icons must be **24x24 dp**.

## Focal area & proportions

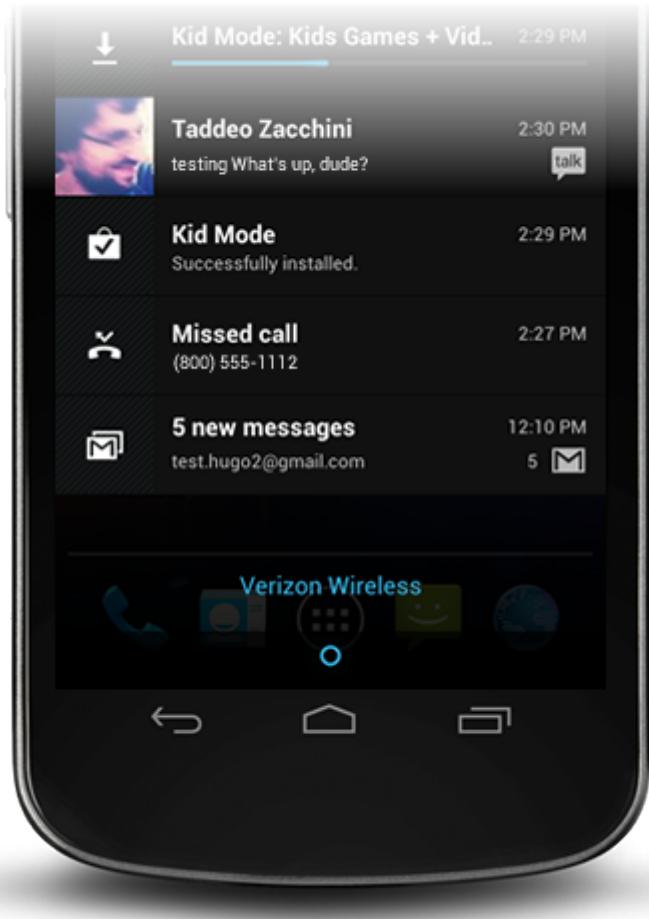
- Full asset, **24x24 dp**  
Optical square, **22x22 dp**

## Style

Keep the style flat and simple, using the same single, visual metaphor as your launcher icon.

## Colors

Notification icons must be entirely white. Also, the system may scale down and/or darken the icons.



## Design Tips

Here are some tips you might find useful as you create icons or other drawable assets for your application. These tips assume you are using Adobe® Photoshop® or a similar raster and vector image-editing program.

### Use vector shapes where possible

Many image-editing programs such as Adobe® Photoshop® allow you to use a combination of vector shapes and raster layers and effects. When possible, use vector shapes so that if the need arises, assets can be scaled up without loss of detail and edge crispness.

Using vectors also makes it easy to align edges and corners to pixel boundaries at smaller resolutions.

### Start with large artboards

Because you will need to create assets for different screen densities, it is best to start your icon designs on large artboards with dimensions that are multiples of the target icon sizes. For example, launcher icons are 48, 72, 96, or 144 pixels wide, depending on screen density (mdpi, hdpi, xhdpi, and xxhdpi, respectively). If you initially draw launcher icons on an 864x864 artboard, it will be easier and cleaner to adjust the icons when you scale the artboard down to the target sizes for final asset creation.

### When scaling, redraw bitmap layers as needed

If you scaled an image up from a bitmap layer, rather than from a vector layer, those layers will need to be redrawn manually to appear crisp at higher densities. For example if a 60x60 circle was painted as a bitmap for mdpi it will need to be repainted as a 90x90 circle for hdpi.

## Use common naming conventions for icon assets

Try to name files so that related assets will group together inside a directory when they are sorted alphabetically. In particular, it helps to use a common prefix for each icon type. For example:

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Note that you are not required to use a shared prefix of any type—doing so is for your convenience only.

## Set up a working space that organizes files by density

Supporting multiple screen densities means you must create multiple versions of the same icon. To help keep the multiple copies of files safe and easier to find, we recommend creating a directory structure in your working space that organizes asset files based on the target density. For example:

```
art/...
    mdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    hdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    xhdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png

xxhdpi/... _pre_production/... working_file.psd finished_asset.png
```

Because the structure in your working space is similar to that of the application, you can quickly determine which assets should be copied to each resources directory. Separating assets by density also helps you detect any variances in filenames across densities, which is important because corresponding assets for different densities must share the same filename.

For comparison, here's the resources directory structure of a typical application:

```
res/...
    drawable-ldpi/...
        finished_asset.png
    drawable-mdpi/...
        finished_asset.png
    drawable-hdpi/...
        finished_asset.png
    drawable-xhdpi/...
        finished_asset.png
```

For more information about how to save resources in the application project, see [Providing Resources](#).

## Remove unnecessary metadata from final assets

Although the Android SDK tools will automatically compress PNGs when packaging application resources into the application binary, a good practice is to remove unnecessary headers and metadata from your PNG assets. Tools such as [OptiPNG](#) or [Pngcrush](#) can ensure that this metadata is removed and that your image asset file sizes are optimized.

# Iconography

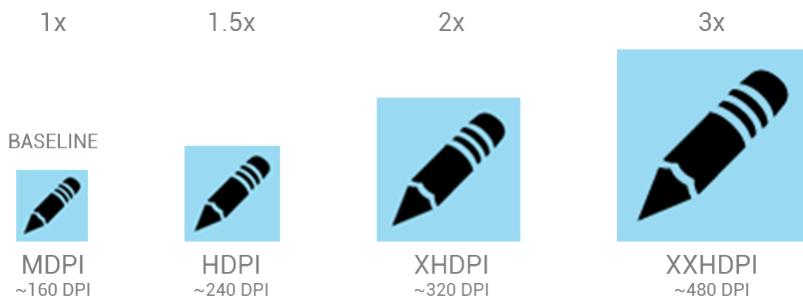
[Previous](#) [Next](#)



An icon is a graphic that takes up a small portion of screen real estate and provides a quick, intuitive representation of an action, a status, or an app.

When you design icons for your app, it's important to keep in mind that your app may be installed on a variety of devices that offer a range of pixel densities, as mentioned in [Devices and Displays](#). But you can make your icons look great on all devices by providing each icon in multiple sizes. When your app runs, Android checks the characteristics of the device screen and loads the appropriate density-specific assets for your app.

Because you will deliver each icon in multiple sizes to support different densities, the design guidelines below refer to the icon dimensions in `dp` units, which are based on the pixel dimensions of a medium-density (MDPI) screen.



So, to create an icon for different densities, you should follow the **2:3:4:6 scaling ratio** between the four primary densities (medium, high, x-high, and xx-high, respectively). For example, consider that the size for a launcher icon is specified to be 48x48 dp. This means the baseline (MDPI) asset is 48x48 px, and the high density (HDPI) asset should be 1.5x the baseline at 72x72 px, and the x-high density (XHDPI) asset should be 2x the baseline at 96x96 px, and so on.

**Note:** Android also supports low-density (LDPI) screens, but you normally don't need to create custom assets at this size because Android effectively down-scales your HDPI assets by 1/2 to match the expected size.

## Launcher

The launcher icon is the visual representation of your app on the Home or All Apps screen. Since the user can change the Home screen's wallpaper, make sure that your launcher icon is clearly visible on any type of background.



## Sizes & scale

- Launcher icons on a mobile device must be **48x48 dp**.
- Launcher icons for display on Google Play must be **512x512 pixels**.

## Proportions

- Full asset, **48x48**

## Style

Use a distinct silhouette. Three-dimensional, front view, with a slight perspective as if viewed from above, so that users perceive some depth.

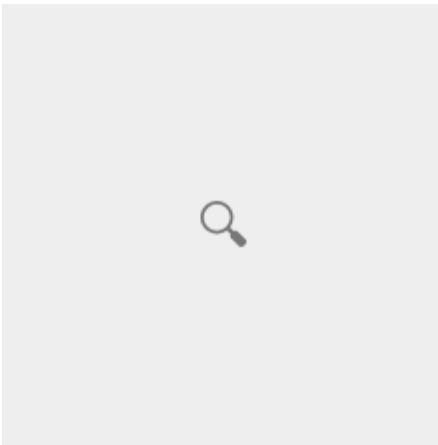


## Action Bar

Action bar icons are graphic buttons that represent the most important actions people can take within your app. Each one should employ a simple metaphor representing a single concept that most people can grasp at a glance.

Pre-defined glyphs should be used for certain common actions such as "refresh" and "share." The download link below provides a package with icons that are scaled for various screen densities and are suitable for use with the Holo Light and Holo Dark themes. The package also includes unstyled icons that you can modify to match your theme, in addition to Adobe® Illustrator® source files for further customization.

[Download the Action Bar Icon Pack](#)



## Sizes & scale

- Action bar icons for phones should be **32x32 dp**.

## Focal area & proportions

- Full asset, **32x32 dp**

Optical square, **24x24 dp**

## Style

Pictographic, flat, not too detailed, with smooth curves or sharp shapes. If the graphic is thin, rotate it 45° left or right to fill the focal space. The thickness of the strokes and negative spaces should be a minimum of 2 dp.

## Colors

Colors: #333333

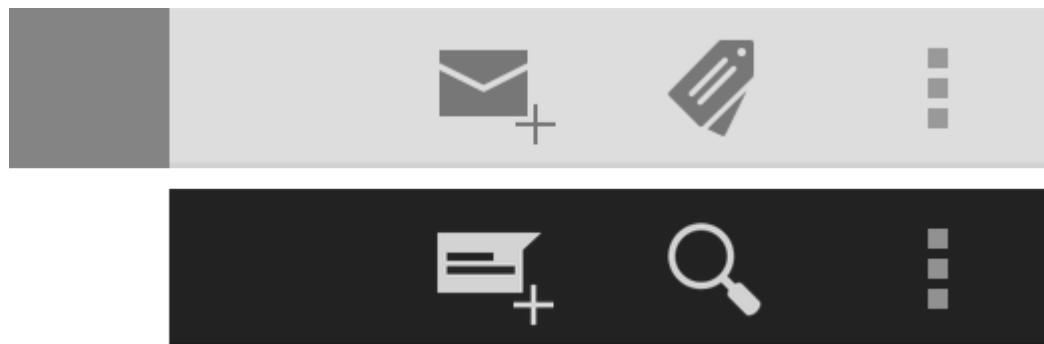
Enabled: **60%** opacity

Disabled: **30%** opacity

Colors: #FFFFFF

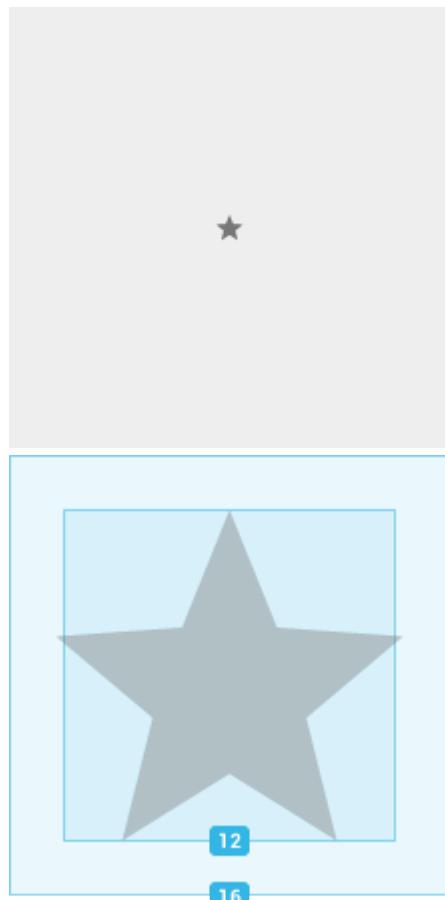
Enabled: **80%** opacity

Disabled: **30%** opacity



## Small / Contextual Icons

Within the body of your app, use small icons to surface actions and/or provide status for specific items. For example, in the Gmail app, each message has a star icon that marks the message as important.





## Sizes & scale

- Small icons should be **16x16 dp**.

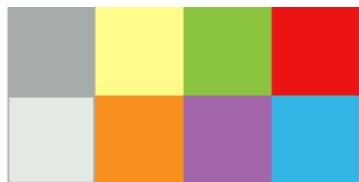
## Focal area & proportions

- Full asset, **16x16 dp**

Optical square, **12x12 dp**

## Style

Neutral, flat, and simple. Filled shapes are easier to see than thin strokes. Use a single visual metaphor so that a user can easily recognize and understand its purpose.



## Colors

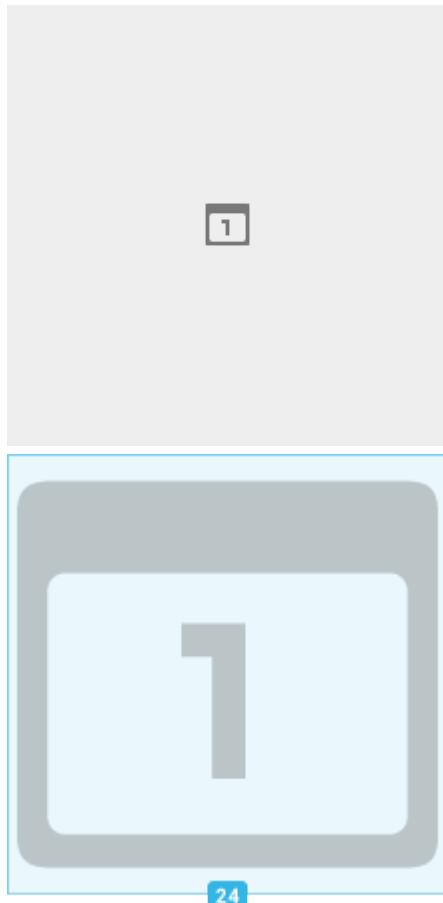
Use non-neutral colors sparingly and with purpose. For example, Gmail uses yellow in the star icon to indicate a bookmarked message. If an icon is actionable, choose a color that contrasts well with the background.

Dec 16  
san biodiesel, commodo        
helvetica aliquip photo

Dec 16  
eimer        
art party, you probably  
hem et officia mustache lo-

## Notification Icons

If your app generates notifications, provide an icon that the system can display in the status bar whenever a new notification is available.





## Sizes & scale

- Notification icons must be **24x24 dp**.

## Focal area & proportions

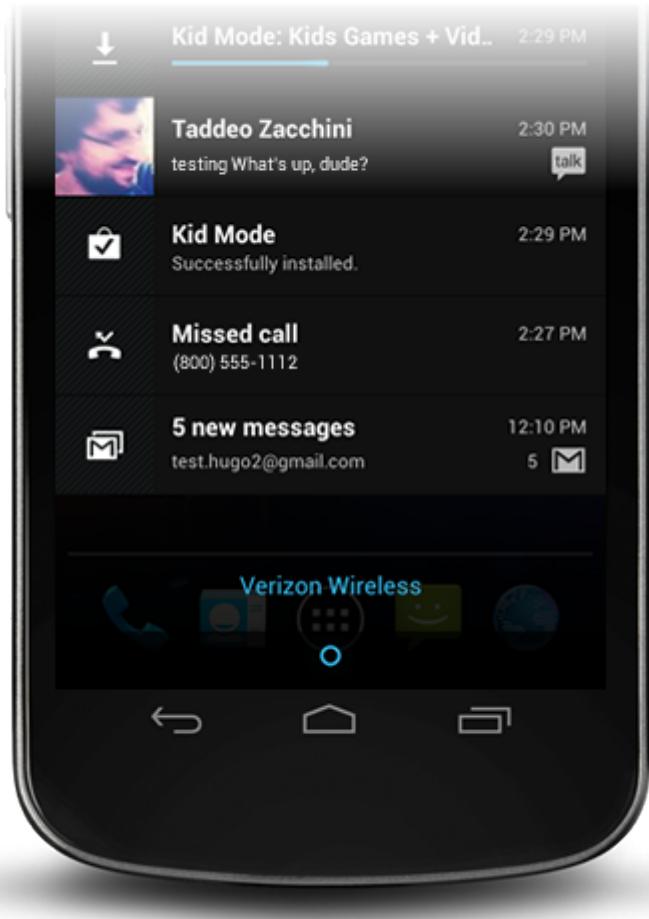
- Full asset, **24x24 dp**  
Optical square, **22x22 dp**

## Style

Keep the style flat and simple, using the same single, visual metaphor as your launcher icon.

## Colors

Notification icons must be entirely white. Also, the system may scale down and/or darken the icons.



## Design Tips

Here are some tips you might find useful as you create icons or other drawable assets for your application. These tips assume you are using Adobe® Photoshop® or a similar raster and vector image-editing program.

### Use vector shapes where possible

Many image-editing programs such as Adobe® Photoshop® allow you to use a combination of vector shapes and raster layers and effects. When possible, use vector shapes so that if the need arises, assets can be scaled up without loss of detail and edge crispness.

Using vectors also makes it easy to align edges and corners to pixel boundaries at smaller resolutions.

### Start with large artboards

Because you will need to create assets for different screen densities, it is best to start your icon designs on large artboards with dimensions that are multiples of the target icon sizes. For example, launcher icons are 48, 72, 96, or 144 pixels wide, depending on screen density (mdpi, hdpi, xhdpi, and xxhdpi, respectively). If you initially draw launcher icons on an 864x864 artboard, it will be easier and cleaner to adjust the icons when you scale the artboard down to the target sizes for final asset creation.

### When scaling, redraw bitmap layers as needed

If you scaled an image up from a bitmap layer, rather than from a vector layer, those layers will need to be redrawn manually to appear crisp at higher densities. For example if a 60x60 circle was painted as a bitmap for mdpi it will need to be repainted as a 90x90 circle for hdpi.

## Use common naming conventions for icon assets

Try to name files so that related assets will group together inside a directory when they are sorted alphabetically. In particular, it helps to use a common prefix for each icon type. For example:

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Note that you are not required to use a shared prefix of any type—doing so is for your convenience only.

## Set up a working space that organizes files by density

Supporting multiple screen densities means you must create multiple versions of the same icon. To help keep the multiple copies of files safe and easier to find, we recommend creating a directory structure in your working space that organizes asset files based on the target density. For example:

```
art/...
    mdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    hdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    xhdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png

xxhdpi/... _pre_production/... working_file.psd finished_asset.png
```

Because the structure in your working space is similar to that of the application, you can quickly determine which assets should be copied to each resources directory. Separating assets by density also helps you detect any variances in filenames across densities, which is important because corresponding assets for different densities must share the same filename.

For comparison, here's the resources directory structure of a typical application:

```
res/...
    drawable-ldpi/...
        finished_asset.png
    drawable-mdpi/...
        finished_asset.png
    drawable-hdpi/...
        finished_asset.png
    drawable-xhdpi/...
        finished_asset.png
```

For more information about how to save resources in the application project, see [Providing Resources](#).

## Remove unnecessary metadata from final assets

Although the Android SDK tools will automatically compress PNGs when packaging application resources into the application binary, a good practice is to remove unnecessary headers and metadata from your PNG assets. Tools such as [OptiPNG](#) or [Pngcrush](#) can ensure that this metadata is removed and that your image asset file sizes are optimized.

# Iconography

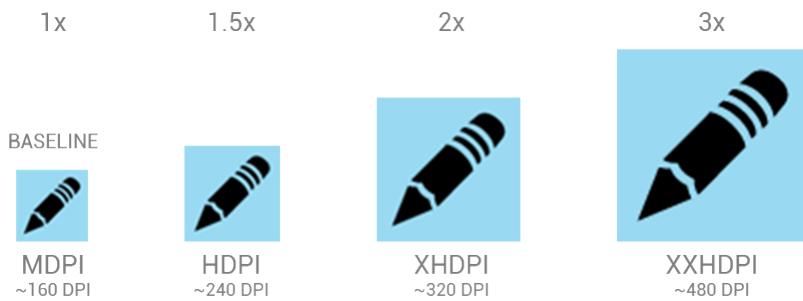
[Previous](#) [Next](#)



An icon is a graphic that takes up a small portion of screen real estate and provides a quick, intuitive representation of an action, a status, or an app.

When you design icons for your app, it's important to keep in mind that your app may be installed on a variety of devices that offer a range of pixel densities, as mentioned in [Devices and Displays](#). But you can make your icons look great on all devices by providing each icon in multiple sizes. When your app runs, Android checks the characteristics of the device screen and loads the appropriate density-specific assets for your app.

Because you will deliver each icon in multiple sizes to support different densities, the design guidelines below refer to the icon dimensions in `dp` units, which are based on the pixel dimensions of a medium-density (MDPI) screen.



So, to create an icon for different densities, you should follow the **2:3:4:6 scaling ratio** between the four primary densities (medium, high, x-high, and xx-high, respectively). For example, consider that the size for a launcher icon is specified to be 48x48 dp. This means the baseline (MDPI) asset is 48x48 px, and the high density (HDPI) asset should be 1.5x the baseline at 72x72 px, and the x-high density (XHDPI) asset should be 2x the baseline at 96x96 px, and so on.

**Note:** Android also supports low-density (LDPI) screens, but you normally don't need to create custom assets at this size because Android effectively down-scales your HDPI assets by 1/2 to match the expected size.

## Launcher

The launcher icon is the visual representation of your app on the Home or All Apps screen. Since the user can change the Home screen's wallpaper, make sure that your launcher icon is clearly visible on any type of background.



## Sizes & scale

- Launcher icons on a mobile device must be **48x48 dp**.
- Launcher icons for display on Google Play must be **512x512 pixels**.

## Proportions

- Full asset, **48x48**

## Style

Use a distinct silhouette. Three-dimensional, front view, with a slight perspective as if viewed from above, so that users perceive some depth.

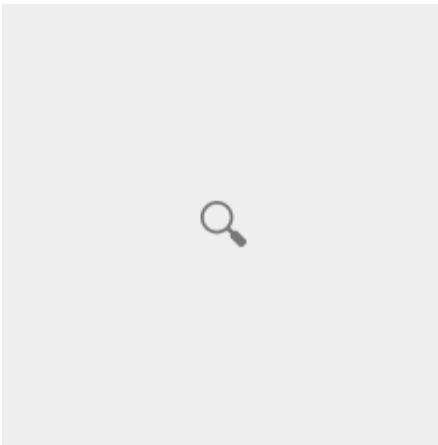


## Action Bar

Action bar icons are graphic buttons that represent the most important actions people can take within your app. Each one should employ a simple metaphor representing a single concept that most people can grasp at a glance.

Pre-defined glyphs should be used for certain common actions such as "refresh" and "share." The download link below provides a package with icons that are scaled for various screen densities and are suitable for use with the Holo Light and Holo Dark themes. The package also includes unstyled icons that you can modify to match your theme, in addition to Adobe® Illustrator® source files for further customization.

[Download the Action Bar Icon Pack](#)



## Sizes & scale

- Action bar icons for phones should be **32x32 dp**.

## Focal area & proportions

- Full asset, **32x32 dp**

Optical square, **24x24 dp**

## Style

Pictographic, flat, not too detailed, with smooth curves or sharp shapes. If the graphic is thin, rotate it 45° left or right to fill the focal space. The thickness of the strokes and negative spaces should be a minimum of 2 dp.

## Colors

Colors: #333333

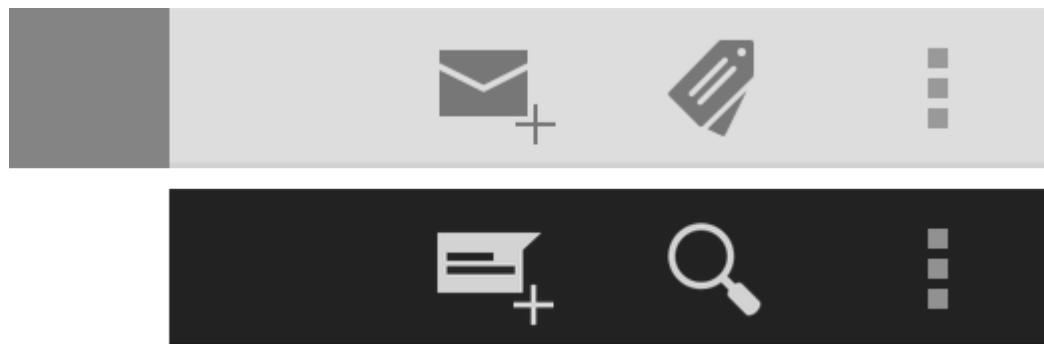
Enabled: **60%** opacity

Disabled: **30%** opacity

Colors: #FFFFFF

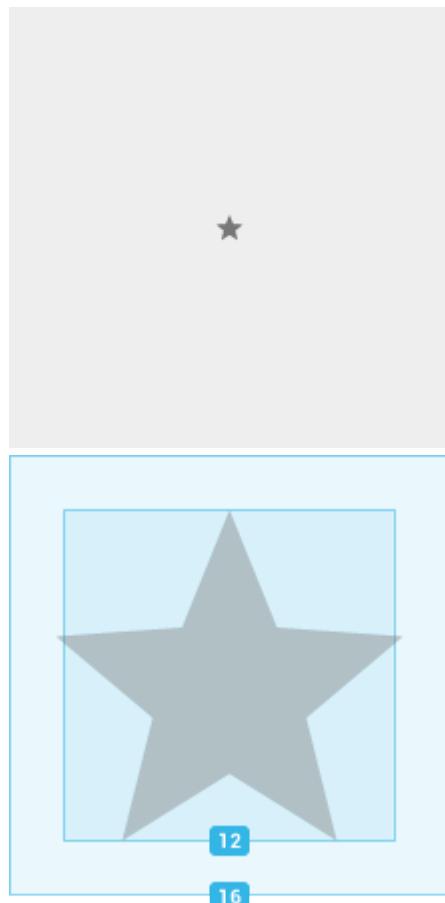
Enabled: **80%** opacity

Disabled: **30%** opacity



## Small / Contextual Icons

Within the body of your app, use small icons to surface actions and/or provide status for specific items. For example, in the Gmail app, each message has a star icon that marks the message as important.





## Sizes & scale

- Small icons should be **16x16 dp**.

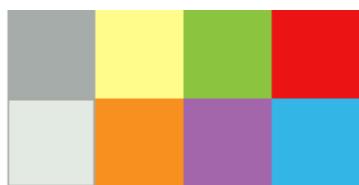
## Focal area & proportions

- Full asset, **16x16 dp**

Optical square, **12x12 dp**

## Style

Neutral, flat, and simple. Filled shapes are easier to see than thin strokes. Use a single visual metaphor so that a user can easily recognize and understand its purpose.



## Colors

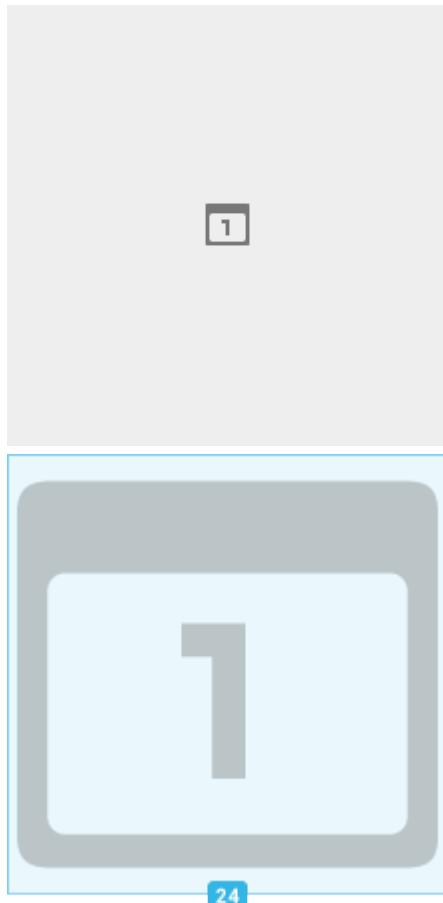
Use non-neutral colors sparingly and with purpose. For example, Gmail uses yellow in the star icon to indicate a bookmarked message. If an icon is actionable, choose a color that contrasts well with the background.

Dec 16  
san biodiesel, commodo        
helvetica aliquip photo

Dec 16  
eimer        
art party, you probably  
hem et officia mustache lo-

## Notification Icons

If your app generates notifications, provide an icon that the system can display in the status bar whenever a new notification is available.





## Sizes & scale

- Notification icons must be **24x24 dp**.

## Focal area & proportions

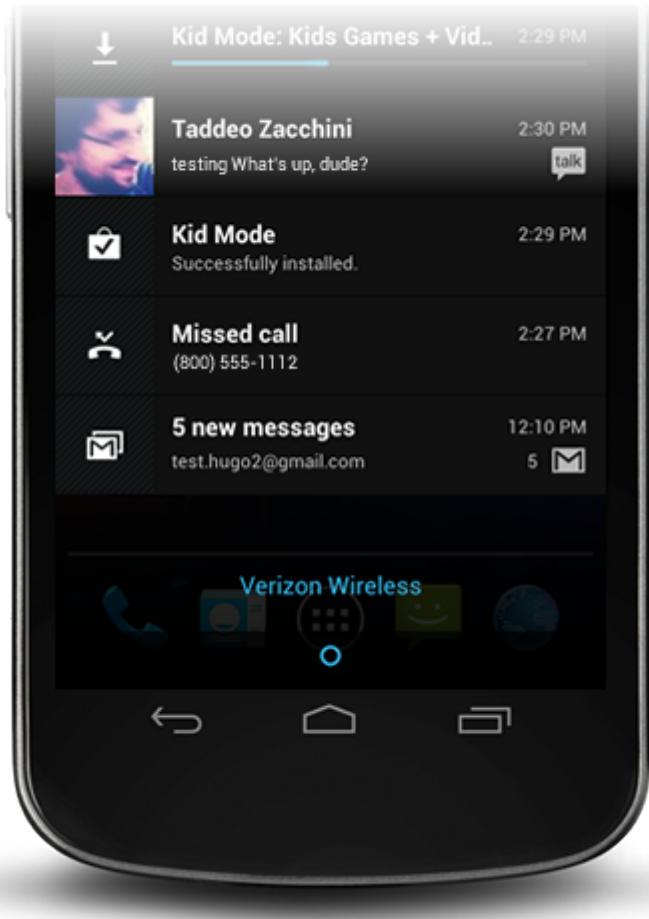
- Full asset, **24x24 dp**  
Optical square, **22x22 dp**

## Style

Keep the style flat and simple, using the same single, visual metaphor as your launcher icon.

## Colors

Notification icons must be entirely white. Also, the system may scale down and/or darken the icons.



## Design Tips

Here are some tips you might find useful as you create icons or other drawable assets for your application. These tips assume you are using Adobe® Photoshop® or a similar raster and vector image-editing program.

### Use vector shapes where possible

Many image-editing programs such as Adobe® Photoshop® allow you to use a combination of vector shapes and raster layers and effects. When possible, use vector shapes so that if the need arises, assets can be scaled up without loss of detail and edge crispness.

Using vectors also makes it easy to align edges and corners to pixel boundaries at smaller resolutions.

### Start with large artboards

Because you will need to create assets for different screen densities, it is best to start your icon designs on large artboards with dimensions that are multiples of the target icon sizes. For example, launcher icons are 48, 72, 96, or 144 pixels wide, depending on screen density (mdpi, hdpi, xhdpi, and xxhdpi, respectively). If you initially draw launcher icons on an 864x864 artboard, it will be easier and cleaner to adjust the icons when you scale the artboard down to the target sizes for final asset creation.

### When scaling, redraw bitmap layers as needed

If you scaled an image up from a bitmap layer, rather than from a vector layer, those layers will need to be redrawn manually to appear crisp at higher densities. For example if a 60x60 circle was painted as a bitmap for mdpi it will need to be repainted as a 90x90 circle for hdpi.

## Use common naming conventions for icon assets

Try to name files so that related assets will group together inside a directory when they are sorted alphabetically. In particular, it helps to use a common prefix for each icon type. For example:

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Note that you are not required to use a shared prefix of any type—doing so is for your convenience only.

## Set up a working space that organizes files by density

Supporting multiple screen densities means you must create multiple versions of the same icon. To help keep the multiple copies of files safe and easier to find, we recommend creating a directory structure in your working space that organizes asset files based on the target density. For example:

```
art/...
    mdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    hdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    xhdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png

xxhdpi/... _pre_production/... working_file.psd finished_asset.png
```

Because the structure in your working space is similar to that of the application, you can quickly determine which assets should be copied to each resources directory. Separating assets by density also helps you detect any variances in filenames across densities, which is important because corresponding assets for different densities must share the same filename.

For comparison, here's the resources directory structure of a typical application:

```
res/...
    drawable-ldpi/...
        finished_asset.png
    drawable-mdpi/...
        finished_asset.png
    drawable-hdpi/...
        finished_asset.png
    drawable-xhdpi/...
        finished_asset.png
```

For more information about how to save resources in the application project, see [Providing Resources](#).

## Remove unnecessary metadata from final assets

Although the Android SDK tools will automatically compress PNGs when packaging application resources into the application binary, a good practice is to remove unnecessary headers and metadata from your PNG assets. Tools such as [OptiPNG](#) or [Pngcrush](#) can ensure that this metadata is removed and that your image asset file sizes are optimized.

# Iconography

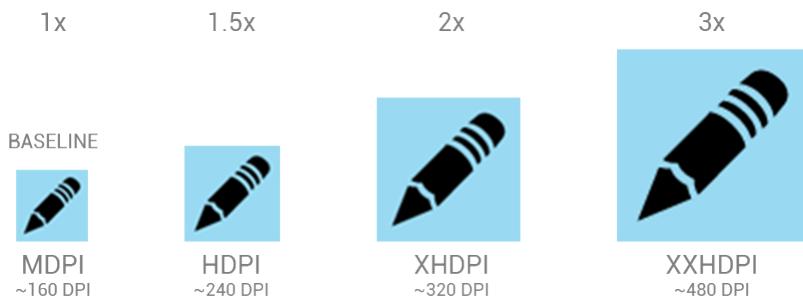
[Previous](#) [Next](#)



An icon is a graphic that takes up a small portion of screen real estate and provides a quick, intuitive representation of an action, a status, or an app.

When you design icons for your app, it's important to keep in mind that your app may be installed on a variety of devices that offer a range of pixel densities, as mentioned in [Devices and Displays](#). But you can make your icons look great on all devices by providing each icon in multiple sizes. When your app runs, Android checks the characteristics of the device screen and loads the appropriate density-specific assets for your app.

Because you will deliver each icon in multiple sizes to support different densities, the design guidelines below refer to the icon dimensions in `dp` units, which are based on the pixel dimensions of a medium-density (MDPI) screen.



So, to create an icon for different densities, you should follow the **2:3:4:6 scaling ratio** between the four primary densities (medium, high, x-high, and xx-high, respectively). For example, consider that the size for a launcher icon is specified to be 48x48 dp. This means the baseline (MDPI) asset is 48x48 px, and the high density (HDPI) asset should be 1.5x the baseline at 72x72 px, and the x-high density (XHDPI) asset should be 2x the baseline at 96x96 px, and so on.

**Note:** Android also supports low-density (LDPI) screens, but you normally don't need to create custom assets at this size because Android effectively down-scales your HDPI assets by 1/2 to match the expected size.

## Launcher

The launcher icon is the visual representation of your app on the Home or All Apps screen. Since the user can change the Home screen's wallpaper, make sure that your launcher icon is clearly visible on any type of background.



## Sizes & scale

- Launcher icons on a mobile device must be **48x48 dp**.
- Launcher icons for display on Google Play must be **512x512 pixels**.

## Proportions

- Full asset, **48x48**

## Style

Use a distinct silhouette. Three-dimensional, front view, with a slight perspective as if viewed from above, so that users perceive some depth.

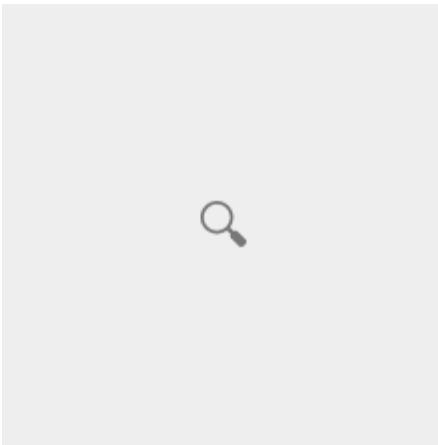


## Action Bar

Action bar icons are graphic buttons that represent the most important actions people can take within your app. Each one should employ a simple metaphor representing a single concept that most people can grasp at a glance.

Pre-defined glyphs should be used for certain common actions such as "refresh" and "share." The download link below provides a package with icons that are scaled for various screen densities and are suitable for use with the Holo Light and Holo Dark themes. The package also includes unstyled icons that you can modify to match your theme, in addition to Adobe® Illustrator® source files for further customization.

[Download the Action Bar Icon Pack](#)



## Sizes & scale

- Action bar icons for phones should be **32x32 dp**.

## Focal area & proportions

- Full asset, **32x32 dp**

Optical square, **24x24 dp**

## Style

Pictographic, flat, not too detailed, with smooth curves or sharp shapes. If the graphic is thin, rotate it 45° left or right to fill the focal space. The thickness of the strokes and negative spaces should be a minimum of 2 dp.

## Colors

Colors: #333333

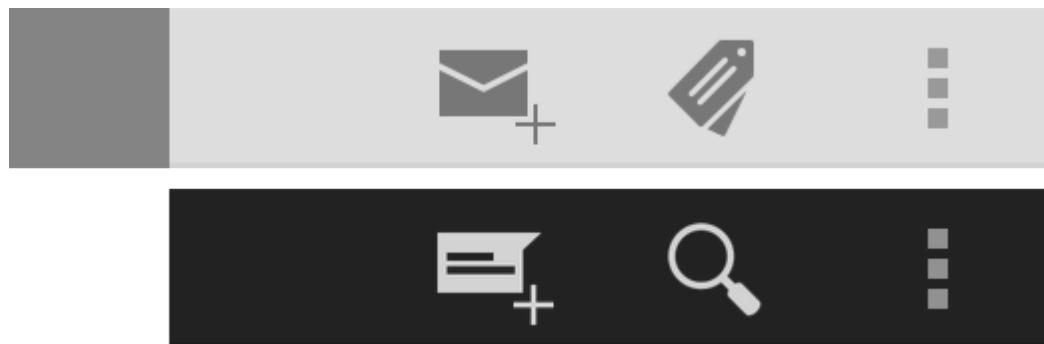
Enabled: **60%** opacity

Disabled: **30%** opacity

Colors: #FFFFFF

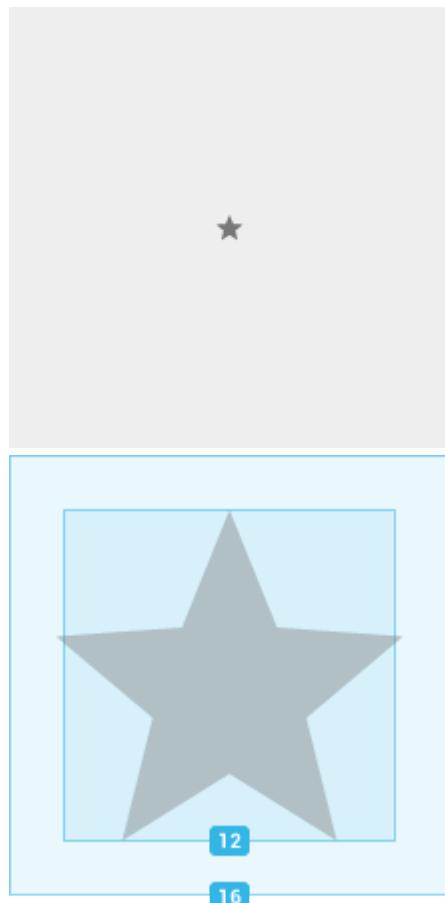
Enabled: **80%** opacity

Disabled: **30%** opacity



## Small / Contextual Icons

Within the body of your app, use small icons to surface actions and/or provide status for specific items. For example, in the Gmail app, each message has a star icon that marks the message as important.





## Sizes & scale

- Small icons should be **16x16 dp**.

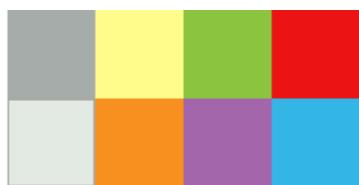
## Focal area & proportions

- Full asset, **16x16 dp**

Optical square, **12x12 dp**

## Style

Neutral, flat, and simple. Filled shapes are easier to see than thin strokes. Use a single visual metaphor so that a user can easily recognize and understand its purpose.



## Colors

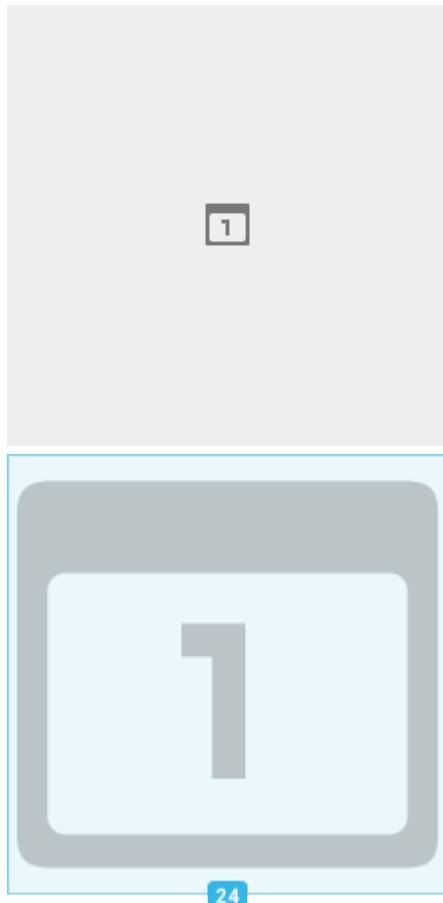
Use non-neutral colors sparingly and with purpose. For example, Gmail uses yellow in the star icon to indicate a bookmarked message. If an icon is actionable, choose a color that contrasts well with the background.

Dec 16  
san biodiesel, commodo        
helvetica aliquip photo

Dec 16  
eimer        
art party, you probably  
hem et officia mustache lo-

## Notification Icons

If your app generates notifications, provide an icon that the system can display in the status bar whenever a new notification is available.





## Sizes & scale

- Notification icons must be **24x24 dp**.

## Focal area & proportions

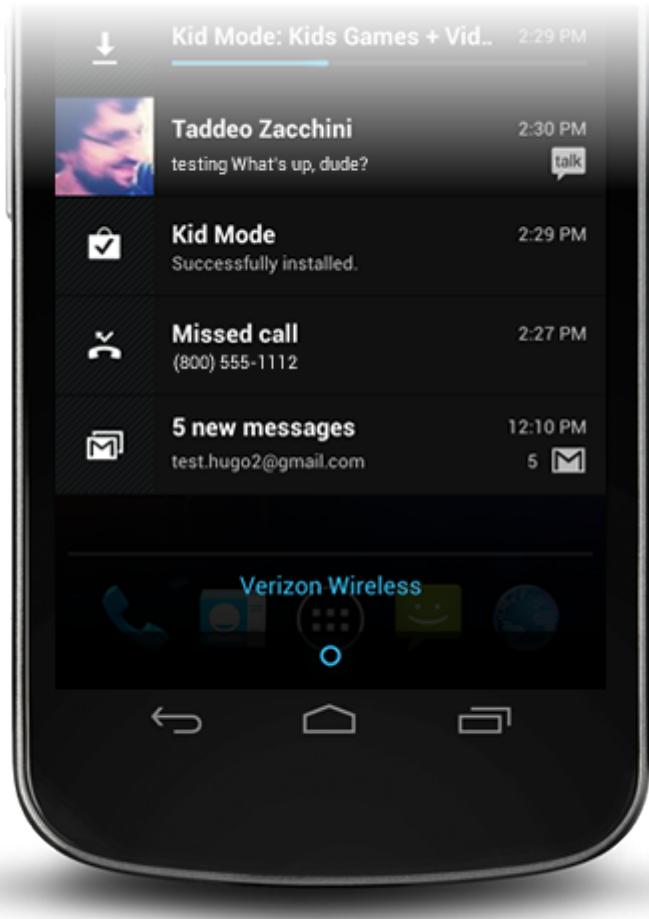
- Full asset, **24x24 dp**  
Optical square, **22x22 dp**

## Style

Keep the style flat and simple, using the same single, visual metaphor as your launcher icon.

## Colors

Notification icons must be entirely white. Also, the system may scale down and/or darken the icons.



## Design Tips

Here are some tips you might find useful as you create icons or other drawable assets for your application. These tips assume you are using Adobe® Photoshop® or a similar raster and vector image-editing program.

### Use vector shapes where possible

Many image-editing programs such as Adobe® Photoshop® allow you to use a combination of vector shapes and raster layers and effects. When possible, use vector shapes so that if the need arises, assets can be scaled up without loss of detail and edge crispness.

Using vectors also makes it easy to align edges and corners to pixel boundaries at smaller resolutions.

### Start with large artboards

Because you will need to create assets for different screen densities, it is best to start your icon designs on large artboards with dimensions that are multiples of the target icon sizes. For example, launcher icons are 48, 72, 96, or 144 pixels wide, depending on screen density (mdpi, hdpi, xhdpi, and xxhdpi, respectively). If you initially draw launcher icons on an 864x864 artboard, it will be easier and cleaner to adjust the icons when you scale the artboard down to the target sizes for final asset creation.

### When scaling, redraw bitmap layers as needed

If you scaled an image up from a bitmap layer, rather than from a vector layer, those layers will need to be redrawn manually to appear crisp at higher densities. For example if a 60x60 circle was painted as a bitmap for mdpi it will need to be repainted as a 90x90 circle for hdpi.

## Use common naming conventions for icon assets

Try to name files so that related assets will group together inside a directory when they are sorted alphabetically. In particular, it helps to use a common prefix for each icon type. For example:

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Note that you are not required to use a shared prefix of any type—doing so is for your convenience only.

## Set up a working space that organizes files by density

Supporting multiple screen densities means you must create multiple versions of the same icon. To help keep the multiple copies of files safe and easier to find, we recommend creating a directory structure in your working space that organizes asset files based on the target density. For example:

```
art/...
    mdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    hdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    xhdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png

xxhdpi/... _pre_production/... working_file.psd finished_asset.png
```

Because the structure in your working space is similar to that of the application, you can quickly determine which assets should be copied to each resources directory. Separating assets by density also helps you detect any variances in filenames across densities, which is important because corresponding assets for different densities must share the same filename.

For comparison, here's the resources directory structure of a typical application:

```
res/...
    drawable-ldpi/...
        finished_asset.png
    drawable-mdpi/...
        finished_asset.png
    drawable-hdpi/...
        finished_asset.png
    drawable-xhdpi/...
        finished_asset.png
```

For more information about how to save resources in the application project, see [Providing Resources](#).

## Remove unnecessary metadata from final assets

Although the Android SDK tools will automatically compress PNGs when packaging application resources into the application binary, a good practice is to remove unnecessary headers and metadata from your PNG assets. Tools such as [OptiPNG](#) or [Pngcrush](#) can ensure that this metadata is removed and that your image asset file sizes are optimized.

# Iconography

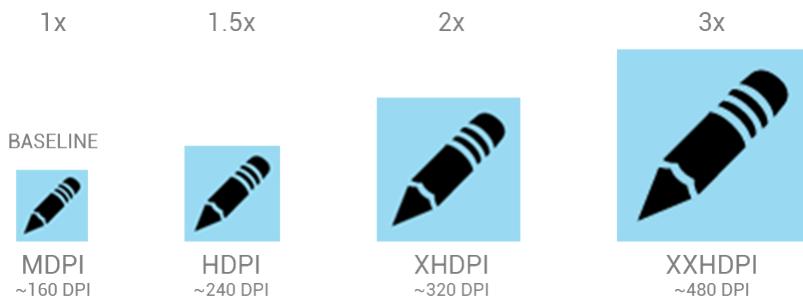
[Previous](#) [Next](#)



An icon is a graphic that takes up a small portion of screen real estate and provides a quick, intuitive representation of an action, a status, or an app.

When you design icons for your app, it's important to keep in mind that your app may be installed on a variety of devices that offer a range of pixel densities, as mentioned in [Devices and Displays](#). But you can make your icons look great on all devices by providing each icon in multiple sizes. When your app runs, Android checks the characteristics of the device screen and loads the appropriate density-specific assets for your app.

Because you will deliver each icon in multiple sizes to support different densities, the design guidelines below refer to the icon dimensions in `dp` units, which are based on the pixel dimensions of a medium-density (MDPI) screen.



So, to create an icon for different densities, you should follow the **2:3:4:6 scaling ratio** between the four primary densities (medium, high, x-high, and xx-high, respectively). For example, consider that the size for a launcher icon is specified to be 48x48 dp. This means the baseline (MDPI) asset is 48x48 px, and the high density (HDPI) asset should be 1.5x the baseline at 72x72 px, and the x-high density (XHDPI) asset should be 2x the baseline at 96x96 px, and so on.

**Note:** Android also supports low-density (LDPI) screens, but you normally don't need to create custom assets at this size because Android effectively down-scales your HDPI assets by 1/2 to match the expected size.

## Launcher

The launcher icon is the visual representation of your app on the Home or All Apps screen. Since the user can change the Home screen's wallpaper, make sure that your launcher icon is clearly visible on any type of background.



## Sizes & scale

- Launcher icons on a mobile device must be **48x48 dp**.
- Launcher icons for display on Google Play must be **512x512 pixels**.

## Proportions

- Full asset, **48x48**

## Style

Use a distinct silhouette. Three-dimensional, front view, with a slight perspective as if viewed from above, so that users perceive some depth.

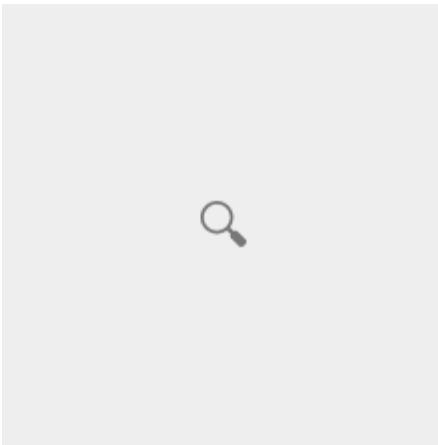


## Action Bar

Action bar icons are graphic buttons that represent the most important actions people can take within your app. Each one should employ a simple metaphor representing a single concept that most people can grasp at a glance.

Pre-defined glyphs should be used for certain common actions such as "refresh" and "share." The download link below provides a package with icons that are scaled for various screen densities and are suitable for use with the Holo Light and Holo Dark themes. The package also includes unstyled icons that you can modify to match your theme, in addition to Adobe® Illustrator® source files for further customization.

[Download the Action Bar Icon Pack](#)



## Sizes & scale

- Action bar icons for phones should be **32x32 dp**.

## Focal area & proportions

- Full asset, **32x32 dp**

Optical square, **24x24 dp**

## Style

Pictographic, flat, not too detailed, with smooth curves or sharp shapes. If the graphic is thin, rotate it 45° left or right to fill the focal space. The thickness of the strokes and negative spaces should be a minimum of 2 dp.

## Colors

Colors: #333333

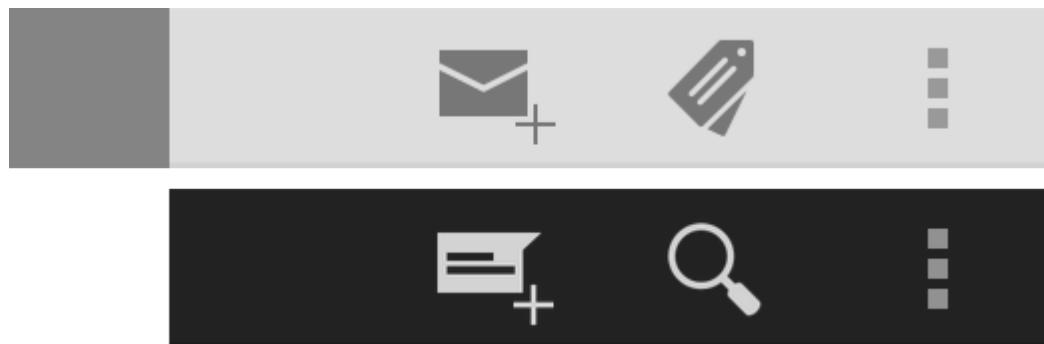
Enabled: **60%** opacity

Disabled: **30%** opacity

Colors: #FFFFFF

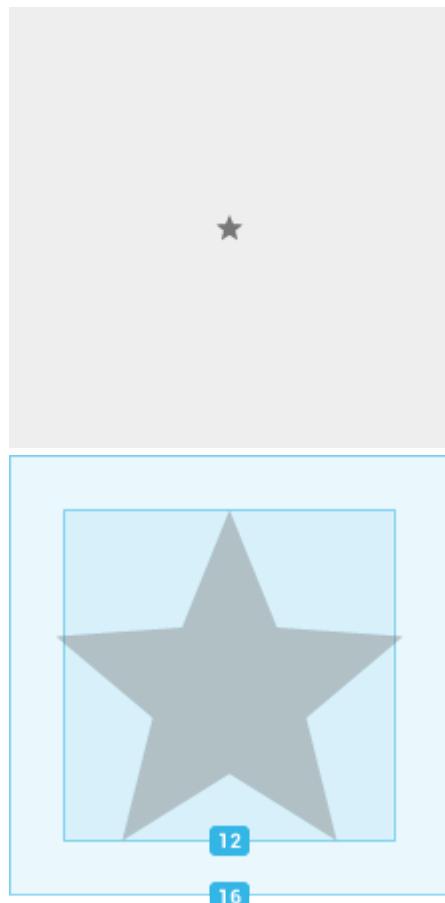
Enabled: **80%** opacity

Disabled: **30%** opacity



## Small / Contextual Icons

Within the body of your app, use small icons to surface actions and/or provide status for specific items. For example, in the Gmail app, each message has a star icon that marks the message as important.





## Sizes & scale

- Small icons should be **16x16 dp**.

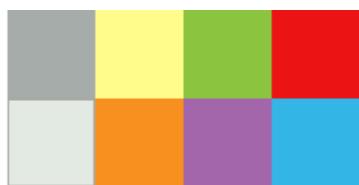
## Focal area & proportions

- Full asset, **16x16 dp**

Optical square, **12x12 dp**

## Style

Neutral, flat, and simple. Filled shapes are easier to see than thin strokes. Use a single visual metaphor so that a user can easily recognize and understand its purpose.



## Colors

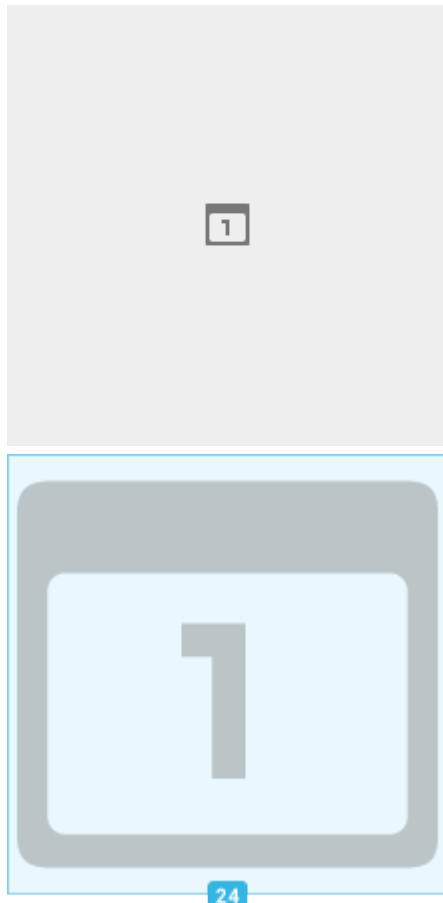
Use non-neutral colors sparingly and with purpose. For example, Gmail uses yellow in the star icon to indicate a bookmarked message. If an icon is actionable, choose a color that contrasts well with the background.

Dec 16  
san biodiesel, commodo        
helvetica aliquip photo

Dec 16  
eimer        
art party, you probably  
hem et officia mustache lo-

## Notification Icons

If your app generates notifications, provide an icon that the system can display in the status bar whenever a new notification is available.





## Sizes & scale

- Notification icons must be **24x24 dp**.

## Focal area & proportions

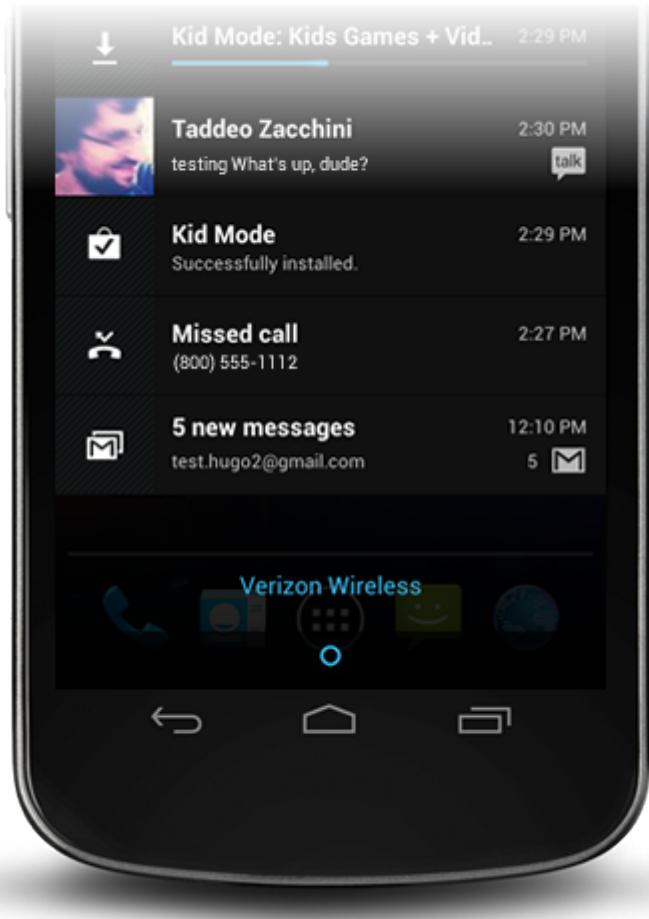
- Full asset, **24x24 dp**  
Optical square, **22x22 dp**

## Style

Keep the style flat and simple, using the same single, visual metaphor as your launcher icon.

## Colors

Notification icons must be entirely white. Also, the system may scale down and/or darken the icons.



## Design Tips

Here are some tips you might find useful as you create icons or other drawable assets for your application. These tips assume you are using Adobe® Photoshop® or a similar raster and vector image-editing program.

### Use vector shapes where possible

Many image-editing programs such as Adobe® Photoshop® allow you to use a combination of vector shapes and raster layers and effects. When possible, use vector shapes so that if the need arises, assets can be scaled up without loss of detail and edge crispness.

Using vectors also makes it easy to align edges and corners to pixel boundaries at smaller resolutions.

### Start with large artboards

Because you will need to create assets for different screen densities, it is best to start your icon designs on large artboards with dimensions that are multiples of the target icon sizes. For example, launcher icons are 48, 72, 96, or 144 pixels wide, depending on screen density (mdpi, hdpi, xhdpi, and xxhdpi, respectively). If you initially draw launcher icons on an 864x864 artboard, it will be easier and cleaner to adjust the icons when you scale the artboard down to the target sizes for final asset creation.

### When scaling, redraw bitmap layers as needed

If you scaled an image up from a bitmap layer, rather than from a vector layer, those layers will need to be redrawn manually to appear crisp at higher densities. For example if a 60x60 circle was painted as a bitmap for mdpi it will need to be repainted as a 90x90 circle for hdpi.

## Use common naming conventions for icon assets

Try to name files so that related assets will group together inside a directory when they are sorted alphabetically. In particular, it helps to use a common prefix for each icon type. For example:

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Note that you are not required to use a shared prefix of any type—doing so is for your convenience only.

## Set up a working space that organizes files by density

Supporting multiple screen densities means you must create multiple versions of the same icon. To help keep the multiple copies of files safe and easier to find, we recommend creating a directory structure in your working space that organizes asset files based on the target density. For example:

```
art/...
    mdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    hdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    xhdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png

xxhdpi/... _pre_production/... working_file.psd finished_asset.png
```

Because the structure in your working space is similar to that of the application, you can quickly determine which assets should be copied to each resources directory. Separating assets by density also helps you detect any variances in filenames across densities, which is important because corresponding assets for different densities must share the same filename.

For comparison, here's the resources directory structure of a typical application:

```
res/...
    drawable-ldpi/...
        finished_asset.png
    drawable-mdpi/...
        finished_asset.png
    drawable-hdpi/...
        finished_asset.png
    drawable-xhdpi/...
        finished_asset.png
```

For more information about how to save resources in the application project, see [Providing Resources](#).

## Remove unnecessary metadata from final assets

Although the Android SDK tools will automatically compress PNGs when packaging application resources into the application binary, a good practice is to remove unnecessary headers and metadata from your PNG assets. Tools such as [OptiPNG](#) or [Pngcrush](#) can ensure that this metadata is removed and that your image asset file sizes are optimized.

# Iconography

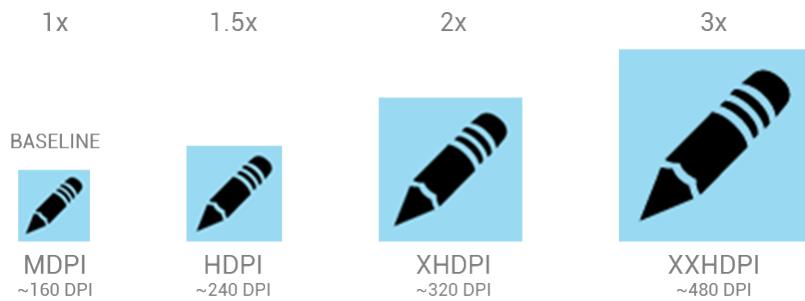
[Previous](#) [Next](#)



An icon is a graphic that takes up a small portion of screen real estate and provides a quick, intuitive representation of an action, a status, or an app.

When you design icons for your app, it's important to keep in mind that your app may be installed on a variety of devices that offer a range of pixel densities, as mentioned in [Devices and Displays](#). But you can make your icons look great on all devices by providing each icon in multiple sizes. When your app runs, Android checks the characteristics of the device screen and loads the appropriate density-specific assets for your app.

Because you will deliver each icon in multiple sizes to support different densities, the design guidelines below refer to the icon dimensions in `dp` units, which are based on the pixel dimensions of a medium-density (MDPI) screen.



So, to create an icon for different densities, you should follow the **2:3:4:6 scaling ratio** between the four primary densities (medium, high, x-high, and xx-high, respectively). For example, consider that the size for a launcher icon is specified to be 48x48 dp. This means the baseline (MDPI) asset is 48x48 px, and the high density (HDPI) asset should be 1.5x the baseline at 72x72 px, and the x-high density (XHDPI) asset should be 2x the baseline at 96x96 px, and so on.

**Note:** Android also supports low-density (LDPI) screens, but you normally don't need to create custom assets at this size because Android effectively down-scales your HDPI assets by 1/2 to match the expected size.

## Launcher

The launcher icon is the visual representation of your app on the Home or All Apps screen. Since the user can change the Home screen's wallpaper, make sure that your launcher icon is clearly visible on any type of background.



## Sizes & scale

- Launcher icons on a mobile device must be **48x48 dp**.
- Launcher icons for display on Google Play must be **512x512 pixels**.

## Proportions

- Full asset, **48x48**

## Style

Use a distinct silhouette. Three-dimensional, front view, with a slight perspective as if viewed from above, so that users perceive some depth.

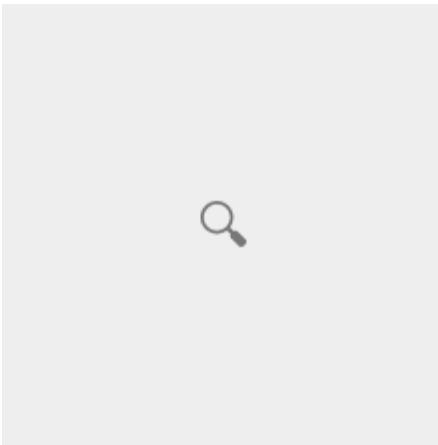


## Action Bar

Action bar icons are graphic buttons that represent the most important actions people can take within your app. Each one should employ a simple metaphor representing a single concept that most people can grasp at a glance.

Pre-defined glyphs should be used for certain common actions such as "refresh" and "share." The download link below provides a package with icons that are scaled for various screen densities and are suitable for use with the Holo Light and Holo Dark themes. The package also includes unstyled icons that you can modify to match your theme, in addition to Adobe® Illustrator® source files for further customization.

[Download the Action Bar Icon Pack](#)



## Sizes & scale

- Action bar icons for phones should be **32x32 dp**.

## Focal area & proportions

- Full asset, **32x32 dp**

Optical square, **24x24 dp**

## Style

Pictographic, flat, not too detailed, with smooth curves or sharp shapes. If the graphic is thin, rotate it 45° left or right to fill the focal space. The thickness of the strokes and negative spaces should be a minimum of 2 dp.

## Colors

Colors: #333333

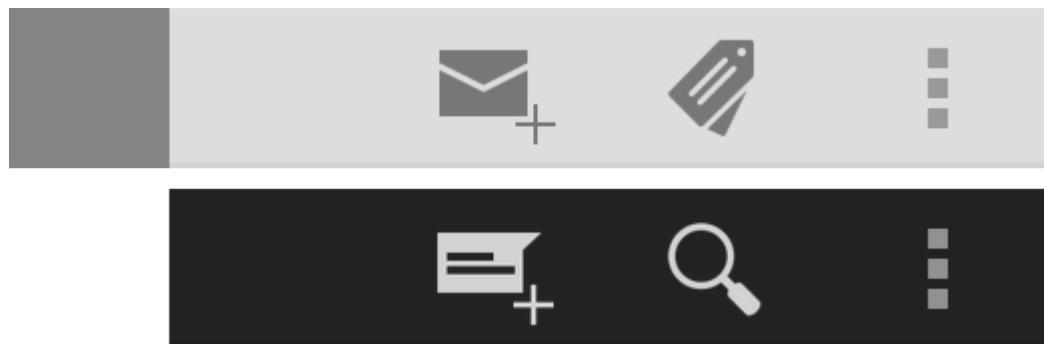
Enabled: **60%** opacity

Disabled: **30%** opacity

Colors: #FFFFFF

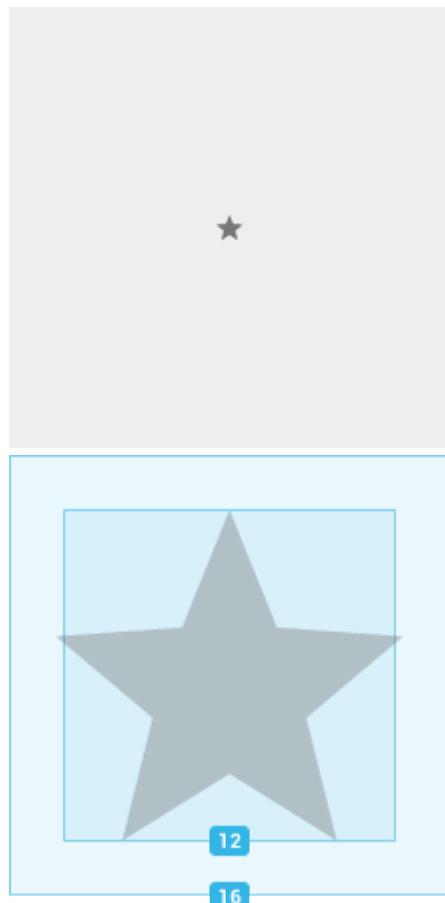
Enabled: **80%** opacity

Disabled: **30%** opacity



## Small / Contextual Icons

Within the body of your app, use small icons to surface actions and/or provide status for specific items. For example, in the Gmail app, each message has a star icon that marks the message as important.





## Sizes & scale

- Small icons should be **16x16 dp**.

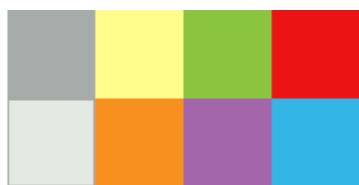
## Focal area & proportions

- Full asset, **16x16 dp**

Optical square, **12x12 dp**

## Style

Neutral, flat, and simple. Filled shapes are easier to see than thin strokes. Use a single visual metaphor so that a user can easily recognize and understand its purpose.



## Colors

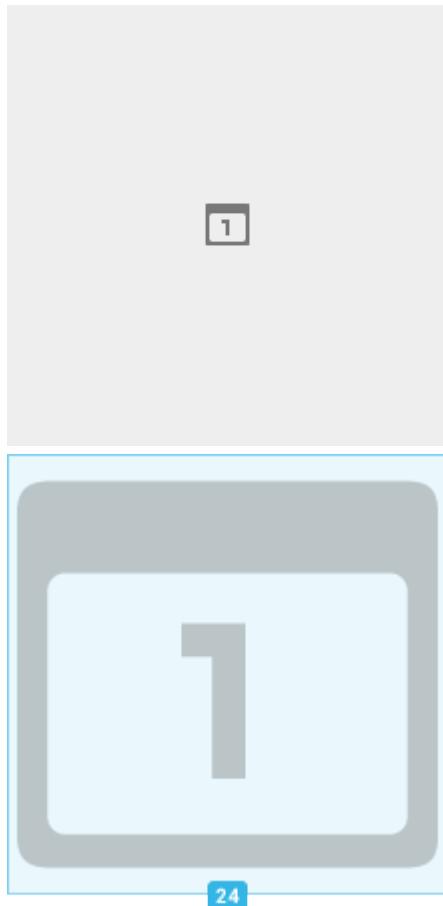
Use non-neutral colors sparingly and with purpose. For example, Gmail uses yellow in the star icon to indicate a bookmarked message. If an icon is actionable, choose a color that contrasts well with the background.

Dec 16  
san biodiesel, commodo        
helvetica aliquip photo

Dec 16  
eimer        
art party, you probably  
hem et officia mustache lo-

## Notification Icons

If your app generates notifications, provide an icon that the system can display in the status bar whenever a new notification is available.





## Sizes & scale

- Notification icons must be **24x24 dp**.

## Focal area & proportions

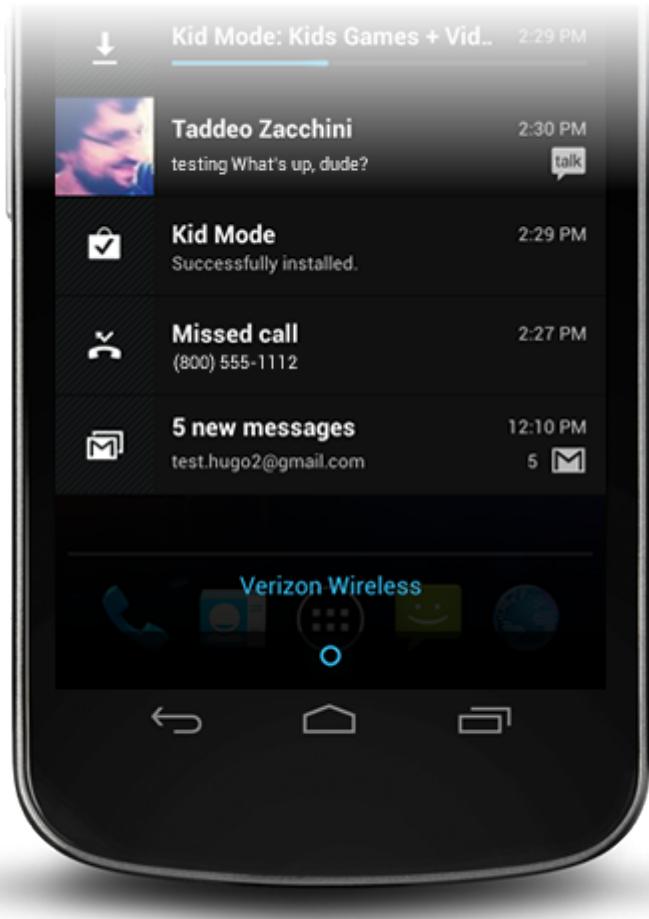
- Full asset, **24x24 dp**  
Optical square, **22x22 dp**

## Style

Keep the style flat and simple, using the same single, visual metaphor as your launcher icon.

## Colors

Notification icons must be entirely white. Also, the system may scale down and/or darken the icons.



## Design Tips

Here are some tips you might find useful as you create icons or other drawable assets for your application. These tips assume you are using Adobe® Photoshop® or a similar raster and vector image-editing program.

### Use vector shapes where possible

Many image-editing programs such as Adobe® Photoshop® allow you to use a combination of vector shapes and raster layers and effects. When possible, use vector shapes so that if the need arises, assets can be scaled up without loss of detail and edge crispness.

Using vectors also makes it easy to align edges and corners to pixel boundaries at smaller resolutions.

### Start with large artboards

Because you will need to create assets for different screen densities, it is best to start your icon designs on large artboards with dimensions that are multiples of the target icon sizes. For example, launcher icons are 48, 72, 96, or 144 pixels wide, depending on screen density (mdpi, hdpi, xhdpi, and xxhdpi, respectively). If you initially draw launcher icons on an 864x864 artboard, it will be easier and cleaner to adjust the icons when you scale the artboard down to the target sizes for final asset creation.

### When scaling, redraw bitmap layers as needed

If you scaled an image up from a bitmap layer, rather than from a vector layer, those layers will need to be redrawn manually to appear crisp at higher densities. For example if a 60x60 circle was painted as a bitmap for mdpi it will need to be repainted as a 90x90 circle for hdpi.

## Use common naming conventions for icon assets

Try to name files so that related assets will group together inside a directory when they are sorted alphabetically. In particular, it helps to use a common prefix for each icon type. For example:

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Note that you are not required to use a shared prefix of any type—doing so is for your convenience only.

## Set up a working space that organizes files by density

Supporting multiple screen densities means you must create multiple versions of the same icon. To help keep the multiple copies of files safe and easier to find, we recommend creating a directory structure in your working space that organizes asset files based on the target density. For example:

```
art/...
    mdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    hdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png
    xhdpi/...
        _pre_production/...
            working_file.psd
            finished_asset.png

xxhdpi/... _pre_production/... working_file.psd finished_asset.png
```

Because the structure in your working space is similar to that of the application, you can quickly determine which assets should be copied to each resources directory. Separating assets by density also helps you detect any variances in filenames across densities, which is important because corresponding assets for different densities must share the same filename.

For comparison, here's the resources directory structure of a typical application:

```
res/...
    drawable-ldpi/...
        finished_asset.png
    drawable-mdpi/...
        finished_asset.png
    drawable-hdpi/...
        finished_asset.png
    drawable-xhdpi/...
        finished_asset.png
```

For more information about how to save resources in the application project, see [Providing Resources](#).

## Remove unnecessary metadata from final assets

Although the Android SDK tools will automatically compress PNGs when packaging application resources into the application binary, a good practice is to remove unnecessary headers and metadata from your PNG assets. Tools such as [OptiPNG](#) or [Pngcrush](#) can ensure that this metadata is removed and that your image asset file sizes are optimized.

# App Widget Design Guidelines

## Quickview

- App Widget layouts should be flexible, resizing to fit their parent container
- As of Android 3.0, app widgets can depict collections of items and provide a representative preview image for the widget gallery
- As of Android 3.1, app widgets can be resizable horizontally and/or vertically
- As of Android 4.0, app widgets have margins automatically applied

## In this document

1. [Standard Widget Anatomy](#)
2. [Designing Widget Layouts and Background Graphics](#)
3. [Using the App Widget Templates Pack](#)

## Downloads

1. [App Widget Templates Pack, v4.0 »](#)

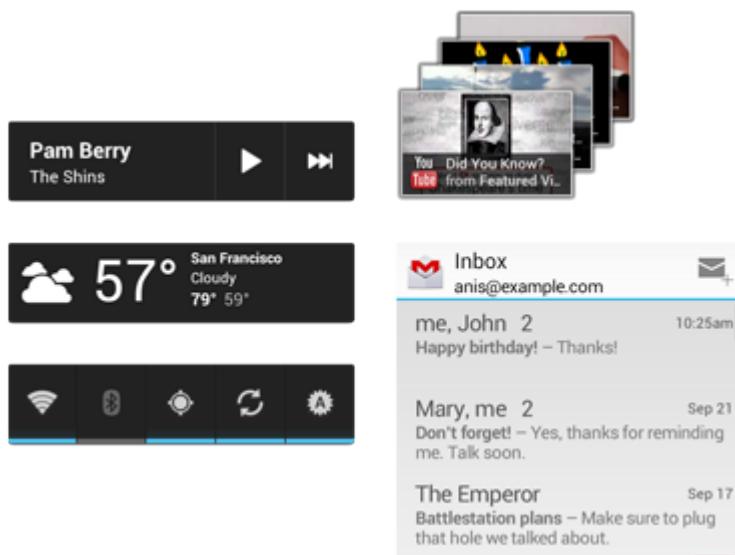
## See also

1. [App Widgets](#)
2. [AppWidgets blog post](#)

## New Guides for App Designers!

Check out the new documents for designers at [Android Design](#).

App widgets (sometimes just "widgets") are a feature introduced in Android 1.5 and vastly improved in Android 3.0 and 3.1. A widget can display an application's most timely or otherwise relevant information at a glance, on a user's Home screen. The standard Android system image includes several widgets, including a widget for the Analog Clock, Music, and other applications.



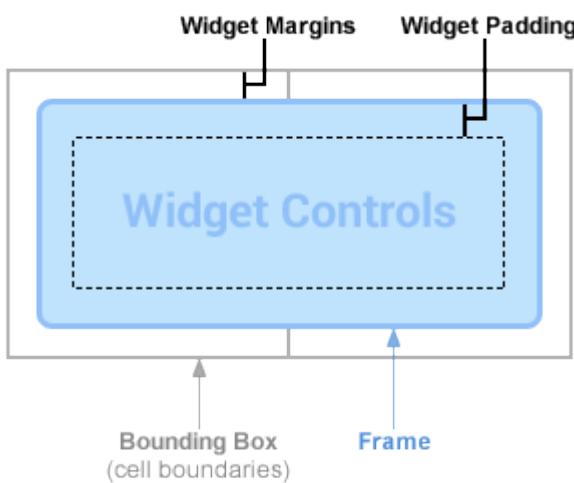
**Figure 1.** Example app widgets in Android 4.0.

This document describes how to design a widget so that it fits graphically with other widgets and with the other elements of the Android Home screen such as launcher icons and shortcuts. It also describes some standards for widget artwork and some widget graphics tips and tricks.

For information about developing widgets, see the [App Widgets](#) section of the *Developer's Guide*.

## Standard Widget Anatomy

Typical Android app widgets have three main components: A bounding box, a frame, and the widget's graphical controls and other elements. App widgets can contain a subset of the View widgets in Android; supported controls include text labels, buttons, and images. For a full list of available Views, see the [Creating the App Widget Layout](#) section in the *Developer's Guide*. Well-designed widgets leave some margins between the edges of the bounding box and the frame, and padding between the inner edges of the frame and the widget's controls.



**Figure 2.** Widgets generally have margins between the bounding box and frame, and padding between the frame and widget controls.

**Note:** As of Android 4.0, app widgets are automatically given margins between the widget frame and the app widget's bounding box to provide better alignment with other widgets and icons on the user's home screen. To take advantage of this strongly recommended behavior, set your application's [targetSdkVersion](#) to 14 or greater.

Widgets designed to fit visually with other widgets on the Home screen take cues from the other elements on the Home screen for alignment; they also use standard shading effects. All of these details are described in this document.

## Determining a size for your widget

Each widget must define a `minWidth` and `minHeight`, indicating the minimum amount of space it should consume by default. When users add a widget to their Home screen, it will generally occupy more than the minimum width and height you specify. Android Home screens offer users a grid of available spaces into which they can place widgets and icons. This grid can vary by device; for example, many handsets offer a 4x4 grid, and tablets can offer a larger, 8x7 grid. **When your widget is added, it will be stretched to occupy the minimum number of cells, horizontally and vertically, required to satisfy its `minWidth` and `minHeight` constraints.** As we discuss in [Designing Widget Layouts and Background Graphics](#) below, using nine-patch backgrounds and flexible layouts for app widgets will allow your widget to gracefully adapt to the device's Home screen grid and remain usable and aesthetically awesome.

While the width and height of a cell—as well as the amount of automatic margins applied to widgets—may vary across devices, you can use the table below to roughly estimate your widget's minimum dimensions, given the desired number of occupied grid cells:

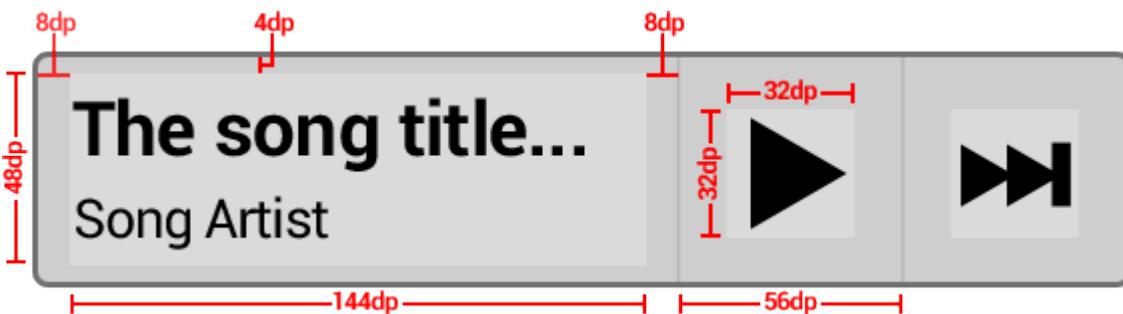
# of Cells	Available Size (dp)
(Columns or Rows) (minWidth or minHeight)	
1	40dp
2	110dp
3	180dp
4	250dp
...	...
$n$	$70 \times n - 30$

It is a good practice to be conservative with `minWidth` and `minHeight`, specifying the minimum size that renders the widget in a good default state. For an example of how to provide a `minWidth` and `minHeight`, suppose you have a music player widget that shows the currently playing song artist and title (vertically stacked), a **Play** button, and a **Next** button:



**Figure 3.** An example music player widget.

Your minimum height should be the height of your two `TextViews` for the artist and title, plus some text margins. Your minimum width should be the minimum usable widths of the **Play** and **Next** buttons, plus the minimum text width (say, the width of 10 characters), plus any horizontal text margins.



**Figure 4.** Example sizes and margins for `minWidth`/`minHeight` calculations. We chose 144dp as an example good minimum width for the text labels.

Example calculations are below:

- $\text{minWidth} = 144\text{dp} + (2 \times 8\text{dp}) + (2 \times 56\text{dp}) = 272\text{dp}$
- $\text{minHeight} = 48\text{dp} + (2 \times 4\text{dp}) = 56\text{dp}$

If there is any inherent content padding in your widget background nine-patch, you should add to `minWidth` and `minHeight` accordingly.

## Resizable widgets

Widgets can be resized horizontally and/or vertically as of Android 3.1, meaning that `minWidth` and `minHeight` effectively become the *default* size for the widget. You can specify the minimum widget size using `minResizeWidth` and `minResizeHeight`; these values should specify the size below which the widget would be illegible or otherwise unusable.

This is generally a preferred feature for collection widgets such as those based on [ListView](#) or [GridView](#).

## Adding margins to your app widget

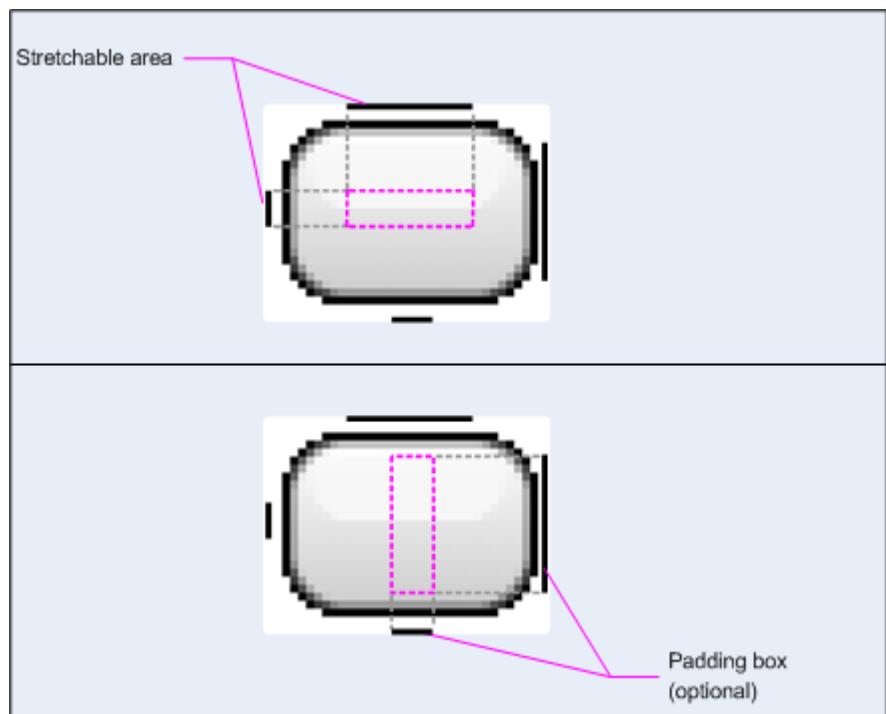
As previously mentioned, Android 4.0 will automatically add small, standard margins to each edge of widgets on the Home screen, for applications that specify a `targetSdkVersion` of 14 or greater. This helps to visually balance the Home screen, and thus **we recommend that you do not add any extra margins outside of your app widget's background shape in Android 4.0.**

It's easy to write a single layout that has custom margins applied for earlier versions of the platform, and has no extra margins for Android 4.0 and greater. See [Adding Margins to App Widgets](#) in the *Developer's Guide* for information on how to achieve this with layout XML.

## Designing Widget Layouts and Background Graphics

Most widgets will have a solid background rectangle or rounded rectangle shape. It is a best practice to define this shape using nine patches; one for each screen density (see [Supporting Multiple Screens](#) for details). Nine-patches can be created with the [draw9patch](#) tool, or simply with a graphics editing program such as Adobe® Photoshop. This will allow the widget background shape to take up the entire available space. The nine-patch should be edge-to-edge with no transparent pixels providing extra margins, save for perhaps a few border pixels for subtle drop shadows or other subtle effects.

**Note:** Just like with controls in activities, you should ensure that interactive controls have distinct visual focused and pressed states using [state list drawables](#).



**Figure 5.** Nine-patch border pixels indicating stretchable regions and content padding.

Some app widgets, such as those using a [StackView](#), have a transparent background. For this case, each individual item in the StackView should use a nine-patch background that is edge-to-edge with little or no border transparent pixels for margins.

For the contents of the widget, you should use flexible layouts such as [RelativeLayout](#), [LinearLayout](#), or [FrameLayout](#). Just as your activity layouts must adapt to different physical screen sizes, widget layouts must adapt to different Home screen grid cell sizes.

Below is an example layout that a music widget showing text information and two buttons can use. It builds upon the previous discussion of adding margins depending on OS version. Note that the most robust and resilient way to add margins to the widget is to wrap the widget frame and contents in a padded [FrameLayout](#).

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/widget_margin">

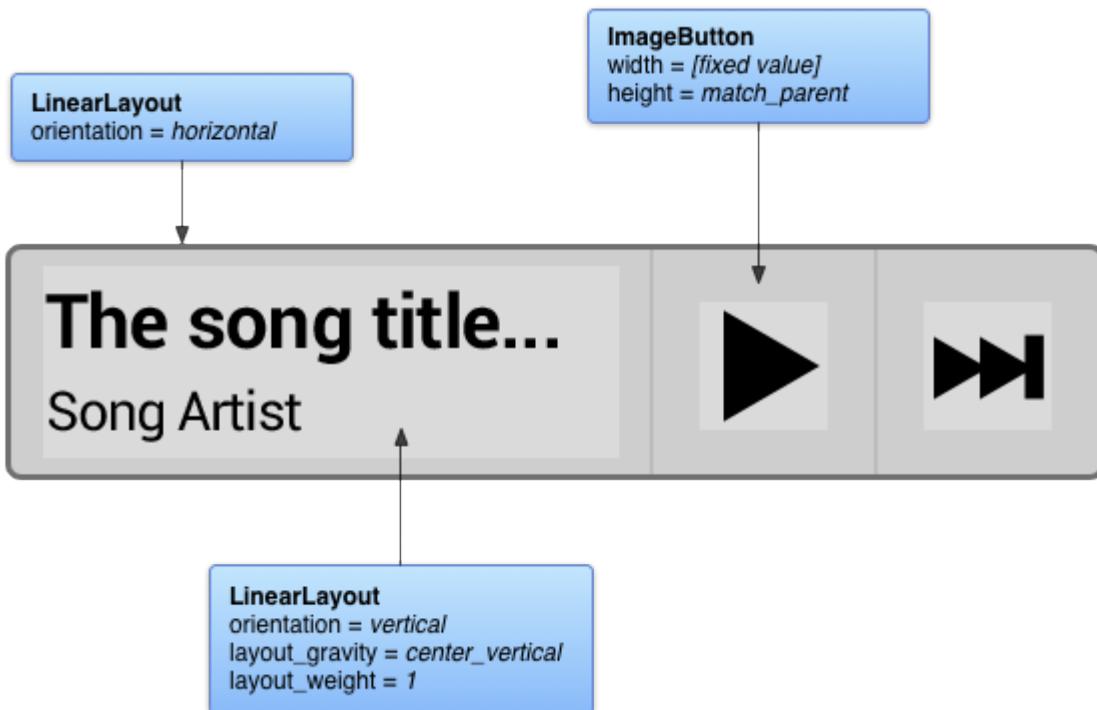
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal"
        android:background="@drawable/my_widget_background">

        <TextView
            android:id="@+id/song_info"
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="1" />

        <Button
            android:id="@+id/play_button"
            android:layout_width="@dimen/my_button_width"
            android:layout_height="match_parent" />

        <Button
            android:id="@+id/skip_button"
            android:layout_width="@dimen/my_button_width"
            android:layout_height="match_parent" />
    </LinearLayout>
</FrameLayout>
```

If you now take a look at the example music widget from the previous section, you can begin to use flexible layouts attributes like so:



**Figure 6.** Excerpt flexible layouts and attributes.

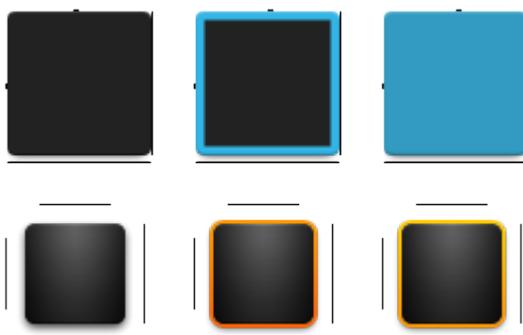
When a user adds the widget to their home screen, on an example Android 4.0 device where each grid cell is  $80\text{dp} \times 100\text{dp}$  in size and 8dp of margins are automatically applied on all sizes, the widget will be stretched, like so:



**Figure 7.** Music widget sitting on an example  $80\text{dp} \times 100\text{dp}$  grid with 8dp of automatic margins added by the system.

## Using the App Widget Templates Pack

When starting to design a new widget, or updating an existing widget, it's a good idea to first look at the widget design templates below. The downloadable package below includes nine-patch background graphics, XML, and source Adobe® Photoshop files for multiple screen densities, OS version widget styles, and widget colors. The template package also contains graphics useful for making your entire widget or parts of your widget (e.g. buttons) interactive.



**Figure 8.** Excerpts from the App Widget Templates Pack (medium-density, dark, Android 4.0/previous styles, default/focused/pressed states).

You can obtain the latest App Widget Templates Pack archive using the link below:

[Download the App Widget Templates Pack for Android 4.0 »](#)

# Appendix

## [Supported Android Media Formats](#)

A list of media codecs included in the Android platform.

## [Intents List: Invoking Google Applications on Android Devices](#)

Intents you can send to invoke Google applications on Android devices.

## [FAQs, Tips, and How-to](#)

How to get things done in Android.

## [Glossary](#)

Glossary of Android terminology.

# Intents List: Invoking Google Applications on Android Devices

For more information about intents, see the [Intents and Intent Filters](#).

The table below lists the intents that your application can send, to invoke Google applications on Android devices in certain ways. For each action/uri pair, the table describes how the receiving Google application handles the intent.

Note that this list is not comprehensive.

Target Application	Intent URI	Intent Action
Browser	http://web_address	VIEW
	https://web_address	Open a browser
	"" (empty string)	WEB_SEARCH
	http://web_address	Opens the file at browser.
Dialer	https://web_address	Calls the entered numbers as defined. Valid examples: <ul style="list-style-type: none"><li>• tel:2125551212</li><li>• tel: (212) 555-1212</li></ul>
	tel: phone_number	CALL
	voicemail:	DIAL
Google Maps	geo:latitude,longitude	Dials (but does not call) the number given (on the phone). Telephone numbers are not described for CALL.
	geo:latitude,longitude?z=zoom	Opens the Maps application or query. The Google Maps API (used) is currently under development.
	geo:0,0?q=my+street+address	
	geo:0,0?q=business+near+city	

The *z* field specifies the zoom level. A value of 1 shows the world at street level, 2 shows the Earth, and so on. Larger zoom levels show smaller areas.

Opens the Street View camera at the specified coordinates (*lat,lng*). A zoom level of 1 shows the entire Earth, and so on. Larger zoom levels show smaller areas.

The *cbll* field is optional. It contains the latitude and longitude of the camera location. The URI is used for Street View camera URLs generated by Google Maps URLs.

*lat* latitude

*lng* longitude

Panorama center point in degrees, expressed clockwise from North.

**Note:** This parameter is optional. It is sent for panoramas.

Panorama center yaw angle in degrees, expressed clockwise from North.

Panorama center pitch angle in degrees, expressed straight down from horizontal.

Panorama center zoom level. Values 2.0 and 4.0 are equivalent to 4x and 8x resolution.

A zoom level of 1 shows the world at street level, 2 shows the Earth, and so on. Larger zoom levels show smaller areas.

FOV (Field of View) in degrees. A value of 360 is equivalent to a full circle.

3 aspect ratio in pixels. This is the aspect ratio of the image that the camera takes. This is useful for panoramic images.

This is the aspect ratio of the image taken by the camera.

An AR value of 1.0 means that the image is square.

This is the lens effect. It is a degree of freedom for portrait-style images.

The next parameter is associated with the camera's focal length.

*mapZoom* is passed to the Street View camera item in the URL.

Google Streetview

google.streetview:cbll=*lat,lng*&cbp=1,*yaw,,pitch,zoom&mz=mapZoom* VIEW

# Glossary

The list below defines some of the basic terminology of the Android platform.

## .apk file

Android application package file. Each Android application is compiled and packaged in a single file that includes all of the application's code (.dex files), resources, assets, and manifest file. The application package file can have any name but *must* use the .apk extension. For example: myExampleAppname.apk. For convenience, an application package file is often referred to as an ".apk".

Related: [Application](#).

## .dex file

Compiled Android application code file.

Android programs are compiled into .dex (Dalvik Executable) files, which are in turn zipped into a single .apk file on the device. .dex files can be created by automatically translating compiled applications written in the Java programming language.

## Action

A description of something that an Intent sender wants done. An action is a string value assigned to an Intent. Action strings can be defined by Android or by a third-party developer. For example, android.intent.action.VIEW for a Web URL, or com.example.rumbler.SHAKE\_PHONE for a custom application to vibrate the phone.

Related: [Intent](#).

## Activity

A single screen in an application, with supporting Java code, derived from the [Activity](#) class. Most commonly, an activity is visibly represented by a full screen window that can receive and handle UI events and perform complex tasks, because of the Window it uses to render its window. Though an Activity is typically full screen, it can also be floating or transparent.

## adb

Android Debug Bridge, a command-line debugging application included with the SDK. It provides tools to browse the device, copy tools on the device, and forward ports for debugging. If you are developing in Eclipse using the ADT Plugin, adb is integrated into your development environment. See [Android Debug Bridge](#) for more information.

## Application

From a component perspective, an Android application consists of one or more activities, services, listeners, and intent receivers. From a source file perspective, an Android application consists of code, resources, assets, and a single manifest. During compilation, these files are packaged in a single file called an application package file (.apk).

Related: [.apk](#), [Activity](#)

## Canvas

A drawing surface that handles compositing of the actual bits against a Bitmap or Surface object. It has methods for standard computer drawing of bitmaps, lines, circles, rectangles, text, and so on, and is bound to a Bitmap or Surface. Canvas is the simplest, easiest way to draw 2D objects on the screen. However, it does not support hardware acceleration, as OpenGL ES does. The base class is [Canvas](#).

Related: [Drawable](#), [OpenGL ES](#).

## **Content Provider**

A data-abstraction layer that you can use to safely expose your application's data to other applications. A content provider is built on the [ContentProvider](#) class, which handles content query strings of a specific format to return data in a specific format. See [Content Providers](#) topic for more information.

Related: [URI Usage in Android](#)

## **Dalvik**

The Android platform's virtual machine. The Dalvik VM is an interpreter-only virtual machine that executes files in the Dalvik Executable (.dex) format, a format that is optimized for efficient storage and memory-mappable execution. The virtual machine is register-based, and it can run classes compiled by a Java language compiler that have been transformed into its native format using the included "dx" tool. The VM runs on top of Posix-compliant operating systems, which it relies on for underlying functionality (such as threading and low level memory management). The Dalvik core class library is intended to provide a familiar development base for those used to programming with Java Standard Edition, but it is geared specifically to the needs of a small mobile device.

## **DDMS**

Dalvik Debug Monitor Service, a GUI debugging application included with the SDK. It provides screen capture, log dump, and process examination capabilities. If you are developing in Eclipse using the ADT Plugin, DDMS is integrated into your development environment. See [Using DDMS](#) to learn more about the program.

## **Dialog**

A floating window that acts as a lightweight form. A dialog can have button controls only and is intended to perform a simple action (such as button choice) and perhaps return a value. A dialog is not intended to persist in the history stack, contain complex layout, or perform complex actions. Android provides a default simple dialog for you with optional buttons, though you can define your own dialog layout. The base class for dialogs is [Dialog](#).

Related: [Activity](#).

## **Drawable**

A compiled visual resource that can be used as a background, title, or other part of the screen. A drawable is typically loaded into another UI element, for example as a background image. A drawable is not able to receive events, but does assign various other properties such as "state" and scheduling, to enable subclasses such as animation objects or image libraries. Many drawable objects are loaded from drawable resource files — xml or bitmap files that describe the image. Drawable resources are compiled into subclasses of [android.graphics.drawable](#). For more information about drawables and other resources, see [Resources](#).

Related: [Resources](#), [Canvas](#)

## **Intent**

An message object that you can use to launch or communicate with other applications/activities asynchronously. An Intent object is an instance of [Intent](#). It includes several criteria fields that you can supply, to determine what application/activity receives the Intent and what the receiver does when handling the Intent. Available criteria include include the desired action, a category, a data string, the MIME type of the data, a handling class, and others. An application sends an Intent to the Android system, rather than sending it directly to another application/activity. The application can send the Intent to a single target application or it can send it as a broadcast, which can in turn be handled by multiple applications sequentially. The Android system is responsible for resolving the best-available receiver for each Intent, based on the criteria supplied in the Intent and the Intent Filters defined by other applications. For more information, see [Intents and Intent Filters](#).

Related: [Intent Filter](#), [Broadcast Receiver](#).

## Intent Filter

A filter object that an application declares in its manifest file, to tell the system what types of Intents each of its components is willing to accept and with what criteria. Through an intent filter, an application can express interest in specific data types, Intent actions, URI formats, and so on. When resolving an Intent, the system evaluates all of the available intent filters in all applications and passes the Intent to the application/activity that best matches the Intent and criteria. For more information, see [Intents and Intent Filters](#).

Related: [Intent](#), [Broadcast Receiver](#).

## Broadcast Receiver

An application class that listens for Intents that are broadcast, rather than being sent to a single target application/activity. The system delivers a broadcast Intent to all interested broadcast receivers, which handle the Intent sequentially.

Related: [Intent](#), [Intent Filter](#).

## Layout Resource

An XML file that describes the layout of an Activity screen.

Related: [Resources](#)

## Manifest File

An XML file that each application must define, to describe the application's package name, version, components (activities, intent filters, services), imported libraries, and describes the various activities, and so on. See [The AndroidManifest.xml File](#) for complete information.

## Nine-patch / 9-patch / Ninepatch image

A resizeable bitmap resource that can be used for backgrounds or other images on the device. See [Nine-Patch Stretchable Image](#) for more information.

Related: [Resources](#).

## OpenGL ES

Android provides OpenGL ES libraries that you can use for fast, complex 3D images. It is harder to use than a Canvas object, but better for 3D objects. The [android.opengl](#) and [javax.microedition.khronos.opengles](#) packages expose OpenGL ES functionality.

Related: [Canvas](#), [Surface](#)

## Resources

Nonprogrammatic application components that are external to the compiled application code, but which can be loaded from application code using a well-known reference format. Android supports a variety of resource types, but a typical application's resources would consist of UI strings, UI layout components, graphics or other media files, and so on. An application uses resources to efficiently support localization and varied device profiles and states. For example, an application would include a separate set of resources for each supported local or device type, and it could include layout resources that are specific to the current screen orientation (landscape or portrait). For more information about resources, see [Resources and Assets](#). The resources of an application are always stored in the `res/*` subfolders of the project.

## Service

An object of class [Service](#) that runs in the background (without any UI presence) to perform various persistent actions, such as playing music or monitoring network activity.

Related: [Activity](#)

## Surface

An object of type [Surface](#) representing a block of memory that gets composited to the screen. A Surface holds a Canvas object for drawing, and provides various helper methods to draw layers and resize the surface. You should not use this class directly; use [SurfaceView](#) instead.

Related: [Canvas](#)

## SurfaceView

A View object that wraps a Surface for drawing, and exposes methods to specify its size and format dynamically. A SurfaceView provides a way to draw independently of the UI thread for resource-intensive operations (such as games or camera previews), but it uses extra memory as a result. SurfaceView supports both Canvas and OpenGL ES graphics. The base class is [SurfaceView](#).

Related: [Surface](#)

## Theme

A set of properties (text size, background color, and so on) bundled together to define various default display settings. Android provides a few standard themes, listed in [R.style](#) (starting with "Theme\_").

## URIs in Android

Android uses URI strings as the basis for requesting data in a content provider (such as to retrieve a list of contacts) and for requesting actions in an Intent (such as opening a Web page in a browser). The URI scheme and format is specialized according to the type of use, and an application can handle specific URI schemes and strings in any way it wants. Some URI schemes are reserved by system components. For example, requests for data from a content provider must use the `content://`. In an Intent, a URI using an `http://` scheme will be handled by the browser.

## View

An object that draws to a rectangular area on the screen and handles click, keystroke, and other interaction events. A View is a base class for most layout components of an Activity or Dialog screen (text boxes, windows, and so on). It receives calls from its parent object (see viewgroup, below) to draw itself, and informs its parent object about where and how big it would like to be (which may or may not be respected by the parent). For more information, see [View](#).

Related: [Viewgroup](#), [Widget](#)

## Viewgroup

A container object that groups a set of child Views. The viewgroup is responsible for deciding where child views are positioned and how large they can be, as well as for calling each to draw itself when appropriate. Some viewgroups are invisible and are for layout only, while others have an intrinsic UI (for instance, a scrolling list box). Viewgroups are all in the [widget](#) package, but extend [ViewGroup](#).

Related: [View](#)

## Widget

One of a set of fully implemented View subclasses that render form elements and other UI components, such as a text box or popup menu. Because a widget is fully implemented, it handles measuring and drawing itself and responding to screen events. Widgets are all in the [android.widget](#) package.

## Window

In an Android application, an object derived from the abstract class [Window](#) that specifies the elements of a generic window, such as the look and feel (title bar text, location and content of menus, and so on). Dia-

log and Activity use an implementation of this class to render a window. You do not need to implement this class or use windows in your application.

# Activities

## Quickview

- An activity provides a user interface for a single screen in your application
- Activities can move into the background and then be resumed with their state restored

## In this document

1. [Creating an Activity](#)
  1. [Implementing a user interface](#)
  2. [Declaring the activity in the manifest](#)
2. [Starting an Activity](#)
  1. [Starting an activity for a result](#)
3. [Shutting Down an Activity](#)
4. [Managing the Activity Lifecycle](#)
  1. [Implementing the lifecycle callbacks](#)
  2. [Saving activity state](#)
  3. [Handling configuration changes](#)
  4. [Coordinating activities](#)

## Key classes

1. [Activity](#)

## See also

1. [Tasks and Back Stack](#)

An [Activity](#) is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. The back stack abides to the basic "last in, first out" stack mechanism, so, when the user is done with the current activity and presses the *Back* button, it is popped from the stack (and destroyed) and the previous activity resumes. (The back stack is discussed more in the [Tasks and Back Stack](#) document.)

When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides you the opportunity to perform specific work that's appropriate to that state change. For instance, when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted. These state transitions are all part of the activity lifecycle.

The rest of this document discusses the basics of how to build and use an activity, including a complete discussion of how the activity lifecycle works, so you can properly manage the transition between various activity states.

## Creating an Activity

To create an activity, you must create a subclass of [Activity](#) (or an existing subclass of it). In your subclass, you need to implement callback methods that the system calls when the activity transitions between various states of its lifecycle, such as when the activity is being created, stopped, resumed, or destroyed. The two most important callback methods are:

### [onCreate\(\)](#)

You must implement this method. The system calls this when creating your activity. Within your implementation, you should initialize the essential components of your activity. Most importantly, this is where you must call [setContentView\(\)](#) to define the layout for the activity's user interface.

### [onPause\(\)](#)

The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

There are several other lifecycle callback methods that you should use in order to provide a fluid user experience between activities and handle unexpected interruptions that cause your activity to be stopped and even destroyed. All of the lifecycle callback methods are discussed later, in the section about [Managing the Activity Lifecycle](#).

## Implementing a user interface

The user interface for an activity is provided by a hierarchy of views—objects derived from the [View](#) class. Each view controls a particular rectangular space within the activity's window and can respond to user interaction. For example, a view might be a button that initiates an action when the user touches it.

Android provides a number of ready-made views that you can use to design and organize your layout. "Widgets" are views that provide a visual (and interactive) elements for the screen, such as a button, text field, checkbox, or just an image. "Layouts" are views derived from [ViewGroup](#) that provide a unique layout model for its child views, such as a linear layout, a grid layout, or relative layout. You can also subclass the [View](#) and [ViewGroup](#) classes (or existing subclasses) to create your own widgets and layouts and apply them to your activity layout.

The most common way to define a layout using views is with an XML layout file saved in your application resources. This way, you can maintain the design of your user interface separately from the source code that defines the activity's behavior. You can set the layout as the UI for your activity with [setContentView\(\)](#), passing the resource ID for the layout. However, you can also create new [Views](#) in your activity code and build a view hierarchy by inserting new [Views](#) into a [ViewGroup](#), then use that layout by passing the root [ViewGroup](#) to [setContentView\(\)](#).

For information about creating a user interface, see the [User Interface](#) documentation.

## Declaring the activity in the manifest

You must declare your activity in the manifest file in order for it to be accessible to the system. To declare your activity, open your manifest file and add an [<activity>](#) element as a child of the [<application>](#) element. For example:

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
...
</manifest >
```

There are several other attributes that you can include in this element, to define properties such as the label for the activity, an icon for the activity, or a theme to style the activity's UI. The [android:name](#) attribute is the only required attribute—it specifies the class name of the activity. Once you publish your application, you should not change this name, because if you do, you might break some functionality, such as application shortcuts (read the blog post, [Things That Cannot Change](#)).

See the [<activity>](#) element reference for more information about declaring your activity in the manifest.

## Using intent filters

An [<activity>](#) element can also specify various intent filters—using the [<intent-filter>](#) element—in order to declare how other application components may activate it.

When you create a new application using the Android SDK tools, the stub activity that's created for you automatically includes an intent filter that declares the activity responds to the "main" action and should be placed in the "launcher" category. The intent filter looks like this:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

The [<action>](#) element specifies that this is the "main" entry point to the application. The [<category>](#) element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).

If you intend for your application to be self-contained and not allow other applications to activate its activities, then you don't need any other intent filters. Only one activity should have the "main" action and "launcher" category, as in the previous example. Activities that you don't want to make available to other applications should have no intent filters and you can start them yourself using explicit intents (as discussed in the following section).

However, if you want your activity to respond to implicit intents that are delivered from other applications (and your own), then you must define additional intent filters for your activity. For each type of intent to which you want to respond, you must include an [<intent-filter>](#) that includes an [<action>](#) element and, optionally, a [<category>](#) element and/or a [<data>](#) element. These elements specify the type of intent to which your activity can respond.

For more information about how your activities can respond to intents, see the [Intents and Intent Filters](#) document.

# Starting an Activity

You can start another activity by calling `startActivity()`, passing it an `Intent` that describes the activity you want to start. The intent specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application). An intent can also carry small amounts of data to be used by the activity that is started.

When working within your own application, you'll often need to simply launch a known activity. You can do so by creating an intent that explicitly defines the activity you want to start, using the class name. For example, here's how one activity starts another activity named `SignInActivity`:

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

However, your application might also want to perform some action, such as send an email, text message, or status update, using data from your activity. In this case, your application might not have its own activities to perform such actions, so you can instead leverage the activities provided by other applications on the device, which can perform the actions for you. This is where intents are really valuable—you can create an intent that describes an action you want to perform and the system launches the appropriate activity from another application. If there are multiple activities that can handle the intent, then the user can select which one to use. For example, if you want to allow the user to send an email message, you can create the following intent:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

The `EXTRA_EMAIL` extra added to the intent is a string array of email addresses to which the email should be sent. When an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form. In this situation, the email application's activity starts and when the user is done, your activity resumes.

## Starting an activity for a result

Sometimes, you might want to receive a result from the activity that you start. In that case, start the activity by calling `startActivityForResult()` (instead of `startActivity()`). To then receive the result from the subsequent activity, implement the `onActivityResult()` callback method. When the subsequent activity is done, it returns a result in an `Intent` to your `onActivityResult()` method.

For example, perhaps you want the user to pick one of their contacts, so your activity can do something with the information in that contact. Here's how you can create such an intent and handle the result:

```
private void pickContact() {
    // Create an intent to "pick" a contact, as defined by the content provider
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode == PICK_CONTACT_REQUEST)
        // Perform a query to the contact's content provider for the contact's
        Cursor cursor = getContentResolver().query(data.getData(),
```

```

        new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not empty
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Do something with the selected contact's name...
        }
    }
}

```

This example shows the basic logic you should use in your [onActivityResult\(\)](#) method in order to handle an activity result. The first condition checks whether the request was successful—if it was, then the `resultCode` will be [RESULT\\_OK](#)—and whether the request to which this result is responding is known—in this case, the `requestCode` matches the second parameter sent with [startActivityForResult\(\)](#). From there, the code handles the activity result by querying the data returned in an [Intent](#) (the `data` parameter).

What happens is, a [ContentResolver](#) performs a query against a content provider, which returns a [Cursor](#) that allows the queried data to be read. For more information, see the [Content Providers](#) document.

For more information about using intents, see the [Intents and Intent Filters](#) document.

## Shutting Down an Activity

You can shut down an activity by calling its [finish\(\)](#) method. You can also shut down a separate activity that you previously started by calling [finishActivity\(\)](#).

**Note:** In most cases, you should not explicitly finish an activity using these methods. As discussed in the following section about the activity lifecycle, the Android system manages the life of an activity for you, so you do not need to finish your own activities. Calling these methods could adversely affect the expected user experience and should only be used when you absolutely do not want the user to return to this instance of the activity.

## Managing the Activity Lifecycle

Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application. The lifecycle of an activity is directly affected by its association with other activities, its task and back stack.

An activity can exist in essentially three states:

### **Resumed**

The activity is in the foreground of the screen and has user focus. (This state is also sometimes referred to as "running".)

### **Paused**

Another activity is in the foreground and has focus, but this one is still visible. That is, another activity is visible on top of this one and that activity is partially transparent or doesn't cover the entire screen. A paused activity is completely alive (the [Activity](#) object is retained in memory, it maintains all state and member information, and remains attached to the window manager), but can be killed by the system in extremely low memory situations.

### **Stopped**

The activity is completely obscured by another activity (the activity is now in the "background"). A stopped activity is also still alive (the [Activity](#) object is retained in memory, it maintains all state and

member information, but is *not* attached to the window manager). However, it is no longer visible to the user and it can be killed by the system when memory is needed elsewhere.

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its [finish\(\)](#) method), or simply killing its process. When the activity is opened again (after being finished or killed), it must be created all over.

## Implementing the lifecycle callbacks

When an activity transitions into and out of the different states described above, it is notified through various callback methods. All of the callback methods are hooks that you can override to do appropriate work when the state of your activity changes. The following skeleton activity includes each of the fundamental lifecycle methods:

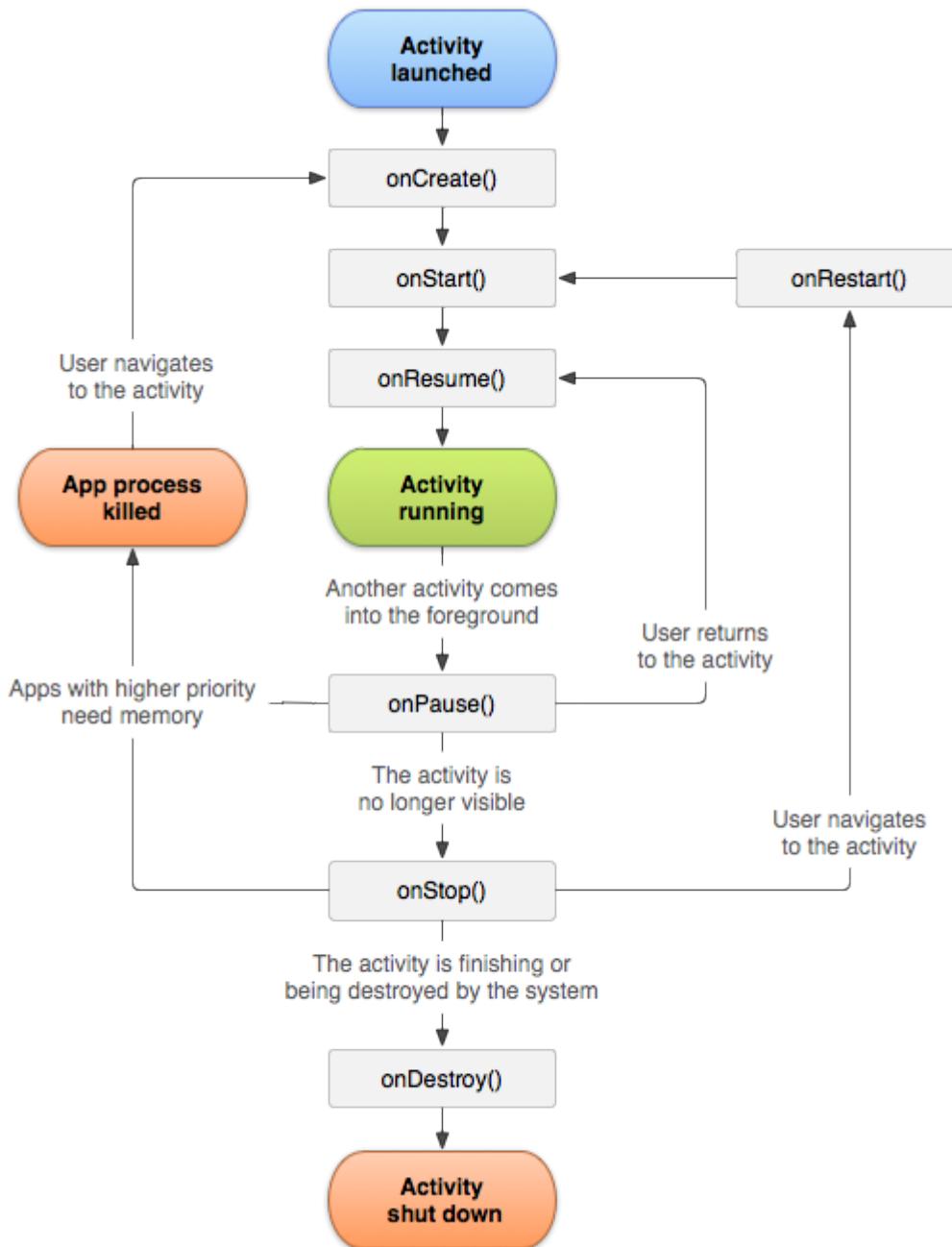
```
public class ExampleActivity extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        // The activity is being created.  
    }  
    @Override  
    protected void onStart\(\) {  
        super.onStart();  
        // The activity is about to become visible.  
    }  
    @Override  
    protected void onResume\(\) {  
        super.onResume();  
        // The activity has become visible (it is now "resumed").  
    }  
    @Override  
    protected void onPause\(\) {  
        super.onPause();  
        // Another activity is taking focus (this activity is about to be "paused").  
    }  
    @Override  
    protected void onStop\(\) {  
        super.onStop();  
        // The activity is no longer visible (it is now "stopped").  
    }  
    @Override  
    protected void onDestroy\(\) {  
        super.onDestroy();  
        // The activity is about to be destroyed.  
    }  
}
```

**Note:** Your implementation of these lifecycle methods must always call the superclass implementation before doing any work, as shown in the examples above.

Taken together, these methods define the entire lifecycle of an activity. By implementing these methods, you can monitor three nested loops in the activity lifecycle:

- The **entire lifetime** of an activity happens between the call to `onCreate()` and the call to `onDestroy()`. Your activity should perform setup of "global" state (such as defining layout) in `onCreate()`, and release all remaining resources in `onDestroy()`. For example, if your activity has a thread running in the background to download data from the network, it might create that thread in `onCreate()` and then stop the thread in `onDestroy()`.
- The **visible lifetime** of an activity happens between the call to `onStart()` and the call to `onStop()`. During this time, the user can see the activity on-screen and interact with it. For example, `onStop()` is called when a new activity starts and this one is no longer visible. Between these two methods, you can maintain resources that are needed to show the activity to the user. For example, you can register a `BroadcastReceiver` in `onStart()` to monitor changes that impact your UI, and unregister it in `onStop()` when the user can no longer see what you are displaying. The system might call `onStart()` and `onStop()` multiple times during the entire lifetime of the activity, as the activity alternates between being visible and hidden to the user.
- The **foreground lifetime** of an activity happens between the call to `onResume()` and the call to `onPause()`. During this time, the activity is in front of all other activities on screen and has user input focus. An activity can frequently transition in and out of the foreground—for example, `onPause()` is called when the device goes to sleep or when a dialog appears. Because this state can transition often, the code in these two methods should be fairly lightweight in order to avoid slow transitions that make the user wait.

Figure 1 illustrates these loops and the paths an activity might take between states. The rectangles represent the callback methods you can implement to perform operations when the activity transitions between states.



**Figure 1.** The activity lifecycle.

The same lifecycle callback methods are listed in table 1, which describes each of the callback methods in more detail and locates each one within the activity's overall lifecycle, including whether the system can kill the activity after the callback method completes.

**Table 1.** A summary of the activity lifecycle's callback methods.

Method	Description	Killable after?	Next
<a href="#">onCreate()</a>	Called when the activity is first created. This is where you should do all of your normal static set up — create views, bind data to lists, and so on. This method is passed a Bundle object containing the activity's previous state, if that state was captured (see <a href="#">Saving Activity State</a> , later).	No	<a href="#">onStart()</a>

Method	Description	Killable after?	Next
	Always followed by <code>onStart()</code> .		
<code>onRestart()</code>	Called after the activity has been stopped, just prior to it being started again.  Always followed by <code>onStart()</code>	No	<code>onStart()</code>
<code>onStart()</code>	Called just before the activity becomes visible to the user.  Followed by <code>onResume()</code> if the activity comes to the foreground, or <code>onStop()</code> if it becomes hidden.	No	<code>onResume()</code> or <code>onStop()</code>
<code>onResume()</code>	Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it.  Always followed by <code>onPause()</code> .	No	<code>onPause()</code>
<code>onPause()</code>	Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.  Followed either by <code>onResume()</code> if the activity returns back to the front, or by <code>onStop()</code> if it becomes invisible to the user.	Yes	<code>onResume()</code> or <code>onStop()</code>
<code>onStop()</code>	Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it.  Followed either by <code>onRestart()</code> if the activity is coming back to interact with the user, or by <code>onDestroy()</code> if this activity is going away.	Yes	<code>onRestart()</code> or <code>onDestroy()</code>
<code>onDestroy()</code>	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing (someone called <code>finish()</code> on	Yes	<i>nothing</i>

Method	Description	Killable after?	Next
	it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the <a href="#">isFinishing()</a> method.		

The column labeled "Killable after?" indicates whether or not the system can kill the process hosting the activity at any time *after the method returns*, without executing another line of the activity's code. Three methods are marked "yes": ([onPause\(\)](#), [onStop\(\)](#), and [onDestroy\(\)](#)). Because [onPause\(\)](#) is the first of the three, once the activity is created, [onPause\(\)](#) is the last method that's guaranteed to be called before the process *can* be killed—if the system must recover memory in an emergency, then [onStop\(\)](#) and [onDestroy\(\)](#) might not be called. Therefore, you should use [onPause\(\)](#) to write crucial persistent data (such as user edits) to storage. However, you should be selective about what information must be retained during [onPause\(\)](#), because any blocking procedures in this method block the transition to the next activity and slow the user experience.

Methods that are marked "No" in the **Killable** column protect the process hosting the activity from being killed from the moment they are called. Thus, an activity is killable from the time [onPause\(\)](#) returns to the time [onResume\(\)](#) is called. It will not again be killable until [onPause\(\)](#) is again called and returns.

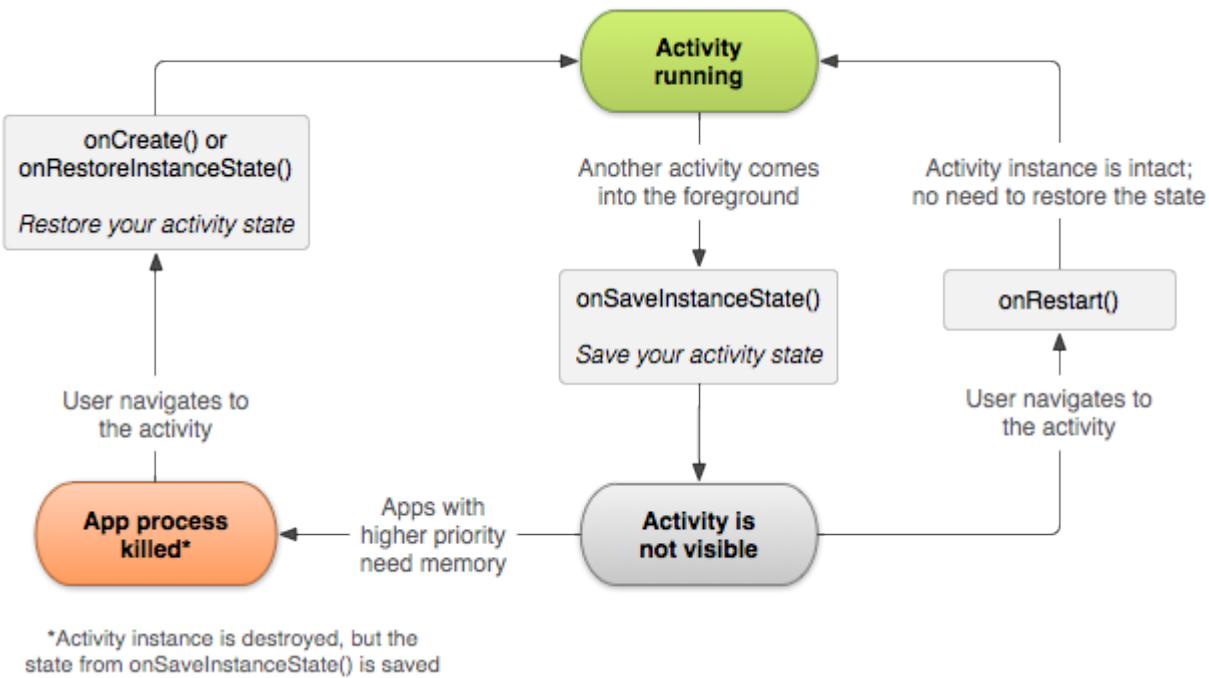
**Note:** An activity that's not technically "killable" by this definition in table 1 might still be killed by the system—but that would happen only in extreme circumstances when there is no other recourse. When an activity might be killed is discussed more in the [Processes and Threading](#) document.

## Saving activity state

The introduction to [Managing the Activity Lifecycle](#) briefly mentions that when an activity is paused or stopped, the state of the activity is retained. This is true because the [Activity](#) object is still held in memory when it is paused or stopped—all information about its members and current state is still alive. Thus, any changes the user made within the activity are retained so that when the activity returns to the foreground (when it "resumes"), those changes are still there.

However, when the system destroys an activity in order to recover memory, the [Activity](#) object is destroyed, so the system cannot simply resume it with its state intact. Instead, the system must recreate the [Activity](#) object if the user navigates back to it. Yet, the user is unaware that the system destroyed the activity and recreated it and, thus, probably expects the activity to be exactly as it was. In this situation, you can ensure that important information about the activity state is preserved by implementing an additional callback method that allows you to save information about the state of your activity: [onSaveInstanceState\(\)](#).

The system calls [onSaveInstanceState\(\)](#) before making the activity vulnerable to destruction. The system passes this method a [Bundle](#) in which you can save state information about the activity as name-value pairs, using methods such as [putString\(\)](#) and [putInt\(\)](#). Then, if the system kills your application process and the user navigates back to your activity, the system recreates the activity and passes the [Bundle](#) to both [onCreate\(\)](#) and [onRestoreInstanceState\(\)](#). Using either of these methods, you can extract your saved state from the [Bundle](#) and restore the activity state. If there is no state information to restore, then the [Bundle](#) passed to you is null (which is the case when the activity is created for the first time).



**Figure 2.** The two ways in which an activity returns to user focus with its state intact: either the activity is destroyed, then recreated and the activity must restore the previously saved state, or the activity is stopped, then resumed and the activity state remains intact.

**Note:** There's no guarantee that `onSaveInstanceState()` will be called before your activity is destroyed, because there are cases in which it won't be necessary to save the state (such as when the user leaves your activity using the *Back* button, because the user is explicitly closing the activity). If the system calls `onSaveInstanceState()`, it does so before `onStop()` and possibly before `onPause()`.

However, even if you do nothing and do not implement `onSaveInstanceState()`, some of the activity state is restored by the `Activity` class's default implementation of `onSaveInstanceState()`. Specifically, the default implementation calls the corresponding `onSaveInstanceState()` method for every `View` in the layout, which allows each view to provide information about itself that should be saved. Almost every widget in the Android framework implements this method as appropriate, such that any visible changes to the UI are automatically saved and restored when your activity is recreated. For example, the `EditText` widget saves any text entered by the user and the `CheckBox` widget saves whether it's checked or not. The only work required by you is to provide a unique ID (with the `android:id` attribute) for each widget you want to save its state. If a widget does not have an ID, then the system cannot save its state.

You can also explicitly stop a view in your layout from saving its state by setting the `android:saveEnabled` attribute to "false" or by calling the `setSaveEnabled()` method. Usually, you should not disable this, but you might if you want to restore the state of the activity UI differently.

Although the default implementation of `onSaveInstanceState()` saves useful information about your activity's UI, you still might need to override it to save additional information. For example, you might need to save member values that changed during the activity's life (which might correlate to values restored in the UI, but the members that hold those UI values are not restored, by default).

Because the default implementation of `onSaveInstanceState()` helps save the state of the UI, if you override the method in order to save additional state information, you should always call the superclass implementation of `onSaveInstanceState()` before doing any work. Likewise, you should also call the superclass implementation of `onRestoreInstanceState()` if you override it, so the default implementation can restore view states.

**Note:** Because `onSaveInstanceState()` is not guaranteed to be called, you should use it only to record the transient state of the activity (the state of the UI)—you should never use it to store persistent data. Instead, you should use `onPause()` to store persistent data (such as data that should be saved to a database) when the user leaves the activity.

A good way to test your application's ability to restore its state is to simply rotate the device so that the screen orientation changes. When the screen orientation changes, the system destroys and recreates the activity in order to apply alternative resources that might be available for the new screen configuration. For this reason alone, it's very important that your activity completely restores its state when it is recreated, because users regularly rotate the screen while using applications.

## Handling configuration changes

Some device configurations can change during runtime (such as screen orientation, keyboard availability, and language). When such a change occurs, Android recreates the running activity (the system calls `onDestroy()`, then immediately calls `onCreate()`). This behavior is designed to help your application adapt to new configurations by automatically reloading your application with alternative resources that you've provided (such as different layouts for different screen orientations and sizes).

If you properly design your activity to handle a restart due to a screen orientation change and restore the activity state as described above, your application will be more resilient to other unexpected events in the activity lifecycle.

The best way to handle such a restart is to save and restore the state of your activity using `onSaveInstanceState()` and `onRestoreInstanceState()` (or `onCreate()`), as discussed in the previous section.

For more information about configuration changes that happen at runtime and how you can handle them, read the guide to [Handling Runtime Changes](#).

## Coordinating activities

When one activity starts another, they both experience lifecycle transitions. The first activity pauses and stops (though, it won't stop if it's still visible in the background), while the other activity is created. In case these activities share data saved to disc or elsewhere, it's important to understand that the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.

The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process and one is starting the other. Here's the order of operations that occur when Activity A starts Activity B:

1. Activity A's `onPause()` method executes.
2. Activity B's `onCreate()`, `onStart()`, and `onResume()` methods execute in sequence. (Activity B now has user focus.)
3. Then, if Activity A is no longer visible on screen, its `onStop()` method executes.

This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another. For example, if you must write to a database when the first activity stops so that the following activity can read it, then you should write to the database during `onPause()` instead of during `onStop()`.

# Services

## Quickview

- A service can run in the background to perform work even while the user is in a different application
- A service can allow other components to bind to it, in order to interact with it and perform interprocess communication
- A service runs in the main thread of the application that hosts it, by default

## In this document

1. [The Basics](#)
  1. [Declaring a service in the manifest](#)
2. [Creating a Started Service](#)
  1. [Extending the IntentService class](#)
  2. [Extending the Service class](#)
  3. [Starting a service](#)
  4. [Stopping a service](#)
3. [Creating a Bound Service](#)
4. [Sending Notifications to the User](#)
5. [Running a Service in the Foreground](#)
6. [Managing the Lifecycle of a Service](#)
  1. [Implementing the lifecycle callbacks](#)

## Key classes

1. [Service](#)
2. [IntentService](#)

## Samples

1. [ServiceStartArguments](#)
2. [LocalService](#)

## See also

1. [Bound Services](#)

A [Service](#) is an application component that can perform long-running operations in the background and does not provide a user interface. Another application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

A service can essentially take two forms:

### Started

A service is "started" when an application component (such as an activity) starts it by calling [startService\(\)](#). Once started, a service can run in the background indefinitely, even if the component that started

it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should stop itself.

## Bound

A service is "bound" when an application component binds to it by calling [bindService\(\)](#). A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Although this documentation generally discusses these two types of services separately, your service can work both ways—it can be started (to run indefinitely) and also allow binding. It's simply a matter of whether you implement a couple callback methods: [onStartCommand\(\)](#) to allow components to start it and [onBind\(\)](#) to allow binding.

Regardless of whether your application is started, bound, or both, any application component can use the service (even from a separate application), in the same way that any component can use an activity—by starting it with an [Intent](#). However, you can declare the service as private, in the manifest file, and block access from other applications. This is discussed more in the section about [Declaring the service in the manifest](#).

**Caution:** A service runs in the main thread of its hosting process—the service does **not** create its own thread and does **not** run in a separate process (unless you specify otherwise). This means that, if your service is going to do any CPU intensive work or blocking operations (such as MP3 playback or networking), you should create a new thread within the service to do that work. By using a separate thread, you will reduce the risk of Application Not Responding (ANR) errors and the application's main thread can remain dedicated to user interaction with your activities.

# The Basics

## Should you use a service or a thread?

A service is simply a component that can run in the background even when the user is not interacting with your application. Thus, you should create a service only if that is what you need.

If you need to perform work outside your main thread, but only while the user is interacting with your application, then you should probably instead create a new thread and not a service. For example, if you want to play some music, but only while your activity is running, you might create a thread in [onCreate\(\)](#), start running it in [onStart\(\)](#), then stop it in [onStop\(\)](#). Also consider using [AsyncTask](#) or [HandlerThread](#), instead of the traditional [Thread](#) class. See the [Processes and Threading](#) document for more information about threads.

Remember that if you do use a service, it still runs in your application's main thread by default, so you should still create a new thread within the service if it performs intensive or blocking operations.

To create a service, you must create a subclass of [Service](#) (or one of its existing subclasses). In your implementation, you need to override some callback methods that handle key aspects of the service lifecycle and provide a mechanism for components to bind to the service, if appropriate. The most important callback methods you should override are:

### [onStartCommand\(\)](#)

The system calls this method when another component, such as an activity, requests that the service be started, by calling [startService\(\)](#). Once this method executes, the service is started and can run in

the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is done, by calling [stopSelf\(\)](#) or [stopService\(\)](#). (If you only want to provide binding, you don't need to implement this method.)

#### [onBind\(\)](#)

The system calls this method when another component wants to bind with the service (such as to perform RPC), by calling [bindService\(\)](#). In your implementation of this method, you must provide an interface that clients use to communicate with the service, by returning an [IBinder](#). You must always implement this method, but if you don't want to allow binding, then you should return null.

#### [onCreate\(\)](#)

The system calls this method when the service is first created, to perform one-time setup procedures (before it calls either [onStartCommand\(\)](#) or [onBind\(\)](#)). If the service is already running, this method is not called.

#### [onDestroy\(\)](#)

The system calls this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc. This is the last call the service receives.

If a component starts the service by calling [startService\(\)](#) (which results in a call to [onStartCommand\(\)](#)), then the service remains running until it stops itself with [stopSelf\(\)](#) or another component stops it by calling [stopService\(\)](#).

If a component calls [bindService\(\)](#) to create the service (and [onStartCommand\(\)](#) is *not* called), then the service runs only as long as the component is bound to it. Once the service is unbound from all clients, the system destroys it.

The Android system will force-stop a service only when memory is low and it must recover system resources for the activity that has user focus. If the service is bound to an activity that has user focus, then it's less likely to be killed, and if the service is declared to [run in the foreground](#) (discussed later), then it will almost never be killed. Otherwise, if the service was started and is long-running, then the system will lower its position in the list of background tasks over time and the service will become highly susceptible to killing—if your service is started, then you must design it to gracefully handle restarts by the system. If the system kills your service, it restarts it as soon as resources become available again (though this also depends on the value you return from [onStartCommand\(\)](#), as discussed later). For more information about when the system might destroy a service, see the [Processes and Threading](#) document.

In the following sections, you'll see how you can create each type of service and how to use it from other application components.

## Declaring a service in the manifest

Like activities (and other components), you must declare all services in your application's manifest file.

To declare your service, add a [`<service>`](#) element as a child of the [`<application>`](#) element. For example:

```
<manifest ... >
  ...
  <application ... >
    <service android:name=".ExampleService" />
  ...

```

```
</application>  
</manifest>
```

There are other attributes you can include in the [`<service>`](#) element to define properties such as permissions required to start the service and the process in which the service should run. The [`android:name`](#) attribute is the only required attribute—it specifies the class name of the service. Once you publish your application, you should not change this name, because if you do, you might break some functionality where explicit intents are used to reference your service (read the blog post, [Things That Cannot Change](#)).

See the [`<service>`](#) element reference for more information about declaring your service in the manifest.

Just like an activity, a service can define intent filters that allow other components to invoke the service using implicit intents. By declaring intent filters, components from any application installed on the user's device can potentially start your service if your service declares an intent filter that matches the intent another application passes to [`startService\(\)`](#).

If you plan on using your service only locally (other applications do not use it), then you don't need to (and should not) supply any intent filters. Without any intent filters, you must start the service using an intent that explicitly names the service class. More information about [starting a service](#) is discussed below.

Additionally, you can ensure that your service is private to your application only if you include the [`an-  
droid:exported`](#) attribute and set it to "false". This is effective even if your service supplies intent filters.

For more information about creating intent filters for your service, see the [Intents and Intent Filters](#) document.

## Creating a Started Service

A started service is one that another component starts by calling [`startService\(\)`](#), resulting in a call to the service's [`onStartCommand\(\)`](#) method.

When a service is started, it has a lifecycle that's independent of the component that started it and the service can run in the background indefinitely, even if the component that started it is destroyed. As such, the service should stop itself when its job is done by calling [`stopSelf\(\)`](#), or another component can stop it by calling [`stopService\(\)`](#).

An application component such as an activity can start the service by calling [`startService\(\)`](#) and passing an [`Intent`](#) that specifies the service and includes any data for the service to use. The service receives this [`Intent`](#) in the [`onStartCommand\(\)`](#) method.

For instance, suppose an activity needs to save some data to an online database. The activity can start a companion service and deliver it the data to save by passing an intent to [`startService\(\)`](#). The service receives the intent in [`onStartCommand\(\)`](#), connects to the Internet and performs the database transaction. When the transaction is done, the service stops itself and it is destroyed.

**Caution:** A service runs in the same process as the application in which it is declared and in the main thread of that application, by default. So, if your service performs intensive or blocking operations while the user interacts with an activity from the same application, the service will slow down activity performance. To avoid impacting application performance, you should start a new thread inside the service.

Traditionally, there are two classes you can extend to create a started service:

## Service

This is the base class for all services. When you extend this class, it's important that you create a new thread in which to do all the service's work, because the service uses your application's main thread, by default, which could slow the performance of any activity your application is running.

## IntentService

This is a subclass of [Service](#) that uses a worker thread to handle all start requests, one at a time. This is the best option if you don't require that your service handle multiple requests simultaneously. All you need to do is implement [onHandleIntent\(\)](#), which receives the intent for each start request so you can do the background work.

The following sections describe how you can implement your service using either one for these classes.

## Extending the IntentService class

Because most started services don't need to handle multiple requests simultaneously (which can actually be a dangerous multi-threading scenario), it's probably best if you implement your service using the [IntentService](#) class.

The [IntentService](#) does the following:

- Creates a default worker thread that executes all intents delivered to [onStartCommand\(\)](#) separate from your application's main thread.
- Creates a work queue that passes one intent at a time to your [onHandleIntent\(\)](#) implementation, so you never have to worry about multi-threading.
- Stops the service after all start requests have been handled, so you never have to call [stopSelf\(\)](#).
- Provides default implementation of [onBind\(\)](#) that returns null.
- Provides a default implementation of [onStartCommand\(\)](#) that sends the intent to the work queue and then to your [onHandleIntent\(\)](#) implementation.

All this adds up to the fact that all you need to do is implement [onHandleIntent\(\)](#) to do the work provided by the client. (Though, you also need to provide a small constructor for the service.)

Here's an example implementation of [IntentService](#):

```
public class HelloIntentService extends IntentService {  
  
    /**  
     * A constructor is required, and must call the super IntentService\(String\)  
     * constructor with a name for the worker thread.  
     */  
    public HelloIntentService() {  
        super("HelloIntentService");  
    }  
  
    /**  
     * The IntentService calls this method from the default worker thread with  
     * the intent that started the service. When this method returns, IntentServic  
     * stops the service, as appropriate.  
     */  
    @Override  
    protected void onHandleIntent(Intent intent) {  
        // Normally we would do some work here, like download a file.  
        // For our sample, we just sleep for 5 seconds.  
    }  
}
```

```

        long endTime = System.currentTimeMillis() + 5*1000;
        while (System.currentTimeMillis() < endTime) {
            synchronized (this) {
                try {
                    wait(endTime - System.currentTimeMillis());
                } catch (Exception e) {
                }
            }
        }
    }
}

```

That's all you need: a constructor and an implementation of [onHandleIntent\(\)](#).

If you decide to also override other callback methods, such as [onCreate\(\)](#), [onStartCommand\(\)](#), or [onDestroy\(\)](#), be sure to call the super implementation, so that the [IntentService](#) can properly handle the life of the worker thread.

For example, [onStartCommand\(\)](#) must return the default implementation (which is how the intent gets delivered to [onHandleIntent\(\)](#)):

```

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();
    return super.onStartCommand(intent, flags, startId);
}

```

Besides [onHandleIntent\(\)](#), the only method from which you don't need to call the super class is [onBind\(\)](#) (but you only need to implement that if your service allows binding).

In the next section, you'll see how the same kind of service is implemented when extending the base [Service](#) class, which is a lot more code, but which might be appropriate if you need to handle simultaneous start requests.

## Extending the Service class

As you saw in the previous section, using [IntentService](#) makes your implementation of a started service very simple. If, however, you require your service to perform multi-threading (instead of processing start requests through a work queue), then you can extend the [Service](#) class to handle each intent.

For comparison, the following example code is an implementation of the [Service](#) class that performs the exact same work as the example above using [IntentService](#). That is, for each start request, it uses a worker thread to perform the job and processes only one request at a time.

```

public class HelloService extends Service {
    private Looper mServiceLooper;
    private ServiceHandler mServiceHandler;

    // Handler that receives messages from the thread
    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
    }
    @Override

```

```
public void handleMessage(Message msg) {
    // Normally we would do some work here, like download a file.
    // For our sample, we just sleep for 5 seconds.
    long endTime = System.currentTimeMillis() + 5*1000;
    while (System.currentTimeMillis() < endTime) {
        synchronized (this) {
            try {
                wait(endTime - System.currentTimeMillis());
            } catch (Exception e) {
            }
        }
    }
    // Stop the service using the startId, so that we don't stop
    // the service in the middle of handling another job
    stopSelf(msg.arg1);
}

@Override
public void onCreate() {
    // Start up the thread running the service. Note that we create a
    // separate thread because the service normally runs in the process's
    // main thread, which we don't want to block. We also make it
    // background priority so CPU-intensive work will not disrupt our UI.
    HandlerThread thread = new HandlerThread("ServiceStartArguments",
        Process.THREAD_PRIORITY_BACKGROUND);
    thread.start();

    // Get the HandlerThread's Looper and use it for our Handler
    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

    // For each start request, send a message to start a job and deliver the
    // start ID so we know which request we're stopping when we finish the job
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    mServiceHandler.sendMessage(msg);

    // If we get killed, after returning from here, restart
    return START_STICKY;
}

@Override
public IBinder onBind(Intent intent) {
    // We don't provide binding, so return null
    return null;
}

@Override
```

```
public void onDestroy() {
    Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
}
}
```

As you can see, it's a lot more work than using [IntentService](#).

However, because you handle each call to [onStartCommand\(\)](#) yourself, you can perform multiple requests simultaneously. That's not what this example does, but if that's what you want, then you can create a new thread for each request and run them right away (instead of waiting for the previous request to finish).

Notice that the [onStartCommand\(\)](#) method must return an integer. The integer is a value that describes how the system should continue the service in the event that the system kills it (as discussed above, the default implementation for [IntentService](#) handles this for you, though you are able to modify it). The return value from [onStartCommand\(\)](#) must be one of the following constants:

#### **START NOT STICKY**

If the system kills the service after [onStartCommand\(\)](#) returns, *do not* recreate the service, unless there are pending intents to deliver. This is the safest option to avoid running your service when not necessary and when your application can simply restart any unfinished jobs.

#### **START STICKY**

If the system kills the service after [onStartCommand\(\)](#) returns, recreate the service and call [onStartCommand\(\)](#), but *do not* redeliver the last intent. Instead, the system calls [onStartCommand\(\)](#) with a null intent, unless there were pending intents to start the service, in which case, those intents are delivered. This is suitable for media players (or similar services) that are not executing commands, but running indefinitely and waiting for a job.

#### **START REDELIVER INTENT**

If the system kills the service after [onStartCommand\(\)](#) returns, recreate the service and call [onStartCommand\(\)](#) with the last intent that was delivered to the service. Any pending intents are delivered in turn. This is suitable for services that are actively performing a job that should be immediately resumed, such as downloading a file.

For more details about these return values, see the linked reference documentation for each constant.

## Starting a Service

You can start a service from an activity or other application component by passing an [Intent](#) (specifying the service to start) to [startService\(\)](#). The Android system calls the service's [onStartCommand\(\)](#) method and passes it the [Intent](#). (You should never call [onStartCommand\(\)](#) directly.)

For example, an activity can start the example service in the previous section (`HelloService`) using an explicit intent with [startService\(\)](#):

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

The [startService\(\)](#) method returns immediately and the Android system calls the service's [onStartCommand\(\)](#) method. If the service is not already running, the system first calls [onCreate\(\)](#), then calls [onStartCommand\(\)](#).

If the service does not also provide binding, the intent delivered with [startService\(\)](#) is the only mode of communication between the application component and the service. However, if you want the service to send a

result back, then the client that starts the service can create a [PendingIntent](#) for a broadcast (with [getBroadcast\(\)](#)) and deliver it to the service in the [Intent](#) that starts the service. The service can then use the broadcast to deliver a result.

Multiple requests to start the service result in multiple corresponding calls to the service's [onStartCommand\(\)](#). However, only one request to stop the service (with [stopSelf\(\)](#) or [stopService\(\)](#)) is required to stop it.

## Stopping a service

A started service must manage its own lifecycle. That is, the system does not stop or destroy the service unless it must recover system memory and the service continues to run after [onStartCommand\(\)](#) returns. So, the service must stop itself by calling [stopSelf\(\)](#) or another component can stop it by calling [stopService\(\)](#).

Once requested to stop with [stopSelf\(\)](#) or [stopService\(\)](#), the system destroys the service as soon as possible.

However, if your service handles multiple requests to [onStartCommand\(\)](#) concurrently, then you shouldn't stop the service when you're done processing a start request, because you might have since received a new start request (stopping at the end of the first request would terminate the second one). To avoid this problem, you can use [stopSelf\(int\)](#) to ensure that your request to stop the service is always based on the most recent start request. That is, when you call [stopSelf\(int\)](#), you pass the ID of the start request (the `startId` delivered to [onStartCommand\(\)](#)) to which your stop request corresponds. Then if the service received a new start request before you were able to call [stopSelf\(int\)](#), then the ID will not match and the service will not stop.

**Caution:** It's important that your application stops its services when it's done working, to avoid wasting system resources and consuming battery power. If necessary, other components can stop the service by calling [stopService\(\)](#). Even if you enable binding for the service, you must always stop the service yourself if it ever received a call to [onStartCommand\(\)](#).

For more information about the lifecycle of a service, see the section below about [Managing the Lifecycle of a Service](#).

## Creating a Bound Service

A bound service is one that allows application components to bind to it by calling [bindService\(\)](#) in order to create a long-standing connection (and generally does not allow components to *start* it by calling [startService\(\)](#)).

You should create a bound service when you want to interact with the service from activities and other components in your application or to expose some of your application's functionality to other applications, through interprocess communication (IPC).

To create a bound service, you must implement the [onBind\(\)](#) callback method to return an [IBinder](#) that defines the interface for communication with the service. Other application components can then call [bindService\(\)](#) to retrieve the interface and begin calling methods on the service. The service lives only to serve the application component that is bound to it, so when there are no components bound to the service, the system destroys it (you do *not* need to stop a bound service in the way you must when the service is started through [onStartCommand\(\)](#)).

To create a bound service, the first thing you must do is define the interface that specifies how a client can communicate with the service. This interface between the service and a client must be an implementation of [IBinder](#) and is what your service must return from the [onBind\(\)](#) callback method. Once the client receives the [IBinder](#), it can begin interacting with the service through that interface.

Multiple clients can bind to the service at once. When a client is done interacting with the service, it calls [unbindService\(\)](#) to unbind. Once there are no clients bound to the service, the system destroys the service.

There are multiple ways to implement a bound service and the implementation is more complicated than a started service, so the bound service discussion appears in a separate document about [Bound Services](#).

## Sending Notifications to the User

Once running, a service can notify the user of events using [Toast Notifications](#) or [Status Bar Notifications](#).

A toast notification is a message that appears on the surface of the current window for a moment then disappears, while a status bar notification provides an icon in the status bar with a message, which the user can select in order to take an action (such as start an activity).

Usually, a status bar notification is the best technique when some background work has completed (such as a file completed downloading) and the user can now act on it. When the user selects the notification from the expanded view, the notification can start an activity (such as to view the downloaded file).

See the [Toast Notifications](#) or [Status Bar Notifications](#) developer guides for more information.

## Running a Service in the Foreground

A foreground service is a service that's considered to be something the user is actively aware of and thus not a candidate for the system to kill when low on memory. A foreground service must provide a notification for the status bar, which is placed under the "Ongoing" heading, which means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground.

For example, a music player that plays music from a service should be set to run in the foreground, because the user is explicitly aware of its operation. The notification in the status bar might indicate the current song and allow the user to launch an activity to interact with the music player.

To request that your service run in the foreground, call [startForeground\(\)](#). This method takes two parameters: an integer that uniquely identifies the notification and the [Notification](#) for the status bar. For example:

```
Notification notification = new Notification(R.drawable.icon, getText(R.string.
    System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIn
notification.setLatestEventInfo(this, getText(R.string.notification_title),
    getText(R.string.notification_message), pendingIntent);
startForeground(ONGOING_NOTIFICATION_ID, notification);
```

**Caution:** The integer ID you give to [startForeground\(\)](#) must not be 0.

To remove the service from the foreground, call [stopForeground\(\)](#). This method takes a boolean, indicating whether to remove the status bar notification as well. This method does *not* stop the service. However, if you stop the service while it's still running in the foreground, then the notification is also removed.

For more information about notifications, see [Creating Status Bar Notifications](#).

## Managing the Lifecycle of a Service

The lifecycle of a service is much simpler than that of an activity. However, it's even more important that you pay close attention to how your service is created and destroyed, because a service can run in the background without the user being aware.

The service lifecycle—from when it's created to when it's destroyed—can follow two different paths:

- A started service

The service is created when another component calls [startService\(\)](#). The service then runs indefinitely and must stop itself by calling [stopSelf\(\)](#). Another component can also stop the service by calling [stopService\(\)](#). When the service is stopped, the system destroys it..

- A bound service

The service is created when another component (a client) calls [bindService\(\)](#). The client then communicates with the service through an [IBinder](#) interface. The client can close the connection by calling [unbindService\(\)](#). Multiple clients can bind to the same service and when all of them unbind, the system destroys the service. (The service does *not* need to stop itself.)

These two paths are not entirely separate. That is, you can bind to a service that was already started with [startService\(\)](#). For example, a background music service could be started by calling [startService\(\)](#) with an [Intent](#) that identifies the music to play. Later, possibly when the user wants to exercise some control over the player or get information about the current song, an activity can bind to the service by calling [bindService\(\)](#). In cases like this, [stopService\(\)](#) or [stopSelf\(\)](#) does not actually stop the service until all clients unbind.

## Implementing the lifecycle callbacks

Like an activity, a service has lifecycle callback methods that you can implement to monitor changes in the service's state and perform work at the appropriate times. The following skeleton service demonstrates each of the lifecycle methods:

```
public class ExampleService extends Service {
    int mStartMode;          // indicates how to behave if the service is killed
    IBinder mBinder;         // interface for clients that bind
    boolean mAllowRebind;    // indicates whether onRebind should be used

    @Override
    public void onCreate() {
        // The service is being created
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // The service is starting, due to a call to startService()
        return mStartMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService()
    }
}
```

```

        return mBinder;
    }

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

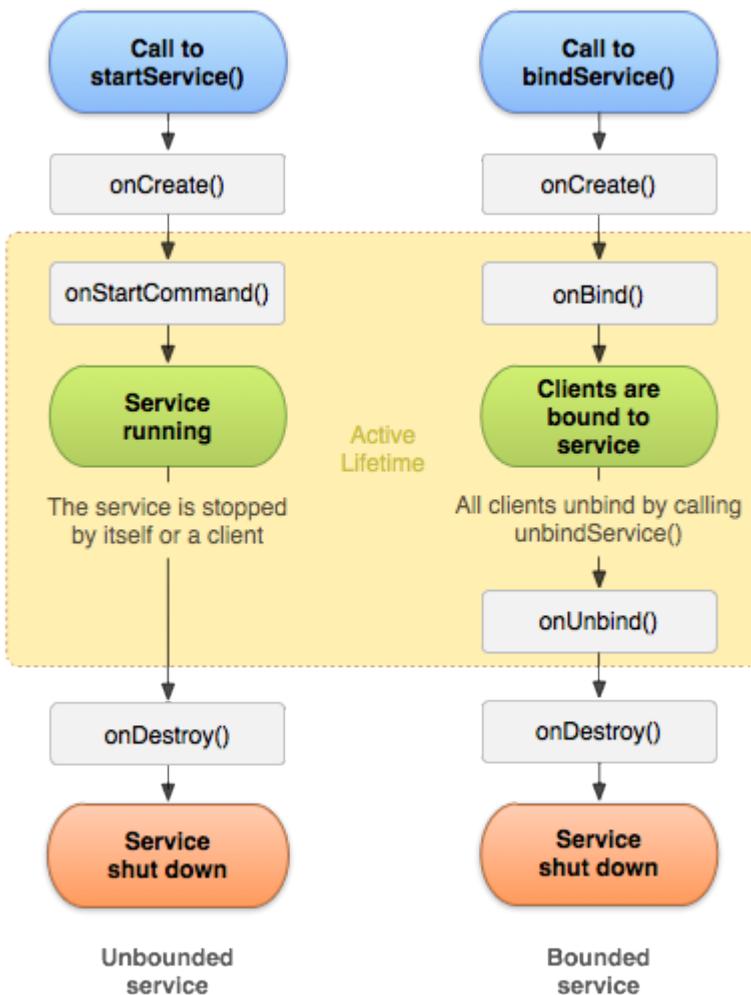
@Override
public void onRebind(Intent intent) {
    // A client is binding to the service with bindService(),
    // after onUnbind() has already been called
}

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

}

```

**Note:** Unlike the activity lifecycle callback methods, you are *not* required to call the superclass implementation of these callback methods.



**Figure 2.** The service lifecycle. The diagram on the left shows the lifecycle when the service is created with `startService()` and the diagram on the right shows the lifecycle when the service is created with `bindService()`.

By implementing these methods, you can monitor two nested loops of the service's lifecycle:

- The **entire lifetime** of a service happens between the time `onCreate()` is called and the time `onDestroy()` returns. Like an activity, a service does its initial setup in `onCreate()` and releases all remaining resources in `onDestroy()`. For example, a music playback service could create the thread where the music will be played in `onCreate()`, then stop the thread in `onDestroy()`.

The `onCreate()` and `onDestroy()` methods are called for all services, whether they're created by `startService()` or `bindService()`.

- The **active lifetime** of a service begins with a call to either `onStartCommand()` or `onBind()`. Each method is handed the `Intent` that was passed to either `startService()` or `bindService()`, respectively.

If the service is started, the active lifetime ends the same time that the entire lifetime ends (the service is still active even after `onStartCommand()` returns). If the service is bound, the active lifetime ends when `onUnbind()` returns.

**Note:** Although a started service is stopped by a call to either `stopSelf()` or `stopService()`, there is not a respective callback for the service (there's no `onStop()` callback). So, unless the service is bound to a client, the system destroys it when the service is stopped—`onDestroy()` is the only callback received.

Figure 2 illustrates the typical callback methods for a service. Although the figure separates services that are created by `startService()` from those created by `bindService()`, keep in mind that any service, no matter how it's started, can potentially allow clients to bind to it. So, a service that was initially started with `onStartCommand()` (by a client calling `startService()`) can still receive a call to `onBind()` (when a client calls `bindService()`).

For more information about creating a service that provides binding, see the [Bound Services](#) document, which includes more information about the `onRebind()` callback method in the section about [Managing the Lifecycle of a Bound Service](#).

# Content Providers

## Topics

1. [Content Provider Basics](#)
2. [Creating a Content Provider](#)
3. [Calendar Provider](#)
4. [Contacts Provider](#)

## Related Samples

1. [Contact Manager](#) application
2. ["Cursor \(People\)"](#)
3. ["Cursor \(Phones\)"](#)
4. [Sample Sync Adapter](#)

Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.

When you want to access data in a content provider, you use the [ContentResolver](#) object in your application's [Context](#) to communicate with the provider as a client. The [ContentResolver](#) object communicates with the provider object, an instance of a class that implements [ContentProvider](#). The provider object receives data requests from clients, performs the requested action, and returns the results.

You don't need to develop your own provider if you don't intend to share your data with other applications. However, you do need your own provider to provide custom search suggestions in your own application. You also need your own provider if you want to copy and paste complex data or files from your application to other applications.

Android itself includes content providers that manage data such as audio, video, images, and personal contact information. You can see some of them listed in the reference documentation for the [android.provider](#) package. With some restrictions, these providers are accessible to any Android application.

The following topics describe content providers in more detail:

### [Content Provider Basics](#)

How to access data in a content provider when the data is organized in tables.

### [Creating a Content Provider](#)

How to create your own content provider.

### [Calendar Provider](#)

How to access the Calendar Provider that is part of the Android platform.

### [Contacts Provider](#)

How to access the Contacts Provider that is part of the Android platform.

# Notifications

## In this document

1. [Notification Display Elements](#)
  1. [Normal view](#)
  2. [Big view](#)
2. [Creating a Notification](#)
  1. [Required notification contents](#)
  2. [Optional notification contents and settings](#)
  3. [Notification actions](#)
  4. [Creating a simple notification](#)
  5. [Applying a big view style to a notification](#)
  6. [Handling compatibility](#)
3. [Managing Notifications](#)
  1. [Updating notifications](#)
  2. [Removing notifications](#)
4. [Preserving Navigation when Starting an Activity](#)
  1. [Setting up a regular activity PendingIntent](#)
  2. [Setting up a special activity PendingIntent](#)
5. [Displaying Progress in a Notification](#)
  1. [Displaying a fixed-duration progress indicator](#)
  2. [Displaying a continuing activity indicator](#)
6. [Custom Notification Layouts](#)

## Key classes

1. [NotificationManager](#)
2. [NotificationCompat](#)

## Videos

1. [Notifications in 4.1](#)

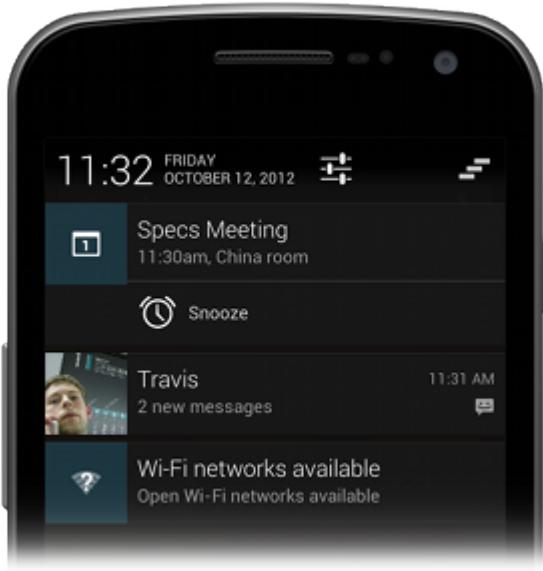
## See also

1. [Android Design: 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.



**Figure 1.** Notifications in the notification area.



**Figure 2.** Notifications in the notification drawer.

## Notification Design

Notifications, as an important part of the Android UI, have their own design guidelines. To learn how to design notifications and their interactions, read the [Android Design Guide Notifications](#) topic.

**Note:** Except where noted, this guide refers to the [NotificationCompat.Builder](#) class in the version 4 [Support Library](#). The class [Notification.Builder](#) was added in Android 3.0.

# Notification Display Elements

Notifications in the notification drawer can appear in one of two visual styles, depending on the version and the state of the drawer:

### Normal view

The standard view of the notifications in the notification drawer.

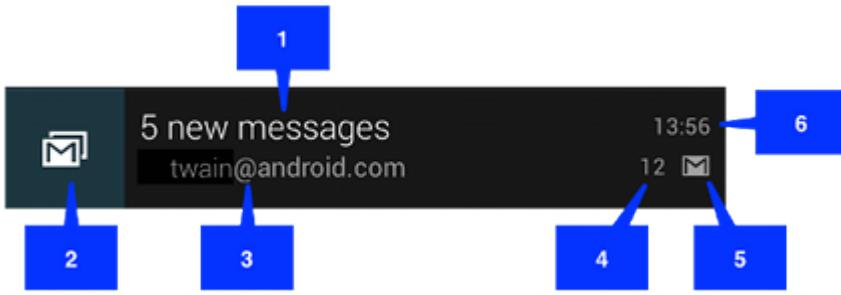
### Big view

A large view that's visible when the notification is expanded. Big view is part of the expanded notification feature available as of Android 4.1.

These styles are described in the following sections.

### Normal view

A notification in normal view appears in an area that's up to 64 dp tall. Even if you create a notification with a big view style, it will appear in normal view until it's expanded. This is an example of a normal view:



**Figure 3.** Notification in normal view.

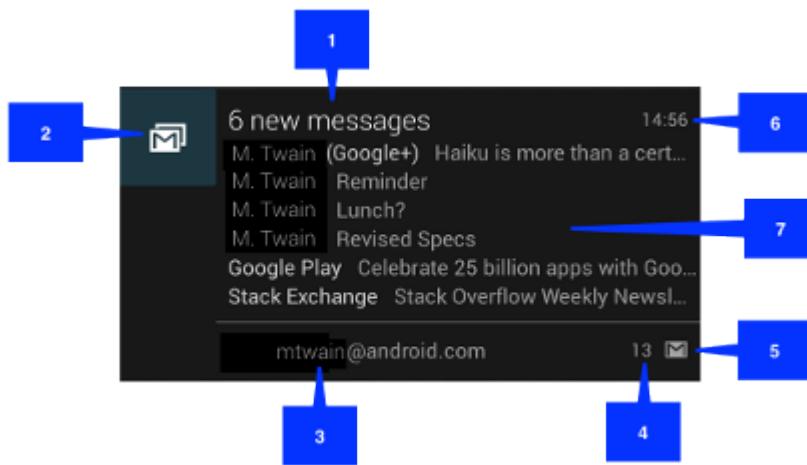
The callouts in the illustration refer to the following:

1. Content title
2. Large icon
3. Content text
4. Content info
5. Small icon
6. Time that the notification was issued. You can set an explicit value with [`setWhen\(\)`](#); if you don't it defaults to the time that the system received the notification.

## Big view

A notification's big view appears only when the notification is expanded, which happens when the notification is at the top of the notification drawer, or when the user expands the notification with a gesture. Expanded notifications are available starting with Android 4.1.

The following screenshot shows an inbox-style notification:



**Figure 4.** Big view notification.

Notice that the big view shares most of its visual elements with the normal view. The only difference is callout number 7, the details area. Each big view style sets this area in a different way. The available styles are:

### Big picture style

The details area contains a bitmap up to 256 dp tall in its detail section.

### Big text style

Displays a large text block in the details section.

## Inbox style

Displays lines of text in the details section.

All of the big view styles also have the following content options that aren't available in normal view:

## Big content title

Allows you to override the normal view's content title with a title that appears only in the expanded view.

## Summary text

Allows you to add a line of text below the details area.

Applying a big view style to a notification is described in the section [Applying a big view style to a notification](#).

# Creating a Notification

You specify the UI information and actions for a notification in a [NotificationCompat.Builder](#) object. To create the notification itself, you 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\(\)](#).

## Required notification contents

A [Notification](#) object *must* contain the following:

- A small icon, set by [setSmallIcon\(\)](#)
- A title, set by [setContentTitle\(\)](#)
- Detail text, set by [setContentText\(\)](#)

## Optional notification contents and settings

All other notification settings and contents are optional. To learn more about them, see the reference documentation for [NotificationCompat.Builder](#).

## Notification actions

Although they're optional, you should add at least one action to your 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.

A notification can provide multiple actions. You should always define the action that's triggered when the user clicks the notification; usually this action opens an [Activity](#) in your application. You can also add buttons to the notification that perform additional actions such as snoozing an alarm or responding immediately to a text message; this feature is available as of Android 4.1. If you use additional action buttons, you must also make their functionality available in an [Activity](#) in your app; see the section [Handling compatibility](#) for more details.

Inside a [Notification](#), the action itself 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\(\)](#).

Starting an [Activity](#) when the user clicks the notification is the most common action scenario. You can also start an [Activity](#) when the user dismisses an [Activity](#). In Android 4.1 and later, you can start an [Activity](#) from an action button. To learn more, read the reference guide for [NotificationCompat.Builder](#).

## Creating a simple notification

The following snippet illustrates a simple notification that specifies an activity to open when the user clicks the notification. Notice that the code creates a [TaskStackBuilder](#) object and uses it to create the [PendingIntent](#) for the action. This pattern is explained in more detail in the section [Preserving Navigation when Starting an Activity](#):

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("My notification")
        .setContentText("Hello World!");
// Creates an explicit intent for an Activity in your app
Intent resultIntent = new Intent(this, ResultActivity.class);

// The stack builder object 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.
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
// Adds the back stack for the Intent (but not the Intent itself)
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);
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// mId allows you to update the notification later on.
mNotificationManager.notify(mId, mBuilder.build());
```

That's it. Your user has now been notified.

## Applying a big view style to a notification

To have a notification appear in a big view when it's expanded, first create a [NotificationCompat.Builder](#) object with the normal view options you want. Next, call [Builder.setStyle\(\)](#) with a big view style object as its argument.

Remember that expanded notifications are not available on platforms prior to Android 4.1. To learn how to handle notifications for Android 4.1 and for earlier platforms, read the section [Handling compatibility](#).

For example, the following code snippet demonstrates how to alter the notification created in the previous snippet to use the Inbox big view style:

```

NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this)
    .setSmallIcon(R.drawable.notification_icon)
    .setContentTitle("Event tracker")
    .setContentText("Events received")
NotificationCompat.InboxStyle inboxStyle =
    new NotificationCompat.InboxStyle();
String[] events = new String[6];
// Sets a title for the Inbox style big view
inboxStyle.setBigContentTitle("Event tracker details:");
...
// Moves events into the big view
for (int i=0; i < events.length; i++) {

    inboxStyle.addLine(events[i]);
}
// Moves the big view style object into the notification object.
mBuilder.setStyle(inboxStyle);
...
// Issue the notification here.

```

## Handling compatibility

Not all notification features are available for a particular version, even though the methods to set them are in the support library class [NotificationCompat.Builder](#). For example, action buttons, which depend on expanded notifications, only appear on Android 4.1 and higher, because expanded notifications themselves are only available on Android 4.1 and higher.

To ensure the best compatibility, create notifications with [NotificationCompat](#) and its subclasses, particularly [NotificationCompat.Builder](#). In addition, follow this process when you implement a notification:

1. Provide all of the notification's functionality to all users, regardless of the version they're using. To do this, verify that all of the functionality is available from an [Activity](#) in your app. You may want to add a new [Activity](#) to do this.

For example, if you want to use [addAction\(\)](#) to provide a control that stops and starts media playback, first implement this control in an [Activity](#) in your app.

2. Ensure that all users can get to the functionality in the [Activity](#), by having it start when users click the notification. To do this, create a [PendingIntent](#) for the [Activity](#). Call [setContentIntent\(\)](#) to add the [PendingIntent](#) to the notification.
3. Now add the expanded notification features you want to use to the notification. Remember that any functionality you add also has to be available in the [Activity](#) that starts when users click the notification.

## Managing Notifications

When you need to issue a notification multiple times for the same type of event, you should avoid making a completely new notification. Instead, you should consider updating a previous notification, either by changing some of its values or by adding to it, or both.

For example, Gmail notifies the user that new emails have arrived by increasing its count of unread messages and by adding a summary of each email to the notification. This is called "stacking" the notification; it's described in more detail in the [Notifications](#) Design guide.

**Note:** This Gmail feature requires the "inbox" big view style, which is part of the expanded notification feature available starting in Android 4.1.

The following section describes how to update notifications and also how to remove them.

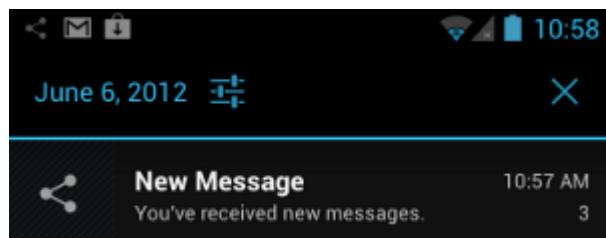
## Updating notifications

To set up a notification so it can be updated, issue it with a notification ID by calling [NotificationManager.notify\(ID, notification\)](#). To update this notification once you've issued it, update or create a [NotificationCompat.Builder](#) object, build a [Notification](#) object from it, and issue the [Notification](#) with the same ID you used previously. If the previous notification is still visible, the system updates it from the contents of the [Notification](#) object. If the previous notification has been dismissed, a new notification is created instead.

The following snippet demonstrates a notification that is updated to reflect the number of events that have occurred. It stacks the notification, showing a summary:

```
mNotificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// Sets an ID for the notification, so it can be updated
int notifyID = 1;
mNotifyBuilder = new NotificationCompat.Builder(this)
        .setContentTitle("New Message")
        .setContentText("You've received new messages.")
        .setSmallIcon(R.drawable.ic_notify_status)
numMessages = 0;
// Start of a loop that processes data and then notifies the user
...
    mNotifyBuilder.setContentText(currentText)
        .setNumber(++numMessages);
// Because the ID remains unchanged, the existing notification is
// updated.
mNotificationManager.notify(
        notifyID,
        mNotifyBuilder.build());
...
```

This produces a notification that looks like this:



**Figure 5.** Updated notification displayed in the notification drawer.

## Removing notifications

Notifications remain visible until one of the following happens:

- The user dismisses the notification either individually or by using "Clear All" (if the notification can be cleared).
- The user clicks the notification, and you called `setAutoCancel()` when you created the notification.
- You call `cancel()` for a specific notification ID. This method also deletes ongoing notifications.
- You call `cancelAll()`, which removes all of the notifications you previously issued.

## Preserving Navigation when Starting an Activity

When you start an [Activity](#) from a notification, you must preserve the user's expected navigation experience. Clicking **Back** should take the user back through the application's normal work flow to the Home screen, and clicking **Recents** should show the [Activity](#) as a separate task. To preserve the navigation experience, you should start the [Activity](#) in a fresh task. How you set up the [PendingIntent](#) to give you a fresh task depends on the nature of the [Activity](#) you're starting. There are two general situations:

### Regular activity

You're starting an [Activity](#) that's part of the application's normal workflow. In this situation, set up the [PendingIntent](#) to start a fresh task, and provide the [PendingIntent](#) with a back stack that reproduces the application's normal **Back** behavior.

Notifications from the Gmail app demonstrate this. When you click a notification for a single email message, you see the message itself. Touching **Back** takes you backwards through Gmail to the Home screen, just as if you had entered Gmail from the Home screen rather than entering it from a notification.

This happens regardless of the application you were in when you touched the notification. For example, if you're in Gmail composing a message, and you click a notification for a single email, you go immediately to that email. Touching **Back** takes you to the inbox and then the Home screen, rather than taking you to the message you were composing.

### Special activity

The user only sees this [Activity](#) if it's started from a notification. In a sense, the [Activity](#) extends the notification by providing information that would be hard to display in the notification itself. For this situation, set up the [PendingIntent](#) to start in a fresh task. There's no need to create a back stack, though, because the started [Activity](#) isn't part of the application's activity flow. Clicking **Back** will still take the user to the Home screen.

## Setting up a regular activity PendingIntent

To set up a [PendingIntent](#) that starts a direct entry [Activity](#), follow these steps:

1. Define your application's [Activity](#) hierarchy in the manifest.
  1. Add support for Android 4.0.3 and earlier. To do this, specify the parent of the [Activity](#) you're starting by adding a `<meta-data>` element as the child of the `<activity>`.

For this element, set `android:name="android.support.PARENT_ACTIVITY"`. Set `android:value="<parent_activity_name>"` where `<parent_activity_name>` is the value of `android:name` for the parent `<activity>` element. See the following XML for an example.

2. Also add support for Android 4.1 and later. To do this, add the `an-droid:parentActivityName` attribute to the `<activity>` element of the `Activity` you're starting.

The final XML should look like this:

```
<activity
    android:name=".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=".ResultActivity"
    android:parentActivityName=".MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

2. Create a back stack based on the `Intent` that starts the `Activity`:

1. Create the `Intent` to start the `Activity`.
2. Create a stack builder by calling `TaskStackBuilder.create()`.
3. Add the back stack to the stack builder by calling `addParentStack()`. For each `Activity` in the hierarchy you've defined in the manifest, the back stack contains an `Intent` object that starts the `Activity`. This method also adds flags that start the stack in a fresh task.

**Note:** Although the argument to `addParentStack()` is a reference to the started `Activity`, the method call doesn't add the `Intent` that starts the `Activity`. Instead, that's taken care of in the next step.

4. Add the `Intent` that starts the `Activity` from the notification, by calling `addNextIntent()`. Pass the `Intent` you created in the first step as the argument to `addNextIntent()`.
5. If you need to, add arguments to `Intent` objects on the stack by calling `TaskStackBuilder.editIntentAt()`. This is sometimes necessary to ensure that the target `Activity` displays meaningful data when the user navigates to it using *Back*.
6. Get a `PendingIntent` for this back stack by calling `getPendingIntent()`. You can then use this `PendingIntent` as the argument to `setContentIntent()`.

The following code snippet demonstrates the process:

```
...
Intent resultIntent = new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
// Adds the back stack
stackBuilder.addParentStack(ResultActivity.class);
// Adds the Intent to the top of the stack
stackBuilder.addNextIntent(resultIntent);
// Gets a PendingIntent containing the entire back stack
PendingIntent resultPendingIntent =
```

```
stackBuilder.getPendingIntent(0, PendingIntent.FLAG_UPDATE_CURRENT);  
...  
NotificationCompat.Builder builder = new NotificationCompat.Builder(this);  
builder.setContentIntent(resultPendingIntent);  
NotificationManager mNotificationManager =  
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);  
mNotificationManager.notify(id, builder.build());
```

## Setting up a special activity PendingIntent

The following section describes how to set up a special activity [PendingIntent](#).

A special [Activity](#) doesn't need a back stack, so you don't have to define its [Activity](#) hierarchy in the manifest, and you don't have to call [addParentStack\(\)](#) to build a back stack. Instead, use the manifest to set up the [Activity](#) task options, and create the [PendingIntent](#) by calling [getActivity\(\)](#):

1. In your manifest, add the following attributes to the [`<activity>`](#) element for the [Activity](#)  
[`android:name="activityclass"`](#)  
The activity's fully-qualified class name.

### [`android:taskAffinity=""`](#)

Combined with the [FLAG\\_ACTIVITY\\_NEW\\_TASK](#) flag that you set in code, this ensures that this [Activity](#) doesn't go into the application's default task. Any existing tasks that have the application's default affinity are not affected.

### [`android:excludeFromRecents="true"`](#)

Excludes the new task from *Recents*, so that the user can't accidentally navigate back to it.

This snippet shows the element:

```
<activity  
    android:name=".ResultActivity"  
...  
    android:launchMode="singleTask"  
    android:taskAffinity=""  
    android:excludeFromRecents="true">  
</activity>  
...
```

2. Build and issue the notification:

1. Create an [Intent](#) that starts the [Activity](#).
2. Set the [Activity](#) to start in a new, empty task by calling [setFlags\(\)](#) with the flags [FLAG\\_ACTIVITY\\_NEW\\_TASK](#) and [FLAG\\_ACTIVITY\\_CLEAR\\_TASK](#).
3. Set any other options you need for the [Intent](#).
4. Create a [PendingIntent](#) from the [Intent](#) by calling [getActivity\(\)](#). You can then use this [PendingIntent](#) as the argument to [setContentIntent\(\)](#).

The following code snippet demonstrates the process:

```
// Instantiate a Builder object.  
NotificationCompat.Builder builder = new NotificationCompat.Builder(this)  
// Creates an Intent for the Activity  
Intent notifyIntent =  
    new Intent(new ComponentName(this, ResultActivity.class));
```

```

// Sets the Activity to start in a new, empty task
notifyIntent.setFlags(FLAG_ACTIVITY_NEW_TASK | FLAG_ACTIVITY_CLEAR_TASK);
// Creates the PendingIntent
PendingIntent notifyIntent =
    PendingIntent.getActivity(
        this,
        0,
        notifyIntent
        PendingIntent.FLAG_UPDATE_CURRENT
    );

// Puts the PendingIntent into the notification builder
builder.setContentIntent(notifyIntent);
// Notifications are issued by sending them to the
// NotificationManager system service.
NotificationManager mNotificationManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
// Builds an anonymous Notification object from the builder, and
// passes it to the NotificationManager
mNotificationManager.notify(id, builder.build());

```

## Displaying Progress in a Notification

Notifications can include an animated progress indicator that shows users the status of an ongoing operation. If you can estimate how long the operation takes and how much of it is complete at any time, use the "determinate" form of the indicator (a progress bar). If you can't estimate the length of the operation, use the "ineterminate" form of the indicator (an activity indicator).

Progress indicators are displayed with the platform's implementation of the [ProgressBar](#) class.

To use a progress indicator on platforms starting with Android 4.0, call [setProgress\(\)](#). For previous versions, you must create your own custom notification layout that includes a [ProgressBar](#) view.

The following sections describe how to display progress in a notification using [setProgress\(\)](#).

### Displaying a fixed-duration progress indicator

To display a determinate progress bar, add the bar to your notification by calling [setProgress\(\) setProgress\(max, progress, false\)](#) and then issue the notification. As your operation proceeds, increment progress, and update the notification. At the end of the operation, progress should equal max. A common way to call [setProgress\(\)](#) is to set max to 100 and then increment progress as a "percent complete" value for the operation.

You can either leave the progress bar showing when the operation is done, or remove it. In either case, remember to update the notification text to show that the operation is complete. To remove the progress bar, call [setProgress\(\) setProgress\(0, 0, false\)](#). For example:

```

...
mNotifyManager =
    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mBuilder = new NotificationCompat.Builder(this);
mBuilder.setContentTitle("Picture Download")
    .setContentText("Download in progress")

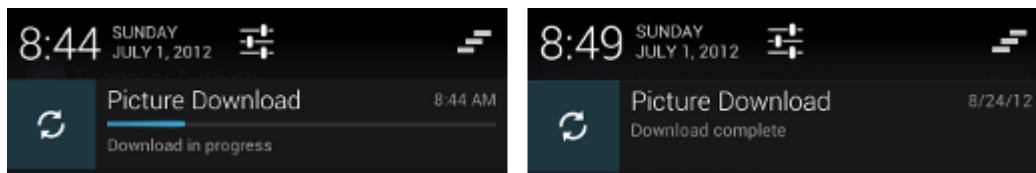
```

```

.setSmallIcon(R.drawable.ic_notification);
// Start a lengthy operation in a background thread
new Thread(
    new Runnable() {
        @Override
        public void run() {
            int incr;
            // Do the "lengthy" operation 20 times
            for (incr = 0; incr <= 100; incr+=5) {
                // Sets the progress indicator to a max value, the
                // current completion percentage, and "determinate"
                // state
                mBuilder.setProgress(100, incr, false);
                // Displays the progress bar for the first time.
                mNotifyManager.notify(0, mBuilder.build());
                // Sleeps the thread, simulating an operation
                // that takes time
                try {
                    // Sleep for 5 seconds
                    Thread.sleep(5*1000);
                } catch (InterruptedException e) {
                    Log.d(TAG, "sleep failure");
                }
            }
            // When the loop is finished, updates the notification
            mBuilder.setContentText("Download complete")
            // Removes the progress bar
            .setProgress(0,0,false);
            mNotifyManager.notify(ID, mBuilder.build());
        }
    }
)
// Starts the thread by calling the run() method in its Runnable
).start();

```

The resulting notifications are shown in figure 6. On the left side is a snapshot of the notification during the operation; on the right side is a snapshot of it after the operation has finished.



**Figure 6.** The progress bar during and after the operation.

## Displaying a continuing activity indicator

To display an indeterminate activity indicator, add it to your notification with [`setProgress\(0, 0, true\)`](#) (the first two arguments are ignored), and issue the notification. The result is an indicator that has the same style as a progress bar, except that its animation is ongoing.

Issue the notification at the beginning of the operation. The animation will run until you modify your notification. When the operation is done, call [`setProgress\(\)`](#) [`setProgress\(0, 0, false\)`](#) and then update the notification to remove the activity indicator. Always do this; otherwise, the animation will run even when

the operation is complete. Also remember to change the notification text to indicate that the operation is complete.

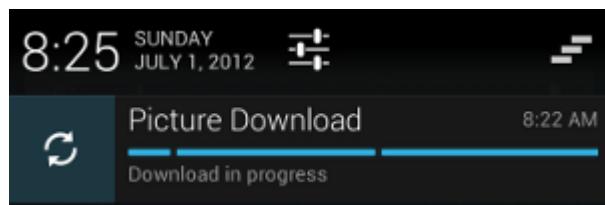
To see how activity indicators work, refer to the preceding snippet. Locate the following lines:

```
// Sets the progress indicator to a max value, the current completion  
// percentage, and "determinate" state  
mBuilder.setProgress(100, incr, false);  
// Issues the notification  
mNotifyManager.notify(0, mBuilder.build());
```

Replace the lines you've found with the following lines:

```
// Sets an activity indicator for an operation of indeterminate length  
mBuilder.setProgress(0, 0, true);  
// Issues the notification  
mNotifyManager.notify(0, mBuilder.build());
```

The resulting indicator is shown in figure 7:



**Figure 7.** An ongoing activity indicator.

## Custom Notification Layouts

The notifications framework allows you to define a custom notification layout, which defines the notification's appearance in a [RemoteViews](#) object. Custom layout notifications are similar to normal notifications, but they're based on a [RemoteViews](#) defined in a XML layout file.

The height available for a custom notification layout depends on the notification view. Normal view layouts are limited to 64 dp, and expanded view layouts are limited to 256 dp.

To define a custom notification layout, start by instantiating a [RemoteViews](#) object that inflates an XML layout file. Then, instead of calling methods such as [setContentTitle\(\)](#), call [setContent\(\)](#). To set content details in the custom notification, use the methods in [RemoteViews](#) to set the values of the view's children:

1. Create an XML layout for the notification in a separate file. You can use any file name you wish, but you must use the extension .xml
2. In your app, use [RemoteViews](#) methods to define your notification's icons and text. Put this [RemoteViews](#) object into your [NotificationCompat.Builder](#) by calling [setContent\(\)](#). Avoid setting a background [Drawable](#) on your [RemoteViews](#) object, because your text color may become unreadable.

The [RemoteViews](#) class also includes methods that you can use to easily add a [Chronometer](#) or [ProgressBar](#) to your notification's layout. For more information about creating custom layouts for your notification, refer to the [RemoteViews](#) reference documentation.

**Caution:** When you use a custom notification layout, take special care to ensure that your custom layout works with different device orientations and resolutions. While this advice applies to all View layouts, it's especially important for notifications because the space in the notification drawer is very restricted. Don't make your custom layout too complex, and be sure to test it in various configurations.

## Using style resources for custom notification text

Always use style resources for the text of a custom notification. The background color of the notification can vary across different devices and versions, and using style resources helps you account for this. Starting in Android 2.3, the system defined a style for the standard notification layout text. If you use the same style in applications that target Android 2.3 or higher, you'll ensure that your text is visible against the display background.

# Intents and Intent Filters

## In this document

1. [Intent Objects](#)
2. [Intent Resolution](#)
3. [Intent filters](#)
4. [Common cases](#)
5. [Using intent matching](#)
6. [Note Pad Example](#)

## Key classes

1. [Intent](#)
2. [IntentFilter](#)
3. [BroadcastReceiver](#)
4. [PackageManager](#)

Three of the core components of an application — activities, services, and broadcast receivers — are activated through messages, called *intents*. Intent messaging is a facility for late run-time binding between components in the same or different applications. The intent itself, an [Intent](#) object, is a passive data structure holding an abstract description of an operation to be performed — or, often in the case of broadcasts, a description of something that has happened and is being announced. There are separate mechanisms for delivering intents to each type of component:

- An Intent object is passed to [Context.startActivity\(\)](#) or [Activity.startActivityForResult\(\)](#) to launch an activity or get an existing activity to do something new. (It can also be passed to [Activity.setResult\(\)](#) to return information to the activity that called `startActivityForResult()`.)
- An Intent object is passed to [Context.startService\(\)](#) to initiate a service or deliver new instructions to an ongoing service. Similarly, an intent can be passed to [Context.bindService\(\)](#) to establish a connection between the calling component and a target service. It can optionally initiate the service if it's not already running.
- Intent objects passed to any of the broadcast methods (such as [Context.sendBroadcast\(\)](#), [Context.sendOrderedBroadcast\(\)](#), or [Context.sendStickyBroadcast\(\)](#)) are delivered to all interested broadcast receivers. Many kinds of broadcasts originate in system code.

In each case, the Android system finds the appropriate activity, service, or set of broadcast receivers to respond to the intent, instantiating them if necessary. There is no overlap within these messaging systems: Broadcast intents are delivered only to broadcast receivers, never to activities or services. An intent passed to `startActivity()` is delivered only to an activity, never to a service or broadcast receiver, and so on.

This document begins with a description of Intent objects. It then describes the rules Android uses to map intents to components — how it resolves which component should receive an intent message. For intents that don't explicitly name a target component, this process involves testing the Intent object against *intent filters* associated with potential targets.

# Intent Objects

An [Intent](#) object is a bundle of information. It contains information of interest to the component that receives the intent (such as the action to be taken and the data to act on) plus information of interest to the Android system (such as the category of component that should handle the intent and instructions on how to launch a target activity). Principally, it can contain the following:

## Component name

The name of the component that should handle the intent. This field is a [ComponentName](#) object — a combination of the fully qualified class name of the target component (for example "com.example.project.app.FreneticActivity") and the package name set in the manifest file of the application where the component resides (for example, "com.example.project"). The package part of the component name and the package name set in the manifest do not necessarily have to match.

The component name is optional. If it is set, the Intent object is delivered to an instance of the designated class. If it is not set, Android uses other information in the Intent object to locate a suitable target — see [Intent Resolution](#), later in this document.

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

## Action

A string naming the action to be performed — or, in the case of broadcast intents, the action that took place and is being reported. The Intent class defines a number of action constants, including these:

Constant	Target component	Action
ACTION_CALL	activity	Initiate a phone call.
ACTION_EDIT	activity	Display data for the user to edit.
ACTION_MAIN	activity	Start up as the initial activity of a task, with no data input and no returned output.
ACTION_SYNC	activity	Synchronize data on a server with data on the mobile device.
ACTION_BATTERY_LOW	broadcast receiver	A warning that the battery is low.
ACTION_HEADSET_PLUG	broadcast receiver	A headset has been plugged into the device, or unplugged from it.
ACTION_SCREEN_ON	broadcast receiver	The screen has been turned on.
ACTION_TIMEZONE_CHANGED	broadcast receiver	The setting for the time zone has changed.

See the [Intent](#) class description for a list of pre-defined constants for generic actions. Other actions are defined elsewhere in the Android API. You can also define your own action strings for activating the components in your application. Those you invent should include the application package as a prefix — for example: "com.example.project.SHOW\_COLOR".

The action largely determines how the rest of the intent is structured — particularly the [data](#) and [extras](#) fields — much as a method name determines a set of arguments and a return value. For this reason, it's a good idea to use action names that are as specific as possible, and to couple them tightly to the other fields

of the intent. In other words, instead of defining an action in isolation, define an entire protocol for the Intent objects your components can handle.

The action in an Intent object is set by the [setAction\(\)](#) method and read by [getAction\(\)](#).

## Data

The URI of the data to be acted on and the MIME type of that data. Different actions are paired with different kinds of data specifications. For example, if the action field is ACTION\_EDIT, the data field would contain the URI of the document to be displayed for editing. If the action is ACTION\_CALL, the data field would be a tel: URI with the number to call. Similarly, if the action is ACTION\_VIEW and the data field is an http: URI, the receiving activity would be called upon to download and display whatever data the URI refers to.

When matching an intent to a component that is capable of handling the data, it's often important to know the type of data (its MIME type) in addition to its URI. For example, a component able to display image data should not be called upon to play an audio file.

In many cases, the data type can be inferred from the URI — particularly content: URIs, which indicate that the data is located on the device and controlled by a content provider (see the [separate discussion on content providers](#)). But the type can also be explicitly set in the Intent object. The [setData\(\)](#) method specifies data only as a URI, [setType\(\)](#) specifies it only as a MIME type, and [setDataAndType\(\)](#) specifies it as both a URI and a MIME type. The URI is read by [getData\(\)](#) and the type by [getType\(\)](#).

## Category

A string containing additional information about the kind of component that should handle the intent. Any number of category descriptions can be placed in an Intent object. As it does for actions, the Intent class defines several category constants, including these:

Constant	Meaning
CATEGORY_BROWSABLE	The target activity can be safely invoked by the browser to display data referenced by a link — for example, an image or an e-mail message.
CATEGORY_GADGET	The activity can be embedded inside of another activity that hosts gadgets.
CATEGORY_HOME	The activity displays the home screen, the first screen the user sees when the device is turned on or when the <i>Home</i> button is pressed.
CATEGORY_LAUNCHER	The activity can be the initial activity of a task and is listed in the top-level application launcher.
CATEGORY_PREFERENCE	The target activity is a preference panel.

See the [Intent](#) class description for the full list of categories.

The [addCategory\(\)](#) method places a category in an Intent object, [removeCategory\(\)](#) deletes a category previously added, and [getCategories\(\)](#) gets the set of all categories currently in the object.

## Extras

Key-value pairs for additional information that should be delivered to the component handling the intent. Just as some actions are paired with particular kinds of data URIs, some are paired with particular extras. For example, an ACTION\_TIMEZONE\_CHANGED intent has a "time-zone" extra that identifies the new time zone, and ACTION\_HEADSET\_PLUG has a "state" extra indicating whether the headset is now plugged in or unplugged, as well as a "name" extra for the type of headset. If you were to invent a SHOW\_COLOR action, the color value would be set in an extra key-value pair.

The Intent object has a series of `put...()` methods for inserting various types of extra data and a similar set of `get...()` methods for reading the data. These methods parallel those for [Bundle](#) objects. In fact, the extras can be installed and read as a Bundle using the [putExtras\(\)](#) and [getExtras\(\)](#) methods.

## Flags

Flags of various sorts. Many instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's launched (for example, whether it belongs in the list of recent activities). All these flags are defined in the Intent class.

The Android system and the applications that come with the platform employ Intent objects both to send out system-originated broadcasts and to activate system-defined components. To see how to structure an intent to activate a system component, consult the [list of intents](#) in the reference.

## Intent Resolution

Intents can be divided into two groups:

- *Explicit intents* designate the target component by its name (the [component name field](#), mentioned earlier, has a value set). Since component names would generally not be known to developers of other applications, explicit intents are typically used for application-internal messages — such as an activity starting a subordinate service or launching a sister activity.
- *Implicit intents* do not name a target (the field for the component name is blank). Implicit intents are often used to activate components in other applications.

Android delivers an explicit intent to an instance of the designated target class. Nothing in the Intent object other than the component name matters for determining which component should get the intent.

A different strategy is needed for implicit intents. In the absence of a designated target, the Android system must find the best component (or components) to handle the intent — a single activity or service to perform the requested action or the set of broadcast receivers to respond to the broadcast announcement. It does so by comparing the contents of the Intent object to *intent filters*, structures associated with components that can potentially receive intents. Filters advertise the capabilities of a component and delimit the intents it can handle. They open the component to the possibility of receiving implicit intents of the advertised type. If a component does not have any intent filters, it can receive only explicit intents. A component with filters can receive both explicit and implicit intents.

Only three aspects of an Intent object are consulted when the object is tested against an intent filter:

action  
data (both URI and data type)  
category

The extras and flags play no part in resolving which component receives an intent.

## Intent filters

To inform the system which implicit intents they can handle, activities, services, and broadcast receivers can have one or more intent filters. Each filter describes a capability of the component, a set of intents that the component is willing to receive. It, in effect, filters in intents of a desired type, while filtering out unwanted intents — but only unwanted implicit intents (those that don't name a target class). An explicit intent is always delivered to its target, no matter what it contains; the filter is not consulted. But an implicit intent is delivered to a component only if it can pass through one of the component's filters.

A component has separate filters for each job it can do, each face it can present to the user. For example, the NoteEditor activity of the sample Note Pad application has two filters — one for starting up with a specific note that the user can view or edit, and another for starting with a new, blank note that the user can fill in and save. (All of Note Pad's filters are described in the [Note Pad Example](#) section, later.)

## Filters and security

An intent filter cannot be relied on for security. While it opens a component to receiving only certain kinds of implicit intents, it does nothing to prevent explicit intents from targeting the component. Even though a filter restricts the intents a component will be asked to handle to certain actions and data sources, someone could always put together an explicit intent with a different action and data source, and name the component as the target.

An intent filter is an instance of the [IntentFilter](#) class. However, since the Android system must know about the capabilities of a component before it can launch that component, intent filters are generally not set up in Java code, but in the application's manifest file (AndroidManifest.xml) as [`<intent-filter>`](#) elements. (The one exception would be filters for broadcast receivers that are registered dynamically by calling [Context.registerReceiver\(\)](#); they are directly created as IntentFilter objects.)

A filter has fields that parallel the action, data, and category fields of an Intent object. An implicit intent is tested against the filter in all three areas. To be delivered to the component that owns the filter, it must pass all three tests. If it fails even one of them, the Android system won't deliver it to the component — at least not on the basis of that filter. However, since a component can have multiple intent filters, an intent that does not pass through one of a component's filters might make it through on another.

Each of the three tests is described in detail below:

### Action test

An [`<intent-filter>`](#) element in the manifest file lists actions as [`<action>`](#) subelements. For example:

```
<intent-filter . . . >
    <action android:name="com.example.project.SHOW_CURRENT" />
    <action android:name="com.example.project.SHOW_RECENT" />
    <action android:name="com.example.project.SHOW_PENDING" />
    . . .
</intent-filter>
```

As the example shows, while an Intent object names just a single action, a filter may list more than one. The list cannot be empty; a filter must contain at least one [`<action>`](#) element, or it will block all intents.

To pass this test, the action specified in the Intent object must match one of the actions listed in the filter. If the object or the filter does not specify an action, the results are as follows:

- If the filter fails to list any actions, there is nothing for an intent to match, so all intents fail the test. No intents can get through the filter.
- On the other hand, an Intent object that doesn't specify an action automatically passes the test — as long as the filter contains at least one action.

### Category test

An [`<intent-filter>`](#) element also lists categories as subelements. For example:

```
<intent-filter . . . >
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    . . .
</intent-filter>
```

Note that the constants described earlier for actions and categories are not used in the manifest file. The full string values are used instead. For instance, the "android.intent.category.BROWSABLE" string in the example above corresponds to the `CATEGORY_BROWSABLE` constant mentioned earlier in this document. Similarly, the string "android.intent.action.EDIT" corresponds to the `ACTION_EDIT` constant.

For an intent to pass the category test, every category in the Intent object must match a category in the filter. The filter can list additional categories, but it cannot omit any that are in the intent.

In principle, therefore, an Intent object with no categories should always pass this test, regardless of what's in the filter. That's mostly true. However, with one exception, Android treats all implicit intents passed to [startActivity\(\)](#) as if they contained at least one category: "android.intent.category.DEFAULT" (the `CATEGORY_DEFAULT` constant). Therefore, activities that are willing to receive implicit intents must include "android.intent.category.DEFAULT" in their intent filters. (Filters with "android.intent.action.MAIN" and "android.intent.category.LAUNCHER" settings are the exception. They mark activities that begin new tasks and that are represented on the launcher screen. They can include "android.intent.category.DEFAULT" in the list of categories, but don't need to.) See [Using intent matching](#), later, for more on these filters.)

## Data test

Like the action and categories, the data specification for an intent filter is contained in a subelement. And, as in those cases, the subelement can appear multiple times, or not at all. For example:

```
<intent-filter . . . >
    <data android:mimeType="video/mpeg" android:scheme="http" . . . />
    <data android:mimeType="audio/mpeg" android:scheme="http" . . . />
    . . .
</intent-filter>
```

Each [`<data>`](#) element can specify a URI and a data type (MIME media type). There are separate attributes — `scheme`, `host`, `port`, and `path` — for each part of the URI:

`scheme://host:port/path`

For example, in the following URI,

`content://com.example.project:200/folder/subfolder/etc`

the scheme is "content", the host is "com.example.project", the port is "200", and the path is "folder/subfolder/etc". The host and port together constitute the URI *authority*; if a host is not specified, the port is ignored.

Each of these attributes is optional, but they are not independent of each other: For an authority to be meaningful, a scheme must also be specified. For a path to be meaningful, both a scheme and an authority must be specified.

When the URI in an Intent object is compared to a URI specification in a filter, it's compared only to the parts of the URI actually mentioned in the filter. For example, if a filter specifies only a scheme, all URIs with that scheme match the filter. If a filter specifies a scheme and an authority but no path, all URIs with the same scheme and authority match, regardless of their paths. If a filter specifies a scheme, an authority, and a path, only URIs with the same scheme, authority, and path match. However, a path specification in the filter can contain wildcards to require only a partial match of the path.

The `type` attribute of a `<data>` element specifies the MIME type of the data. It's more common in filters than a URI. Both the Intent object and the filter can use a "\*" wildcard for the subtype field — for example, "text/\*" or "audio/\*" — indicating any subtype matches.

The data test compares both the URI and the data type in the Intent object to a URI and data type specified in the filter. The rules are as follows:

- a. An Intent object that contains neither a URI nor a data type passes the test only if the filter likewise does not specify any URIs or data types.
- b. An Intent object that contains a URI but no data type (and a type cannot be inferred from the URI) passes the test only if its URI matches a URI in the filter and the filter likewise does not specify a type. This will be the case only for URIs like `mailto:` and `tel:` that do not refer to actual data.
- c. An Intent object that contains a data type but not a URI passes the test only if the filter lists the same data type and similarly does not specify a URI.
- d. An Intent object that contains both a URI and a data type (or a data type can be inferred from the URI) passes the data type part of the test only if its type matches a type listed in the filter. It passes the URI part of the test either if its URI matches a URI in the filter or if it has a `content:` or `file:` URI and the filter does not specify a URI. In other words, a component is presumed to support `content:` and `file:` data if its filter lists only a data type.

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

## Common cases

The last rule shown above for the data test, rule (d), reflects the expectation that components are able to get local data from a file or content provider. Therefore, their filters can list just a data type and do not need to explicitly name the `content:` and `file:` schemes. This is a typical case. A `<data>` element like the following, for example, tells Android that the component can get image data from a content provider and display it:

```
<data android:mimeType="image/*" />
```

Since most available data is dispensed by content providers, filters that specify a data type but not a URI are perhaps the most common.

Another common configuration is filters with a scheme and a data type. For example, a `<data>` element like the following tells Android that the component can get video data from the network and display it:

```
<data android:scheme="http" android:type="video/*" />
```

Consider, for example, what the browser application does when the user follows a link on a web page. It first tries to display the data (as it could if the link was to an HTML page). If it can't display the data, it puts together an implicit intent with the scheme and data type and tries to start an activity that can do the job. If there are no

takers, it asks the download manager to download the data. That puts it under the control of a content provider, so a potentially larger pool of activities (those with filters that just name a data type) can respond.

Most applications also have a way to start fresh, without a reference to any particular data. Activities that can initiate applications have filters with "android.intent.action.MAIN" specified as the action. If they are to be represented in the application launcher, they also specify the "android.intent.category.LAUNCHER" category:

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

## Using intent matching

Intents are matched against intent filters not only to discover a target component to activate, but also to discover something about the set of components on the device. For example, the Android system populates the application launcher, the top-level screen that shows the applications that are available for the user to launch, by finding all the activities with intent filters that specify the "android.intent.action.MAIN" action and "android.intent.category.LAUNCHER" category (as illustrated in the previous section). It then displays the icons and labels of those activities in the launcher. Similarly, it discovers the home screen by looking for the activity with "android.intent.category.HOME" in its filter.

Your application can use intent matching in a similar way. The [PackageManager](#) has a set of `query...` methods that return all components that can accept a particular intent, and a similar series of `resolve...` methods that determine the best component to respond to an intent. For example, [queryIntentActivities\(\)](#) returns a list of all activities that can perform the intent passed as an argument, and [queryIntentServices\(\)](#) returns a similar list of services. Neither method activates the components; they just list the ones that can respond. There's a similar method, [queryBroadcastReceivers\(\)](#), for broadcast receivers.

## Note Pad Example

The Note Pad sample application enables users to browse through a list of notes, view details about individual items in the list, edit the items, and add a new item to the list. This section looks at the intent filters declared in its manifest file. (If you're working offline in the SDK, you can find all the source files for this sample application, including its manifest file, at `<sdk>/samples/NotePad/index.html`. If you're viewing the documentation online, the source files are in the [Tutorials and Sample Code](#) section [here](#).)

In its manifest file, the Note Pad application declares three activities, each with at least one intent filter. It also declares a content provider that manages the note data. Here is the manifest file in its entirety:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.notepad">
    <application android:icon="@drawable/app_notes"
        android:label="@string/app_name" >

        <provider android:name="NotePadProvider"
            android:authorities="com.google.provider.NotePad" />

        <activity android:name="NotesList" android:label="@string/title_notes_1"
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="android.intent.action.PICK" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.GET_CONTENT" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
</activity>

<activity android:name="NoteEditor"
          android:theme="@android:style/Theme.Light"
          android:label="@string/title_note" >
    <intent-filter android:label="@string/resolve_edit">
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="com.android.notepad.action.EDIT_NOTE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.INSERT" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
    </intent-filter>
</activity>

<activity android:name="TitleEditor"
          android:label="@string/title_edit_title"
          android:theme="@android:style/Theme.Dialog">
    <intent-filter android:label="@string/resolve_title">
        <action android:name="com.android.notepad.action.EDIT_TITLE" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.ALTERNATIVE" />
        <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
</activity>

</application>
</manifest>

```

The first activity, NotesList, is distinguished from the other activities by the fact that it operates on a directory of notes (the note list) rather than on a single note. It would generally serve as the initial user interface into the application. It can do three things as described by its three intent filters:

1. <intent-filter>
 

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

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

This filter declares the main entry point into the Note Pad application. The standard `MAIN` action is an entry point that does not require any other information in the Intent (no data specification, for example), and the `LAUNCHER` category says that this entry point should be listed in the application launcher.

```
2. <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.PICK" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

This filter declares the things that the activity can do on a directory of notes. It can allow the user to view or edit the directory (via the `VIEW` and `EDIT` actions), or to pick a particular note from the directory (via the `PICK` action).

The `mimeType` attribute of the [`<data>`](#) element specifies the kind of data that these actions operate on. It indicates that the activity can get a Cursor over zero or more items (`vnd.android.cursor.dir`) from a content provider that holds Note Pad data (`vnd.google.note`). The Intent object that launches the activity would include a `content:` URI specifying the exact data of this type that the activity should open.

Note also the `DEFAULT` category supplied in this filter. It's there because the [`Context.startActivity\(\)`](#) and [`Activity.startActivityForResult\(\)`](#) methods treat all intents as if they contained the `DEFAULT` category — with just two exceptions:

- Intents that explicitly name the target activity
- Intents consisting of the `MAIN` action and `LAUNCHER` category

Therefore, the `DEFAULT` category is *required* for all filters — except for those with the `MAIN` action and `LAUNCHER` category. (Intent filters are not consulted for explicit intents.)

```
3. <intent-filter>
    <action android:name="android.intent.action.GET_CONTENT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

This filter describes the activity's ability to return a note selected by the user without requiring any specification of the directory the user should choose from. The `GET_CONTENT` action is similar to the `PICK` action. In both cases, the activity returns the URI for a note selected by the user. (In each case, it's returned to the activity that called [`startActivityForResult\(\)`](#) to start the `NoteList` activity.) Here, however, the caller specifies the type of data desired instead of the directory of data the user will be picking from.

The data type, `vnd.android.cursor.item/vnd.google.note`, indicates the type of data the activity can return — a URI for a single note. From the returned URI, the caller can get a Cursor for exactly one item (`vnd.android.cursor.item`) from the content provider that holds Note Pad data (`vnd.google.note`).

In other words, for the `PICK` action in the previous filter, the data type indicates the type of data the activity could display to the user. For the `GET_CONTENT` filter, it indicates the type of data the activity can return to the caller.

Given these capabilities, the following intents will resolve to the `NotesList` activity:

**action: `android.intent.action.MAIN`**

Launches the activity with no data specified.

**action: `android.intent.action.MAIN`**

**category: `android.intent.category.LAUNCHER`**

Launches the activity with no data selected specified. This is the actual intent used by the Launcher to populate its top-level list. All activities with filters that match this action and category are added to the list.

**action: `android.intent.action.VIEW`**

**data: `content://com.google.provider.NotePad/notes`**

Asks the activity to display a list of all the notes under `content://com.google.provider.NotePad/notes`. The user can then browse through the list and get information about the items in it.

**action: `android.intent.action.PICK`**

**data: `content://com.google.provider.NotePad/notes`**

Asks the activity to display a list of the notes under `content://com.google.provider.NotePad/notes`. The user can then pick a note from the list, and the activity will return the URI for that item back to the activity that started the `NoteList` activity.

**action: `android.intent.action.GET_CONTENT`**

**data type: `vnd.android.cursor.item/vnd.google.note`**

Asks the activity to supply a single item of Note Pad data.

The second activity, `NoteEditor`, shows users a single note entry and allows them to edit it. It can do two things as described by its two intent filters:

1. 

```
<intent-filter android:label="@string/resolve_edit">
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="com.android.notepad.action.EDIT_NOTE" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

The first, primary, purpose of this activity is to enable the user to interact with a single note — to either `VIEW` the note or `EDIT` it. (The `EDIT_NOTE` category is a synonym for `EDIT`.) The intent would contain the URI for data matching the MIME type `vnd.android.cursor.item/vnd.google.note` — that is, the URI for a single, specific note. It would typically be a URI that was returned by the `PICK` or `GET_CONTENT` actions of the `NoteList` activity.

As before, this filter lists the `DEFAULT` category so that the activity can be launched by intents that don't explicitly specify the `NoteEditor` class.

2. 

```
<intent-filter>
    <action android:name="android.intent.action.INSERT" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

```
<data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

The secondary purpose of this activity is to enable the user to create a new note, which it will `INSERT` into an existing directory of notes. The intent would contain the URI for data matching the MIME type `vnd.android.cursor.dir/vnd.google.note` — that is, the URI for the directory where the note should be placed.

Given these capabilities, the following intents will resolve to the `NoteEditor` activity:

**action: `android.intent.action.VIEW`**

**data: `content://com.google.provider.NotePad/notes/ID`**

Asks the activity to display the content of the note identified by *ID*. (For details on how `content:` URIs specify individual members of a group, see [Content Providers](#).)

**action: `android.intent.action.EDIT`**

**data: `content://com.google.provider.NotePad/notes/ID`**

Asks the activity to display the content of the note identified by *ID*, and to let the user edit it. If the user saves the changes, the activity updates the data for the note in the content provider.

**action: `android.intent.action.INSERT`**

**data: `content://com.google.provider.NotePad/notes`**

Asks the activity to create a new, empty note in the notes list at `content://com.google.provider.NotePad/notes` and allow the user to edit it. If the user saves the note, its URI is returned to the caller.

The last activity, `TitleEditor`, enables the user to edit the title of a note. This could be implemented by directly invoking the activity (by explicitly setting its component name in the Intent), without using an intent filter. But here we take the opportunity to show how to publish alternative operations on existing data:

```
<intent-filter android:label="@string/resolve_title">
    <action android:name="com.android.notepad.action.EDIT_TITLE" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.ALTERNATIVE" />
    <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

The single intent filter for this activity uses a custom action called `"com.android.notepad.action.EDIT_TITLE"`. It must be invoked on a specific note (data type `vnd.android.cursor.item/vnd.google.note`), like the previous `VIEW` and `EDIT` actions. However, here the activity displays the title contained in the note data, not the content of the note itself.

In addition to supporting the usual `DEFAULT` category, the title editor also supports two other standard categories: `ALTERNATIVE` and `SELECTED_ALTERNATIVE`. These categories identify activities that can be presented to users in a menu of options (much as the `LAUNCHER` category identifies activities that should be presented to user in the application launcher). Note that the filter also supplies an explicit label (via `android:label="@string/resolve_title"`) to better control what users see when presented with this activity as an alternative action to the data they are currently viewing. (For more information on these categories and building options menus, see the [PackageManager.queryIntentActivityOptions\(\)](#) and [Menu.addIntentOptions\(\)](#) methods.)

Given these capabilities, the following intent will resolve to the `TitleEditor` activity:

**action: com.android.notePad.action.EDIT\_TITLE**

**data: content://com.google.provider.NotePad/notes/*ID***

Asks the activity to display the title associated with note *ID*, and allow the user to edit the title.

# Activities

## Quickview

- An activity provides a user interface for a single screen in your application
- Activities can move into the background and then be resumed with their state restored

## In this document

1. [Creating an Activity](#)
  1. [Implementing a user interface](#)
  2. [Declaring the activity in the manifest](#)
2. [Starting an Activity](#)
  1. [Starting an activity for a result](#)
3. [Shutting Down an Activity](#)
4. [Managing the Activity Lifecycle](#)
  1. [Implementing the lifecycle callbacks](#)
  2. [Saving activity state](#)
  3. [Handling configuration changes](#)
  4. [Coordinating activities](#)

## Key classes

1. [Activity](#)

## See also

1. [Tasks and Back Stack](#)

An [Activity](#) is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. The back stack abides to the basic "last in, first out" stack mechanism, so, when the user is done with the current activity and presses the *Back* button, it is popped from the stack (and destroyed) and the previous activity resumes. (The back stack is discussed more in the [Tasks and Back Stack](#) document.)

When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides you the opportunity to perform specific work that's appropriate to that state change. For instance, when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted. These state transitions are all part of the activity lifecycle.

The rest of this document discusses the basics of how to build and use an activity, including a complete discussion of how the activity lifecycle works, so you can properly manage the transition between various activity states.

## Creating an Activity

To create an activity, you must create a subclass of [Activity](#) (or an existing subclass of it). In your subclass, you need to implement callback methods that the system calls when the activity transitions between various states of its lifecycle, such as when the activity is being created, stopped, resumed, or destroyed. The two most important callback methods are:

### [onCreate\(\)](#)

You must implement this method. The system calls this when creating your activity. Within your implementation, you should initialize the essential components of your activity. Most importantly, this is where you must call [setContentView\(\)](#) to define the layout for the activity's user interface.

### [onPause\(\)](#)

The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

There are several other lifecycle callback methods that you should use in order to provide a fluid user experience between activities and handle unexpected interruptions that cause your activity to be stopped and even destroyed. All of the lifecycle callback methods are discussed later, in the section about [Managing the Activity Lifecycle](#).

## Implementing a user interface

The user interface for an activity is provided by a hierarchy of views—objects derived from the [View](#) class. Each view controls a particular rectangular space within the activity's window and can respond to user interaction. For example, a view might be a button that initiates an action when the user touches it.

Android provides a number of ready-made views that you can use to design and organize your layout. "Widgets" are views that provide a visual (and interactive) elements for the screen, such as a button, text field, checkbox, or just an image. "Layouts" are views derived from [ViewGroup](#) that provide a unique layout model for its child views, such as a linear layout, a grid layout, or relative layout. You can also subclass the [View](#) and [ViewGroup](#) classes (or existing subclasses) to create your own widgets and layouts and apply them to your activity layout.

The most common way to define a layout using views is with an XML layout file saved in your application resources. This way, you can maintain the design of your user interface separately from the source code that defines the activity's behavior. You can set the layout as the UI for your activity with [setContentView\(\)](#), passing the resource ID for the layout. However, you can also create new [Views](#) in your activity code and build a view hierarchy by inserting new [Views](#) into a [ViewGroup](#), then use that layout by passing the root [ViewGroup](#) to [setContentView\(\)](#).

For information about creating a user interface, see the [User Interface](#) documentation.

## Declaring the activity in the manifest

You must declare your activity in the manifest file in order for it to be accessible to the system. To declare your activity, open your manifest file and add an [<activity>](#) element as a child of the [<application>](#) element. For example:

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
...
</manifest >
```

There are several other attributes that you can include in this element, to define properties such as the label for the activity, an icon for the activity, or a theme to style the activity's UI. The [android:name](#) attribute is the only required attribute—it specifies the class name of the activity. Once you publish your application, you should not change this name, because if you do, you might break some functionality, such as application shortcuts (read the blog post, [Things That Cannot Change](#)).

See the [<activity>](#) element reference for more information about declaring your activity in the manifest.

## Using intent filters

An [<activity>](#) element can also specify various intent filters—using the [<intent-filter>](#) element—in order to declare how other application components may activate it.

When you create a new application using the Android SDK tools, the stub activity that's created for you automatically includes an intent filter that declares the activity responds to the "main" action and should be placed in the "launcher" category. The intent filter looks like this:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

The [<action>](#) element specifies that this is the "main" entry point to the application. The [<category>](#) element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).

If you intend for your application to be self-contained and not allow other applications to activate its activities, then you don't need any other intent filters. Only one activity should have the "main" action and "launcher" category, as in the previous example. Activities that you don't want to make available to other applications should have no intent filters and you can start them yourself using explicit intents (as discussed in the following section).

However, if you want your activity to respond to implicit intents that are delivered from other applications (and your own), then you must define additional intent filters for your activity. For each type of intent to which you want to respond, you must include an [<intent-filter>](#) that includes an [<action>](#) element and, optionally, a [<category>](#) element and/or a [<data>](#) element. These elements specify the type of intent to which your activity can respond.

For more information about how your activities can respond to intents, see the [Intents and Intent Filters](#) document.

# Starting an Activity

You can start another activity by calling `startActivity()`, passing it an `Intent` that describes the activity you want to start. The intent specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application). An intent can also carry small amounts of data to be used by the activity that is started.

When working within your own application, you'll often need to simply launch a known activity. You can do so by creating an intent that explicitly defines the activity you want to start, using the class name. For example, here's how one activity starts another activity named `SignInActivity`:

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

However, your application might also want to perform some action, such as send an email, text message, or status update, using data from your activity. In this case, your application might not have its own activities to perform such actions, so you can instead leverage the activities provided by other applications on the device, which can perform the actions for you. This is where intents are really valuable—you can create an intent that describes an action you want to perform and the system launches the appropriate activity from another application. If there are multiple activities that can handle the intent, then the user can select which one to use. For example, if you want to allow the user to send an email message, you can create the following intent:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

The `EXTRA_EMAIL` extra added to the intent is a string array of email addresses to which the email should be sent. When an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form. In this situation, the email application's activity starts and when the user is done, your activity resumes.

## Starting an activity for a result

Sometimes, you might want to receive a result from the activity that you start. In that case, start the activity by calling `startActivityForResult()` (instead of `startActivity()`). To then receive the result from the subsequent activity, implement the `onActivityResult()` callback method. When the subsequent activity is done, it returns a result in an `Intent` to your `onActivityResult()` method.

For example, perhaps you want the user to pick one of their contacts, so your activity can do something with the information in that contact. Here's how you can create such an intent and handle the result:

```
private void pickContact() {
    // Create an intent to "pick" a contact, as defined by the content provider
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode == PICK_CONTACT_REQUEST)
        // Perform a query to the contact's content provider for the contact's
        Cursor cursor = getContentResolver().query(data.getData(),
```

```

        new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not empty
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Do something with the selected contact's name...
        }
    }
}

```

This example shows the basic logic you should use in your [onActivityResult\(\)](#) method in order to handle an activity result. The first condition checks whether the request was successful—if it was, then the `resultCode` will be [RESULT\\_OK](#)—and whether the request to which this result is responding is known—in this case, the `requestCode` matches the second parameter sent with [startActivityForResult\(\)](#). From there, the code handles the activity result by querying the data returned in an [Intent](#) (the `data` parameter).

What happens is, a [ContentResolver](#) performs a query against a content provider, which returns a [Cursor](#) that allows the queried data to be read. For more information, see the [Content Providers](#) document.

For more information about using intents, see the [Intents and Intent Filters](#) document.

## Shutting Down an Activity

You can shut down an activity by calling its [finish\(\)](#) method. You can also shut down a separate activity that you previously started by calling [finishActivity\(\)](#).

**Note:** In most cases, you should not explicitly finish an activity using these methods. As discussed in the following section about the activity lifecycle, the Android system manages the life of an activity for you, so you do not need to finish your own activities. Calling these methods could adversely affect the expected user experience and should only be used when you absolutely do not want the user to return to this instance of the activity.

## Managing the Activity Lifecycle

Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application. The lifecycle of an activity is directly affected by its association with other activities, its task and back stack.

An activity can exist in essentially three states:

### **Resumed**

The activity is in the foreground of the screen and has user focus. (This state is also sometimes referred to as "running".)

### **Paused**

Another activity is in the foreground and has focus, but this one is still visible. That is, another activity is visible on top of this one and that activity is partially transparent or doesn't cover the entire screen. A paused activity is completely alive (the [Activity](#) object is retained in memory, it maintains all state and member information, and remains attached to the window manager), but can be killed by the system in extremely low memory situations.

### **Stopped**

The activity is completely obscured by another activity (the activity is now in the "background"). A stopped activity is also still alive (the [Activity](#) object is retained in memory, it maintains all state and

member information, but is *not* attached to the window manager). However, it is no longer visible to the user and it can be killed by the system when memory is needed elsewhere.

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its [finish\(\)](#) method), or simply killing its process. When the activity is opened again (after being finished or killed), it must be created all over.

## Implementing the lifecycle callbacks

When an activity transitions into and out of the different states described above, it is notified through various callback methods. All of the callback methods are hooks that you can override to do appropriate work when the state of your activity changes. The following skeleton activity includes each of the fundamental lifecycle methods:

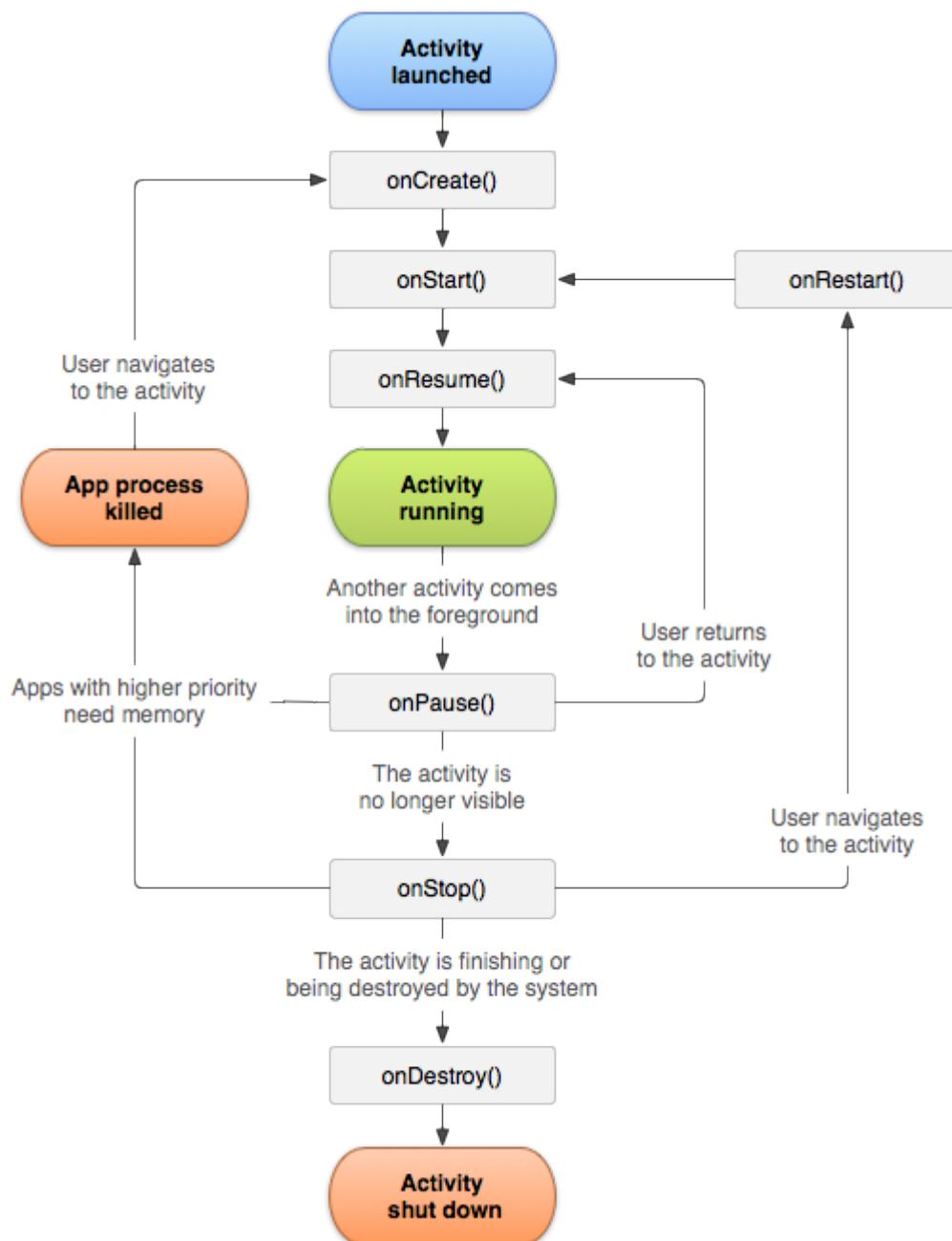
```
public class ExampleActivity extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        // The activity is being created.  
    }  
    @Override  
    protected void onStart\(\) {  
        super.onStart();  
        // The activity is about to become visible.  
    }  
    @Override  
    protected void onResume\(\) {  
        super.onResume();  
        // The activity has become visible (it is now "resumed").  
    }  
    @Override  
    protected void onPause\(\) {  
        super.onPause();  
        // Another activity is taking focus (this activity is about to be "paused").  
    }  
    @Override  
    protected void onStop\(\) {  
        super.onStop();  
        // The activity is no longer visible (it is now "stopped").  
    }  
    @Override  
    protected void onDestroy\(\) {  
        super.onDestroy();  
        // The activity is about to be destroyed.  
    }  
}
```

**Note:** Your implementation of these lifecycle methods must always call the superclass implementation before doing any work, as shown in the examples above.

Taken together, these methods define the entire lifecycle of an activity. By implementing these methods, you can monitor three nested loops in the activity lifecycle:

- The **entire lifetime** of an activity happens between the call to `onCreate()` and the call to `onDestroy()`. Your activity should perform setup of "global" state (such as defining layout) in `onCreate()`, and release all remaining resources in `onDestroy()`. For example, if your activity has a thread running in the background to download data from the network, it might create that thread in `onCreate()` and then stop the thread in `onDestroy()`.
- The **visible lifetime** of an activity happens between the call to `onStart()` and the call to `onStop()`. During this time, the user can see the activity on-screen and interact with it. For example, `onStop()` is called when a new activity starts and this one is no longer visible. Between these two methods, you can maintain resources that are needed to show the activity to the user. For example, you can register a `BroadcastReceiver` in `onStart()` to monitor changes that impact your UI, and unregister it in `onStop()` when the user can no longer see what you are displaying. The system might call `onStart()` and `onStop()` multiple times during the entire lifetime of the activity, as the activity alternates between being visible and hidden to the user.
- The **foreground lifetime** of an activity happens between the call to `onResume()` and the call to `onPause()`. During this time, the activity is in front of all other activities on screen and has user input focus. An activity can frequently transition in and out of the foreground—for example, `onPause()` is called when the device goes to sleep or when a dialog appears. Because this state can transition often, the code in these two methods should be fairly lightweight in order to avoid slow transitions that make the user wait.

Figure 1 illustrates these loops and the paths an activity might take between states. The rectangles represent the callback methods you can implement to perform operations when the activity transitions between states.



**Figure 1.** The activity lifecycle.

The same lifecycle callback methods are listed in table 1, which describes each of the callback methods in more detail and locates each one within the activity's overall lifecycle, including whether the system can kill the activity after the callback method completes.

**Table 1.** A summary of the activity lifecycle's callback methods.

Method	Description	Killable after?	Next
<a href="#">onCreate()</a>	Called when the activity is first created. This is where you should do all of your normal static set up — create views, bind data to lists, and so on. This method is passed a Bundle object containing the activity's previous state, if that state was captured (see <a href="#">Saving Activity State</a> , later).	No	<a href="#">onStart()</a>

Method	Description	Killable after?	Next
	Always followed by <code>onStart()</code> .		
<code>onRestart()</code>	Called after the activity has been stopped, just prior to it being started again.  Always followed by <code>onStart()</code>	No	<code>onStart()</code>
<code>onStart()</code>	Called just before the activity becomes visible to the user.  Followed by <code>onResume()</code> if the activity comes to the foreground, or <code>onStop()</code> if it becomes hidden.	No	<code>onResume()</code> or <code>onStop()</code>
<code>onResume()</code>	Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it.  Always followed by <code>onPause()</code> .	No	<code>onPause()</code>
<code>onPause()</code>	Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.  Followed either by <code>onResume()</code> if the activity returns back to the front, or by <code>onStop()</code> if it becomes invisible to the user.	Yes	<code>onResume()</code> or <code>onStop()</code>
<code>onStop()</code>	Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it.  Followed either by <code>onRestart()</code> if the activity is coming back to interact with the user, or by <code>onDestroy()</code> if this activity is going away.	Yes	<code>onRestart()</code> or <code>onDestroy()</code>
<code>onDestroy()</code>	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing (someone called <code>finish()</code> on	Yes	<i>nothing</i>

Method	Description	Killable after?	Next
	it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the <a href="#">isFinishing()</a> method.		

The column labeled "Killable after?" indicates whether or not the system can kill the process hosting the activity at any time *after the method returns*, without executing another line of the activity's code. Three methods are marked "yes": ([onPause\(\)](#), [onStop\(\)](#), and [onDestroy\(\)](#)). Because [onPause\(\)](#) is the first of the three, once the activity is created, [onPause\(\)](#) is the last method that's guaranteed to be called before the process *can* be killed—if the system must recover memory in an emergency, then [onStop\(\)](#) and [onDestroy\(\)](#) might not be called. Therefore, you should use [onPause\(\)](#) to write crucial persistent data (such as user edits) to storage. However, you should be selective about what information must be retained during [onPause\(\)](#), because any blocking procedures in this method block the transition to the next activity and slow the user experience.

Methods that are marked "No" in the **Killable** column protect the process hosting the activity from being killed from the moment they are called. Thus, an activity is killable from the time [onPause\(\)](#) returns to the time [onResume\(\)](#) is called. It will not again be killable until [onPause\(\)](#) is again called and returns.

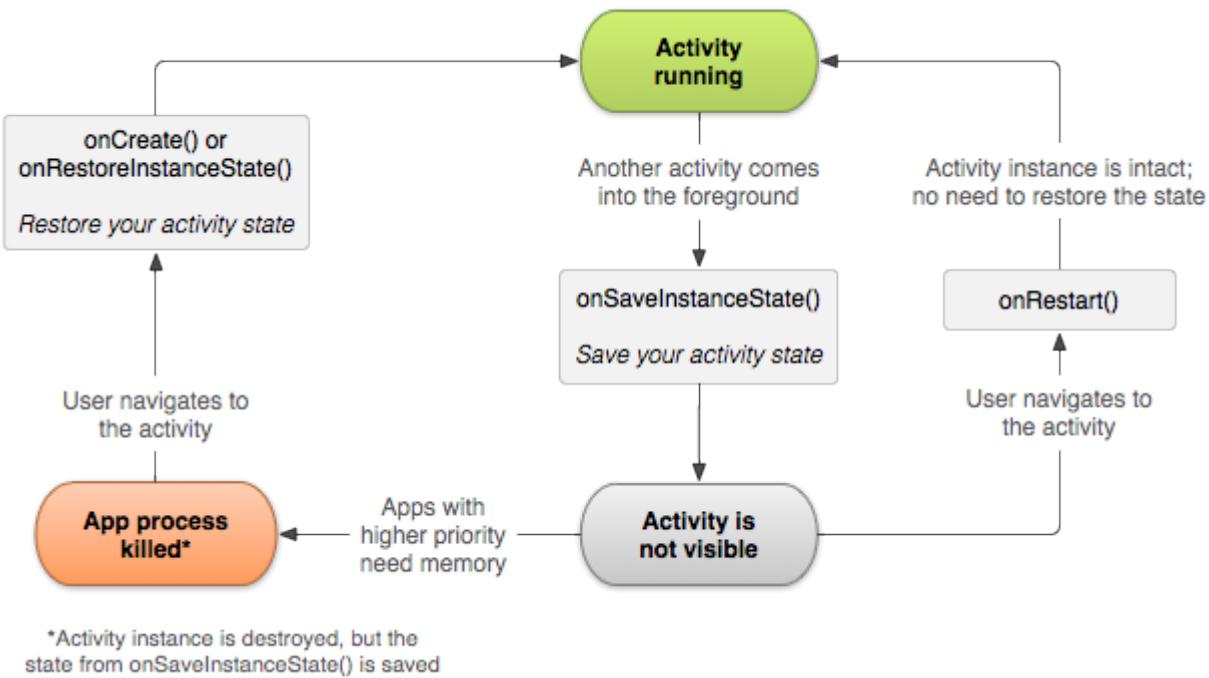
**Note:** An activity that's not technically "killable" by this definition in table 1 might still be killed by the system—but that would happen only in extreme circumstances when there is no other recourse. When an activity might be killed is discussed more in the [Processes and Threading](#) document.

## Saving activity state

The introduction to [Managing the Activity Lifecycle](#) briefly mentions that when an activity is paused or stopped, the state of the activity is retained. This is true because the [Activity](#) object is still held in memory when it is paused or stopped—all information about its members and current state is still alive. Thus, any changes the user made within the activity are retained so that when the activity returns to the foreground (when it "resumes"), those changes are still there.

However, when the system destroys an activity in order to recover memory, the [Activity](#) object is destroyed, so the system cannot simply resume it with its state intact. Instead, the system must recreate the [Activity](#) object if the user navigates back to it. Yet, the user is unaware that the system destroyed the activity and recreated it and, thus, probably expects the activity to be exactly as it was. In this situation, you can ensure that important information about the activity state is preserved by implementing an additional callback method that allows you to save information about the state of your activity: [onSaveInstanceState\(\)](#).

The system calls [onSaveInstanceState\(\)](#) before making the activity vulnerable to destruction. The system passes this method a [Bundle](#) in which you can save state information about the activity as name-value pairs, using methods such as [putString\(\)](#) and [putInt\(\)](#). Then, if the system kills your application process and the user navigates back to your activity, the system recreates the activity and passes the [Bundle](#) to both [onCreate\(\)](#) and [onRestoreInstanceState\(\)](#). Using either of these methods, you can extract your saved state from the [Bundle](#) and restore the activity state. If there is no state information to restore, then the [Bundle](#) passed to you is null (which is the case when the activity is created for the first time).



**Figure 2.** The two ways in which an activity returns to user focus with its state intact: either the activity is destroyed, then recreated and the activity must restore the previously saved state, or the activity is stopped, then resumed and the activity state remains intact.

**Note:** There's no guarantee that `onSaveInstanceState()` will be called before your activity is destroyed, because there are cases in which it won't be necessary to save the state (such as when the user leaves your activity using the *Back* button, because the user is explicitly closing the activity). If the system calls `onSaveInstanceState()`, it does so before `onStop()` and possibly before `onPause()`.

However, even if you do nothing and do not implement `onSaveInstanceState()`, some of the activity state is restored by the `Activity` class's default implementation of `onSaveInstanceState()`. Specifically, the default implementation calls the corresponding `onSaveInstanceState()` method for every `View` in the layout, which allows each view to provide information about itself that should be saved. Almost every widget in the Android framework implements this method as appropriate, such that any visible changes to the UI are automatically saved and restored when your activity is recreated. For example, the `EditText` widget saves any text entered by the user and the `CheckBox` widget saves whether it's checked or not. The only work required by you is to provide a unique ID (with the `android:id` attribute) for each widget you want to save its state. If a widget does not have an ID, then the system cannot save its state.

You can also explicitly stop a view in your layout from saving its state by setting the `android:saveEnabled` attribute to "false" or by calling the `setSaveEnabled()` method. Usually, you should not disable this, but you might if you want to restore the state of the activity UI differently.

Although the default implementation of `onSaveInstanceState()` saves useful information about your activity's UI, you still might need to override it to save additional information. For example, you might need to save member values that changed during the activity's life (which might correlate to values restored in the UI, but the members that hold those UI values are not restored, by default).

Because the default implementation of `onSaveInstanceState()` helps save the state of the UI, if you override the method in order to save additional state information, you should always call the superclass implementation of `onSaveInstanceState()` before doing any work. Likewise, you should also call the superclass implementation of `onRestoreInstanceState()` if you override it, so the default implementation can restore view states.

**Note:** Because `onSaveInstanceState()` is not guaranteed to be called, you should use it only to record the transient state of the activity (the state of the UI)—you should never use it to store persistent data. Instead, you should use `onPause()` to store persistent data (such as data that should be saved to a database) when the user leaves the activity.

A good way to test your application's ability to restore its state is to simply rotate the device so that the screen orientation changes. When the screen orientation changes, the system destroys and recreates the activity in order to apply alternative resources that might be available for the new screen configuration. For this reason alone, it's very important that your activity completely restores its state when it is recreated, because users regularly rotate the screen while using applications.

## Handling configuration changes

Some device configurations can change during runtime (such as screen orientation, keyboard availability, and language). When such a change occurs, Android recreates the running activity (the system calls `onDestroy()`, then immediately calls `onCreate()`). This behavior is designed to help your application adapt to new configurations by automatically reloading your application with alternative resources that you've provided (such as different layouts for different screen orientations and sizes).

If you properly design your activity to handle a restart due to a screen orientation change and restore the activity state as described above, your application will be more resilient to other unexpected events in the activity lifecycle.

The best way to handle such a restart is to save and restore the state of your activity using `onSaveInstanceState()` and `onRestoreInstanceState()` (or `onCreate()`), as discussed in the previous section.

For more information about configuration changes that happen at runtime and how you can handle them, read the guide to [Handling Runtime Changes](#).

## Coordinating activities

When one activity starts another, they both experience lifecycle transitions. The first activity pauses and stops (though, it won't stop if it's still visible in the background), while the other activity is created. In case these activities share data saved to disc or elsewhere, it's important to understand that the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.

The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process and one is starting the other. Here's the order of operations that occur when Activity A starts Activity B:

1. Activity A's `onPause()` method executes.
2. Activity B's `onCreate()`, `onStart()`, and `onResume()` methods execute in sequence. (Activity B now has user focus.)
3. Then, if Activity A is no longer visible on screen, its `onStop()` method executes.

This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another. For example, if you must write to a database when the first activity stops so that the following activity can read it, then you should write to the database during `onPause()` instead of during `onStop()`.

# Services

## Quickview

- A service can run in the background to perform work even while the user is in a different application
- A service can allow other components to bind to it, in order to interact with it and perform interprocess communication
- A service runs in the main thread of the application that hosts it, by default

## In this document

1. [The Basics](#)
  1. [Declaring a service in the manifest](#)
2. [Creating a Started Service](#)
  1. [Extending the IntentService class](#)
  2. [Extending the Service class](#)
  3. [Starting a service](#)
  4. [Stopping a service](#)
3. [Creating a Bound Service](#)
4. [Sending Notifications to the User](#)
5. [Running a Service in the Foreground](#)
6. [Managing the Lifecycle of a Service](#)
  1. [Implementing the lifecycle callbacks](#)

## Key classes

1. [Service](#)
2. [IntentService](#)

## Samples

1. [ServiceStartArguments](#)
2. [LocalService](#)

## See also

1. [Bound Services](#)

A [Service](#) is an application component that can perform long-running operations in the background and does not provide a user interface. Another application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

A service can essentially take two forms:

### Started

A service is "started" when an application component (such as an activity) starts it by calling [startService\(\)](#). Once started, a service can run in the background indefinitely, even if the component that started

it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should stop itself.

## Bound

A service is "bound" when an application component binds to it by calling [bindService\(\)](#). A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Although this documentation generally discusses these two types of services separately, your service can work both ways—it can be started (to run indefinitely) and also allow binding. It's simply a matter of whether you implement a couple callback methods: [onStartCommand\(\)](#) to allow components to start it and [onBind\(\)](#) to allow binding.

Regardless of whether your application is started, bound, or both, any application component can use the service (even from a separate application), in the same way that any component can use an activity—by starting it with an [Intent](#). However, you can declare the service as private, in the manifest file, and block access from other applications. This is discussed more in the section about [Declaring the service in the manifest](#).

**Caution:** A service runs in the main thread of its hosting process—the service does **not** create its own thread and does **not** run in a separate process (unless you specify otherwise). This means that, if your service is going to do any CPU intensive work or blocking operations (such as MP3 playback or networking), you should create a new thread within the service to do that work. By using a separate thread, you will reduce the risk of Application Not Responding (ANR) errors and the application's main thread can remain dedicated to user interaction with your activities.

# The Basics

## Should you use a service or a thread?

A service is simply a component that can run in the background even when the user is not interacting with your application. Thus, you should create a service only if that is what you need.

If you need to perform work outside your main thread, but only while the user is interacting with your application, then you should probably instead create a new thread and not a service. For example, if you want to play some music, but only while your activity is running, you might create a thread in [onCreate\(\)](#), start running it in [onStart\(\)](#), then stop it in [onStop\(\)](#). Also consider using [AsyncTask](#) or [HandlerThread](#), instead of the traditional [Thread](#) class. See the [Processes and Threading](#) document for more information about threads.

Remember that if you do use a service, it still runs in your application's main thread by default, so you should still create a new thread within the service if it performs intensive or blocking operations.

To create a service, you must create a subclass of [Service](#) (or one of its existing subclasses). In your implementation, you need to override some callback methods that handle key aspects of the service lifecycle and provide a mechanism for components to bind to the service, if appropriate. The most important callback methods you should override are:

### [onStartCommand\(\)](#)

The system calls this method when another component, such as an activity, requests that the service be started, by calling [startService\(\)](#). Once this method executes, the service is started and can run in

the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is done, by calling [stopSelf\(\)](#) or [stopService\(\)](#). (If you only want to provide binding, you don't need to implement this method.)

#### [onBind\(\)](#)

The system calls this method when another component wants to bind with the service (such as to perform RPC), by calling [bindService\(\)](#). In your implementation of this method, you must provide an interface that clients use to communicate with the service, by returning an [IBinder](#). You must always implement this method, but if you don't want to allow binding, then you should return null.

#### [onCreate\(\)](#)

The system calls this method when the service is first created, to perform one-time setup procedures (before it calls either [onStartCommand\(\)](#) or [onBind\(\)](#)). If the service is already running, this method is not called.

#### [onDestroy\(\)](#)

The system calls this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc. This is the last call the service receives.

If a component starts the service by calling [startService\(\)](#) (which results in a call to [onStartCommand\(\)](#)), then the service remains running until it stops itself with [stopSelf\(\)](#) or another component stops it by calling [stopService\(\)](#).

If a component calls [bindService\(\)](#) to create the service (and [onStartCommand\(\)](#) is *not* called), then the service runs only as long as the component is bound to it. Once the service is unbound from all clients, the system destroys it.

The Android system will force-stop a service only when memory is low and it must recover system resources for the activity that has user focus. If the service is bound to an activity that has user focus, then it's less likely to be killed, and if the service is declared to [run in the foreground](#) (discussed later), then it will almost never be killed. Otherwise, if the service was started and is long-running, then the system will lower its position in the list of background tasks over time and the service will become highly susceptible to killing—if your service is started, then you must design it to gracefully handle restarts by the system. If the system kills your service, it restarts it as soon as resources become available again (though this also depends on the value you return from [onStartCommand\(\)](#), as discussed later). For more information about when the system might destroy a service, see the [Processes and Threading](#) document.

In the following sections, you'll see how you can create each type of service and how to use it from other application components.

## Declaring a service in the manifest

Like activities (and other components), you must declare all services in your application's manifest file.

To declare your service, add a [`<service>`](#) element as a child of the [`<application>`](#) element. For example:

```
<manifest ... >
  ...
  <application ... >
    <service android:name=".ExampleService" />
  ...

```

```
</application>  
</manifest>
```

There are other attributes you can include in the [`<service>`](#) element to define properties such as permissions required to start the service and the process in which the service should run. The [`android:name`](#) attribute is the only required attribute—it specifies the class name of the service. Once you publish your application, you should not change this name, because if you do, you might break some functionality where explicit intents are used to reference your service (read the blog post, [Things That Cannot Change](#)).

See the [`<service>`](#) element reference for more information about declaring your service in the manifest.

Just like an activity, a service can define intent filters that allow other components to invoke the service using implicit intents. By declaring intent filters, components from any application installed on the user's device can potentially start your service if your service declares an intent filter that matches the intent another application passes to [`startService\(\)`](#).

If you plan on using your service only locally (other applications do not use it), then you don't need to (and should not) supply any intent filters. Without any intent filters, you must start the service using an intent that explicitly names the service class. More information about [starting a service](#) is discussed below.

Additionally, you can ensure that your service is private to your application only if you include the [`an-  
droid:exported`](#) attribute and set it to "false". This is effective even if your service supplies intent filters.

For more information about creating intent filters for your service, see the [Intents and Intent Filters](#) document.

## Creating a Started Service

A started service is one that another component starts by calling [`startService\(\)`](#), resulting in a call to the service's [`onStartCommand\(\)`](#) method.

When a service is started, it has a lifecycle that's independent of the component that started it and the service can run in the background indefinitely, even if the component that started it is destroyed. As such, the service should stop itself when its job is done by calling [`stopSelf\(\)`](#), or another component can stop it by calling [`stopService\(\)`](#).

An application component such as an activity can start the service by calling [`startService\(\)`](#) and passing an [`Intent`](#) that specifies the service and includes any data for the service to use. The service receives this [`Intent`](#) in the [`onStartCommand\(\)`](#) method.

For instance, suppose an activity needs to save some data to an online database. The activity can start a companion service and deliver it the data to save by passing an intent to [`startService\(\)`](#). The service receives the intent in [`onStartCommand\(\)`](#), connects to the Internet and performs the database transaction. When the transaction is done, the service stops itself and it is destroyed.

**Caution:** A service runs in the same process as the application in which it is declared and in the main thread of that application, by default. So, if your service performs intensive or blocking operations while the user interacts with an activity from the same application, the service will slow down activity performance. To avoid impacting application performance, you should start a new thread inside the service.

Traditionally, there are two classes you can extend to create a started service:

## Service

This is the base class for all services. When you extend this class, it's important that you create a new thread in which to do all the service's work, because the service uses your application's main thread, by default, which could slow the performance of any activity your application is running.

## IntentService

This is a subclass of [Service](#) that uses a worker thread to handle all start requests, one at a time. This is the best option if you don't require that your service handle multiple requests simultaneously. All you need to do is implement [onHandleIntent\(\)](#), which receives the intent for each start request so you can do the background work.

The following sections describe how you can implement your service using either one for these classes.

## Extending the IntentService class

Because most started services don't need to handle multiple requests simultaneously (which can actually be a dangerous multi-threading scenario), it's probably best if you implement your service using the [IntentService](#) class.

The [IntentService](#) does the following:

- Creates a default worker thread that executes all intents delivered to [onStartCommand\(\)](#) separate from your application's main thread.
- Creates a work queue that passes one intent at a time to your [onHandleIntent\(\)](#) implementation, so you never have to worry about multi-threading.
- Stops the service after all start requests have been handled, so you never have to call [stopSelf\(\)](#).
- Provides default implementation of [onBind\(\)](#) that returns null.
- Provides a default implementation of [onStartCommand\(\)](#) that sends the intent to the work queue and then to your [onHandleIntent\(\)](#) implementation.

All this adds up to the fact that all you need to do is implement [onHandleIntent\(\)](#) to do the work provided by the client. (Though, you also need to provide a small constructor for the service.)

Here's an example implementation of [IntentService](#):

```
public class HelloIntentService extends IntentService {  
  
    /**  
     * A constructor is required, and must call the super IntentService\(String\)  
     * constructor with a name for the worker thread.  
     */  
    public HelloIntentService() {  
        super("HelloIntentService");  
    }  
  
    /**  
     * The IntentService calls this method from the default worker thread with  
     * the intent that started the service. When this method returns, IntentServic  
     * stops the service, as appropriate.  
     */  
    @Override  
    protected void onHandleIntent(Intent intent) {  
        // Normally we would do some work here, like download a file.  
        // For our sample, we just sleep for 5 seconds.  
    }  
}
```

```

        long endTime = System.currentTimeMillis() + 5*1000;
        while (System.currentTimeMillis() < endTime) {
            synchronized (this) {
                try {
                    wait(endTime - System.currentTimeMillis());
                } catch (Exception e) {
                }
            }
        }
    }
}

```

That's all you need: a constructor and an implementation of [onHandleIntent\(\)](#).

If you decide to also override other callback methods, such as [onCreate\(\)](#), [onStartCommand\(\)](#), or [onDestroy\(\)](#), be sure to call the super implementation, so that the [IntentService](#) can properly handle the life of the worker thread.

For example, [onStartCommand\(\)](#) must return the default implementation (which is how the intent gets delivered to [onHandleIntent\(\)](#)):

```

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();
    return super.onStartCommand(intent, flags, startId);
}

```

Besides [onHandleIntent\(\)](#), the only method from which you don't need to call the super class is [onBind\(\)](#) (but you only need to implement that if your service allows binding).

In the next section, you'll see how the same kind of service is implemented when extending the base [Service](#) class, which is a lot more code, but which might be appropriate if you need to handle simultaneous start requests.

## Extending the Service class

As you saw in the previous section, using [IntentService](#) makes your implementation of a started service very simple. If, however, you require your service to perform multi-threading (instead of processing start requests through a work queue), then you can extend the [Service](#) class to handle each intent.

For comparison, the following example code is an implementation of the [Service](#) class that performs the exact same work as the example above using [IntentService](#). That is, for each start request, it uses a worker thread to perform the job and processes only one request at a time.

```

public class HelloService extends Service {
    private Looper mServiceLooper;
    private ServiceHandler mServiceHandler;

    // Handler that receives messages from the thread
    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
    }
    @Override

```

```
public void handleMessage(Message msg) {
    // Normally we would do some work here, like download a file.
    // For our sample, we just sleep for 5 seconds.
    long endTime = System.currentTimeMillis() + 5*1000;
    while (System.currentTimeMillis() < endTime) {
        synchronized (this) {
            try {
                wait(endTime - System.currentTimeMillis());
            } catch (Exception e) {
            }
        }
    }
    // Stop the service using the startId, so that we don't stop
    // the service in the middle of handling another job
    stopSelf(msg.arg1);
}

@Override
public void onCreate() {
    // Start up the thread running the service. Note that we create a
    // separate thread because the service normally runs in the process's
    // main thread, which we don't want to block. We also make it
    // background priority so CPU-intensive work will not disrupt our UI.
    HandlerThread thread = new HandlerThread("ServiceStartArguments",
        Process.THREAD_PRIORITY_BACKGROUND);
    thread.start();

    // Get the HandlerThread's Looper and use it for our Handler
    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

    // For each start request, send a message to start a job and deliver the
    // start ID so we know which request we're stopping when we finish the job
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    mServiceHandler.sendMessage(msg);

    // If we get killed, after returning from here, restart
    return START_STICKY;
}

@Override
public IBinder onBind(Intent intent) {
    // We don't provide binding, so return null
    return null;
}

@Override
```

```
public void onDestroy() {
    Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
}
}
```

As you can see, it's a lot more work than using [IntentService](#).

However, because you handle each call to [onStartCommand\(\)](#) yourself, you can perform multiple requests simultaneously. That's not what this example does, but if that's what you want, then you can create a new thread for each request and run them right away (instead of waiting for the previous request to finish).

Notice that the [onStartCommand\(\)](#) method must return an integer. The integer is a value that describes how the system should continue the service in the event that the system kills it (as discussed above, the default implementation for [IntentService](#) handles this for you, though you are able to modify it). The return value from [onStartCommand\(\)](#) must be one of the following constants:

#### **START NOT STICKY**

If the system kills the service after [onStartCommand\(\)](#) returns, *do not* recreate the service, unless there are pending intents to deliver. This is the safest option to avoid running your service when not necessary and when your application can simply restart any unfinished jobs.

#### **START STICKY**

If the system kills the service after [onStartCommand\(\)](#) returns, recreate the service and call [onStartCommand\(\)](#), but *do not* redeliver the last intent. Instead, the system calls [onStartCommand\(\)](#) with a null intent, unless there were pending intents to start the service, in which case, those intents are delivered. This is suitable for media players (or similar services) that are not executing commands, but running indefinitely and waiting for a job.

#### **START REDELIVER INTENT**

If the system kills the service after [onStartCommand\(\)](#) returns, recreate the service and call [onStartCommand\(\)](#) with the last intent that was delivered to the service. Any pending intents are delivered in turn. This is suitable for services that are actively performing a job that should be immediately resumed, such as downloading a file.

For more details about these return values, see the linked reference documentation for each constant.

## Starting a Service

You can start a service from an activity or other application component by passing an [Intent](#) (specifying the service to start) to [startService\(\)](#). The Android system calls the service's [onStartCommand\(\)](#) method and passes it the [Intent](#). (You should never call [onStartCommand\(\)](#) directly.)

For example, an activity can start the example service in the previous section (`HelloService`) using an explicit intent with [startService\(\)](#):

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

The [startService\(\)](#) method returns immediately and the Android system calls the service's [onStartCommand\(\)](#) method. If the service is not already running, the system first calls [onCreate\(\)](#), then calls [onStartCommand\(\)](#).

If the service does not also provide binding, the intent delivered with [startService\(\)](#) is the only mode of communication between the application component and the service. However, if you want the service to send a

result back, then the client that starts the service can create a [PendingIntent](#) for a broadcast (with [getBroadcast\(\)](#)) and deliver it to the service in the [Intent](#) that starts the service. The service can then use the broadcast to deliver a result.

Multiple requests to start the service result in multiple corresponding calls to the service's [onStartCommand\(\)](#). However, only one request to stop the service (with [stopSelf\(\)](#) or [stopService\(\)](#)) is required to stop it.

## Stopping a service

A started service must manage its own lifecycle. That is, the system does not stop or destroy the service unless it must recover system memory and the service continues to run after [onStartCommand\(\)](#) returns. So, the service must stop itself by calling [stopSelf\(\)](#) or another component can stop it by calling [stopService\(\)](#).

Once requested to stop with [stopSelf\(\)](#) or [stopService\(\)](#), the system destroys the service as soon as possible.

However, if your service handles multiple requests to [onStartCommand\(\)](#) concurrently, then you shouldn't stop the service when you're done processing a start request, because you might have since received a new start request (stopping at the end of the first request would terminate the second one). To avoid this problem, you can use [stopSelf\(int\)](#) to ensure that your request to stop the service is always based on the most recent start request. That is, when you call [stopSelf\(int\)](#), you pass the ID of the start request (the `startId` delivered to [onStartCommand\(\)](#)) to which your stop request corresponds. Then if the service received a new start request before you were able to call [stopSelf\(int\)](#), then the ID will not match and the service will not stop.

**Caution:** It's important that your application stops its services when it's done working, to avoid wasting system resources and consuming battery power. If necessary, other components can stop the service by calling [stopService\(\)](#). Even if you enable binding for the service, you must always stop the service yourself if it ever received a call to [onStartCommand\(\)](#).

For more information about the lifecycle of a service, see the section below about [Managing the Lifecycle of a Service](#).

## Creating a Bound Service

A bound service is one that allows application components to bind to it by calling [bindService\(\)](#) in order to create a long-standing connection (and generally does not allow components to *start* it by calling [startService\(\)](#)).

You should create a bound service when you want to interact with the service from activities and other components in your application or to expose some of your application's functionality to other applications, through interprocess communication (IPC).

To create a bound service, you must implement the [onBind\(\)](#) callback method to return an [IBinder](#) that defines the interface for communication with the service. Other application components can then call [bindService\(\)](#) to retrieve the interface and begin calling methods on the service. The service lives only to serve the application component that is bound to it, so when there are no components bound to the service, the system destroys it (you do *not* need to stop a bound service in the way you must when the service is started through [onStartCommand\(\)](#)).

To create a bound service, the first thing you must do is define the interface that specifies how a client can communicate with the service. This interface between the service and a client must be an implementation of [IBinder](#) and is what your service must return from the [onBind\(\)](#) callback method. Once the client receives the [IBinder](#), it can begin interacting with the service through that interface.

Multiple clients can bind to the service at once. When a client is done interacting with the service, it calls [unbindService\(\)](#) to unbind. Once there are no clients bound to the service, the system destroys the service.

There are multiple ways to implement a bound service and the implementation is more complicated than a started service, so the bound service discussion appears in a separate document about [Bound Services](#).

## Sending Notifications to the User

Once running, a service can notify the user of events using [Toast Notifications](#) or [Status Bar Notifications](#).

A toast notification is a message that appears on the surface of the current window for a moment then disappears, while a status bar notification provides an icon in the status bar with a message, which the user can select in order to take an action (such as start an activity).

Usually, a status bar notification is the best technique when some background work has completed (such as a file completed downloading) and the user can now act on it. When the user selects the notification from the expanded view, the notification can start an activity (such as to view the downloaded file).

See the [Toast Notifications](#) or [Status Bar Notifications](#) developer guides for more information.

## Running a Service in the Foreground

A foreground service is a service that's considered to be something the user is actively aware of and thus not a candidate for the system to kill when low on memory. A foreground service must provide a notification for the status bar, which is placed under the "Ongoing" heading, which means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground.

For example, a music player that plays music from a service should be set to run in the foreground, because the user is explicitly aware of its operation. The notification in the status bar might indicate the current song and allow the user to launch an activity to interact with the music player.

To request that your service run in the foreground, call [startForeground\(\)](#). This method takes two parameters: an integer that uniquely identifies the notification and the [Notification](#) for the status bar. For example:

```
Notification notification = new Notification(R.drawable.icon, getText(R.string.
    System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIn
notification.setLatestEventInfo(this, getText(R.string.notification_title),
    getText(R.string.notification_message), pendingIntent);
startForeground(ONGOING_NOTIFICATION_ID, notification);
```

**Caution:** The integer ID you give to [startForeground\(\)](#) must not be 0.

To remove the service from the foreground, call [stopForeground\(\)](#). This method takes a boolean, indicating whether to remove the status bar notification as well. This method does *not* stop the service. However, if you stop the service while it's still running in the foreground, then the notification is also removed.

For more information about notifications, see [Creating Status Bar Notifications](#).

## Managing the Lifecycle of a Service

The lifecycle of a service is much simpler than that of an activity. However, it's even more important that you pay close attention to how your service is created and destroyed, because a service can run in the background without the user being aware.

The service lifecycle—from when it's created to when it's destroyed—can follow two different paths:

- A started service

The service is created when another component calls [startService\(\)](#). The service then runs indefinitely and must stop itself by calling [stopSelf\(\)](#). Another component can also stop the service by calling [stopService\(\)](#). When the service is stopped, the system destroys it..

- A bound service

The service is created when another component (a client) calls [bindService\(\)](#). The client then communicates with the service through an [IBinder](#) interface. The client can close the connection by calling [unbindService\(\)](#). Multiple clients can bind to the same service and when all of them unbind, the system destroys the service. (The service does *not* need to stop itself.)

These two paths are not entirely separate. That is, you can bind to a service that was already started with [startService\(\)](#). For example, a background music service could be started by calling [startService\(\)](#) with an [Intent](#) that identifies the music to play. Later, possibly when the user wants to exercise some control over the player or get information about the current song, an activity can bind to the service by calling [bindService\(\)](#). In cases like this, [stopService\(\)](#) or [stopSelf\(\)](#) does not actually stop the service until all clients unbind.

## Implementing the lifecycle callbacks

Like an activity, a service has lifecycle callback methods that you can implement to monitor changes in the service's state and perform work at the appropriate times. The following skeleton service demonstrates each of the lifecycle methods:

```
public class ExampleService extends Service {
    int mStartMode;          // indicates how to behave if the service is killed
    IBinder mBinder;         // interface for clients that bind
    boolean mAllowRebind;    // indicates whether onRebind should be used

    @Override
    public void onCreate() {
        // The service is being created
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // The service is starting, due to a call to startService()
        return mStartMode;
    }
    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService()
    }
}
```

```

        return mBinder;
    }

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

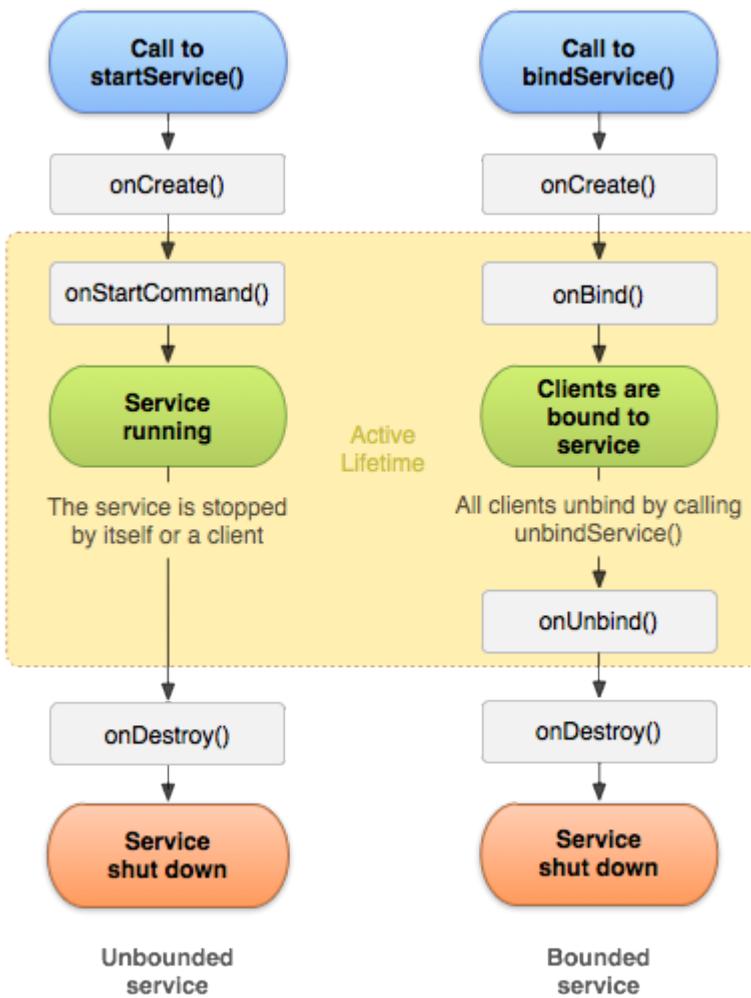
@Override
public void onRebind(Intent intent) {
    // A client is binding to the service with bindService(),
    // after onUnbind() has already been called
}

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

}

```

**Note:** Unlike the activity lifecycle callback methods, you are *not* required to call the superclass implementation of these callback methods.



**Figure 2.** The service lifecycle. The diagram on the left shows the lifecycle when the service is created with `startService()` and the diagram on the right shows the lifecycle when the service is created with `bindService()`.

By implementing these methods, you can monitor two nested loops of the service's lifecycle:

- The **entire lifetime** of a service happens between the time `onCreate()` is called and the time `onDestroy()` returns. Like an activity, a service does its initial setup in `onCreate()` and releases all remaining resources in `onDestroy()`. For example, a music playback service could create the thread where the music will be played in `onCreate()`, then stop the thread in `onDestroy()`.

The `onCreate()` and `onDestroy()` methods are called for all services, whether they're created by `startService()` or `bindService()`.

- The **active lifetime** of a service begins with a call to either `onStartCommand()` or `onBind()`. Each method is handed the `Intent` that was passed to either `startService()` or `bindService()`, respectively.

If the service is started, the active lifetime ends the same time that the entire lifetime ends (the service is still active even after `onStartCommand()` returns). If the service is bound, the active lifetime ends when `onUnbind()` returns.

**Note:** Although a started service is stopped by a call to either `stopSelf()` or `stopService()`, there is not a respective callback for the service (there's no `onStop()` callback). So, unless the service is bound to a client, the system destroys it when the service is stopped—`onDestroy()` is the only callback received.

Figure 2 illustrates the typical callback methods for a service. Although the figure separates services that are created by `startService()` from those created by `bindService()`, keep in mind that any service, no matter how it's started, can potentially allow clients to bind to it. So, a service that was initially started with `onStartCommand()` (by a client calling `startService()`) can still receive a call to `onBind()` (when a client calls `bindService()`).

For more information about creating a service that provides binding, see the [Bound Services](#) document, which includes more information about the `onRebind()` callback method in the section about [Managing the Lifecycle of a Bound Service](#).

# Content Providers

## Topics

1. [Content Provider Basics](#)
2. [Creating a Content Provider](#)
3. [Calendar Provider](#)
4. [Contacts Provider](#)

## Related Samples

1. [Contact Manager](#) application
2. ["Cursor \(People\)"](#)
3. ["Cursor \(Phones\)"](#)
4. [Sample Sync Adapter](#)

Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.

When you want to access data in a content provider, you use the [ContentResolver](#) object in your application's [Context](#) to communicate with the provider as a client. The [ContentResolver](#) object communicates with the provider object, an instance of a class that implements [ContentProvider](#). The provider object receives data requests from clients, performs the requested action, and returns the results.

You don't need to develop your own provider if you don't intend to share your data with other applications. However, you do need your own provider to provide custom search suggestions in your own application. You also need your own provider if you want to copy and paste complex data or files from your application to other applications.

Android itself includes content providers that manage data such as audio, video, images, and personal contact information. You can see some of them listed in the reference documentation for the [android.provider](#) package. With some restrictions, these providers are accessible to any Android application.

The following topics describe content providers in more detail:

### [Content Provider Basics](#)

How to access data in a content provider when the data is organized in tables.

### [Creating a Content Provider](#)

How to create your own content provider.

### [Calendar Provider](#)

How to access the Calendar Provider that is part of the Android platform.

### [Contacts Provider](#)

How to access the Contacts Provider that is part of the Android platform.

# In this document

1. [What is API Level?](#)
2. [Uses of API Level in Android](#)
3. [Development Considerations](#)
  1. [Application forward compatibility](#)
  2. [Application backward compatibility](#)
  3. [Selecting a platform version and API Level](#)
  4. [Declaring a minimum API Level](#)
  5. [Testing against higher API Levels](#)
4. [Using a Provisional API Level](#)
5. [Filtering the Reference Documentation by API Level](#)



## Google Play Filtering

Google Play uses the `<uses-sdk>` attributes declared in your app manifest to filter your app from devices that do not meet its platform version requirements. Before setting these attributes, make sure that you understand [Google Play filters](#).

### syntax:

```
<uses-sdk android:minSdkVersion="integer"  
        android:targetSdkVersion="integer"  
        android:maxSdkVersion="integer" />
```

### contained in:

[`<manifest>`](#)

### description:

Lets you express an application's compatibility with one or more versions of the Android platform, by means of an API Level integer. The API Level expressed by an application will be compared to the API Level of a given Android system, which may vary among different Android devices.

Despite its name, this element is used to specify the API Level, *not* the version number of the SDK (software development kit) or Android platform. The API Level is always a single integer. You cannot derive the API Level from its associated Android version number (for example, it is not the same as the major version or the sum of the major and minor versions).

Also read the document about [Versioning Your Applications](#).

### attributes:

`android:minSdkVersion`

An integer designating the minimum API Level required for the application to run. The Android system will prevent the user from installing the application if the system's API Level is lower than the value specified in this attribute. You should always declare this attribute.

**Caution:** If you do not declare this attribute, the system assumes a default value of "1", which indicates that your application is compatible with all versions of Android. If your application is *not* compatible with all versions (for instance, it uses APIs introduced in API Level 3) and you have not declared the proper `minSdkVersion`, then when installed on a system with an API Level less than 3,

the application will crash during runtime when attempting to access the unavailable APIs. For this reason, be certain to declare the appropriate API Level in the `minSdkVersion` attribute.

#### **android:targetSdkVersion**

An integer designating the API Level that the application targets. If not set, the default value equals that given to `minSdkVersion`.

This attribute informs the system that you have tested against the target version and the system should not enable any compatibility behaviors to maintain your app's forward-compatibility with the target version. The application is still able to run on older versions (down to `minSdkVersion`).

As Android evolves with each new version, some behaviors and even appearances might change. However, if the API level of the platform is higher than the version declared by your app's `targetSdkVersion`, the system may enable compatibility behaviors to ensure that your app continues to work the way you expect. You can disable such compatibility behaviors by specifying `targetSdkVersion` to match the API level of the platform on which it's running. For example, setting this value to "11" or higher allows the system to apply a new default theme (Holo) to your app when running on Android 3.0 or higher and also disables [screen compatibility mode](#) when running on larger screens (because support for API level 11 implicitly supports larger screens).

There are many compatibility behaviors that the system may enable based on the value you set for this attribute. Several of these behaviors are described by the corresponding platform versions in the [Build.VERSION\\_CODES](#) reference.

To maintain your application along with each Android release, you should increase the value of this attribute to match the latest API level, then thoroughly test your application on the corresponding platform version.

Introduced in: API Level 4

#### **android:maxSdkVersion**

An integer designating the maximum API Level on which the application is designed to run.

In Android 1.5, 1.6, 2.0, and 2.0.1, the system checks the value of this attribute when installing an application and when re-validating the application after a system update. In either case, if the application's `maxSdkVersion` attribute is lower than the API Level used by the system itself, then the system will not allow the application to be installed. In the case of re-validation after system update, this effectively removes your application from the device.

To illustrate how this attribute can affect your application after system updates, consider the following example:

An application declaring `maxSdkVersion="5"` in its manifest is published on Google Play. A user whose device is running Android 1.6 (API Level 4) downloads and installs the app. After a few weeks, the user receives an over-the-air system update to Android 2.0 (API Level 5). After the update is installed, the system checks the application's `maxSdkVersion` and successfully re-validates it. The application functions as normal. However, some time later, the device receives another system update, this time to Android 2.0.1 (API Level 6). After the update, the system can no longer re-validate the application because the system's own API Level (6) is now higher than the maximum supported by the application (5). The system prevents the application from being visible to the user, in effect removing it from the device.

**Warning:** Declaring this attribute is not recommended. First, there is no need to set the attribute as means of blocking deployment of your application onto new versions of the Android platform as they

are released. By design, new versions of the platform are fully backward-compatible. Your application should work properly on new versions, provided it uses only standard APIs and follows development best practices. Second, note that in some cases, declaring the attribute can **result in your application being removed from users' devices after a system update** to a higher API Level. Most devices on which your application is likely to be installed will receive periodic system updates over the air, so you should consider their effect on your application before setting this attribute.

Introduced in: API Level 4

Future versions of Android (beyond Android 2.0.1) will no longer check or enforce the `maxSdkVersion` attribute during installation or re-validation. Google Play will continue to use the attribute as a filter, however, when presenting users with applications available for download.

**introduced in:**

API Level 1

## What is API Level?

API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.

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

- A core set of packages and classes
- A set of XML elements and attributes for declaring a manifest file
- A set of XML elements and attributes for declaring and accessing resources
- A set of Intents
- A set of permissions that applications can request, as well as permission enforcements included in the system

Each successive version of the Android platform can include updates to the Android application framework API that it delivers.

Updates to the framework API are designed so that the new API remains compatible with earlier versions of the API. That is, most changes in the API are additive and introduce new or replacement functionality. As parts of the API are upgraded, the older replaced parts are deprecated but are not removed, so that existing applications can still use them. In a very small number of cases, parts of the API may be modified or removed, although typically such changes are only needed to ensure API robustness and application or system security. All other API parts from earlier revisions are carried forward without modification.

The framework API that an Android platform delivers is specified using an integer identifier called "API Level". Each Android platform version supports exactly one API Level, although support is implicit for all earlier API Levels (down to API Level 1). The initial release of the Android platform provided API Level 1 and subsequent releases have incremented the API Level.

The table below specifies the API Level supported by each version of the Android platform. For information about the relative numbers of devices that are running each version, see the [Platform Versions dashboards page](#).

Platform Version	API Level	VERSION_CODE	Notes
<a href="#">Android 4.3</a>	<a href="#">18</a>	<a href="#">JELLY_BEAN_MR2</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.2, 4.2.2</a>	<a href="#">17</a>	<a href="#">JELLY_BEAN_MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.1, 4.1.1</a>	<a href="#">16</a>	<a href="#">JELLY_BEAN</a>	<a href="#">Platform Highlights</a>

<a href="#">Android 4.0.3, 4.0.4</a>	<a href="#">15</a>	<a href="#">ICE CREAM SANDWICH MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.0, 4.0.1, 4.0.2</a>	<a href="#">14</a>	<a href="#">ICE CREAM SANDWICH</a>	
<a href="#">Android 3.2</a>	<a href="#">13</a>	<a href="#">HONEYCOMB MR2</a>	
<a href="#">Android 3.1.x</a>	<a href="#">12</a>	<a href="#">HONEYCOMB MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 3.0.x</a>	<a href="#">11</a>	<a href="#">HONEYCOMB</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.4</a>	<a href="#">10</a>	<a href="#">GINGERBREAD MR1</a>	
<a href="#">Android 2.3.3</a>			
<a href="#">Android 2.3.2</a>			<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.1</a>	<a href="#">9</a>	<a href="#">GINGERBREAD</a>	
<a href="#">Android 2.3</a>			
<a href="#">Android 2.2.x</a>	<a href="#">8</a>	<a href="#">FROYO</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.1.x</a>	<a href="#">7</a>	<a href="#">ECLAIR_MR1</a>	
<a href="#">Android 2.0.1</a>	<a href="#">6</a>	<a href="#">ECLAIR_0_1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.0</a>	<a href="#">5</a>	<a href="#">ECLAIR</a>	
<a href="#">Android 1.6</a>	<a href="#">4</a>	<a href="#">DONUT</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.5</a>	<a href="#">3</a>	<a href="#">CUPCAKE</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.1</a>	<a href="#">2</a>	<a href="#">BASE_1_1</a>	
Android 1.0	<a href="#">1</a>	<a href="#">BASE</a>	

## Uses of API Level in Android

The API Level identifier serves a key role in ensuring the best possible experience for users and application developers:

- It lets the Android platform describe the maximum framework API revision that it supports
- It lets applications describe the framework API revision that they require
- It lets the system negotiate the installation of applications on the user's device, such that version-incompatible applications are not installed.

Each Android platform version stores its API Level identifier internally, in the Android system itself.

Applications can use a manifest element provided by the framework API — `<uses-sdk>` — to describe the minimum and maximum API Levels under which they are able to run, as well as the preferred API Level that they are designed to support. The element offers three key attributes:

- `android:minSdkVersion` — Specifies the minimum API Level on which the application is able to run. The default value is "1".
- `android:targetSdkVersion` — Specifies the API Level on which the application is designed to run. In some cases, this allows the application to use manifest elements or behaviors defined in the target API Level, rather than being restricted to using only those defined for the minimum API Level.
- `android:maxSdkVersion` — Specifies the maximum API Level on which the application is able to run. **Important:** Please read the [`<uses-sdk>`](#) documentation before using this attribute.

For example, to specify the minimum system API Level that an application requires in order to run, the application would include in its manifest a `<uses-sdk>` element with a `android:minSdkVersion` attribute. The value of `android:minSdkVersion` would be the integer corresponding to the API Level of the earliest version of the Android platform under which the application can run.

When the user attempts to install an application, or when revalidating an application after a system update, the Android system first checks the `<uses-sdk>` attributes in the application's manifest and compares the values against its own internal API Level. The system allows the installation to begin only if these conditions are met:

- If a `android:minSdkVersion` attribute is declared, its value must be less than or equal to the system's API Level integer. If not declared, the system assumes that the application requires API Level 1.
- If a `android:maxSdkVersion` attribute is declared, its value must be equal to or greater than the system's API Level integer. If not declared, the system assumes that the application has no maximum API Level. Please read the [`<uses-sdk>`](#) documentation for more information about how the system handles this attribute.

When declared in an application's manifest, a `<uses-sdk>` element might look like this:

```
<manifest>
  <uses-sdk android:minSdkVersion="5" />
  ...
</manifest>
```

The principal reason that an application would declare an API Level in `android:minSdkVersion` is to tell the Android system that it is using APIs that were *introduced* in the API Level specified. If the application were to be somehow installed on a platform with a lower API Level, then it would crash at run-time when it tried to access APIs that don't exist. The system prevents such an outcome by not allowing the application to be installed if the lowest API Level it requires is higher than that of the platform version on the target device.

For example, the [`android.appwidget`](#) package was introduced with API Level 3. If an application uses that API, it must declare a `android:minSdkVersion` attribute with a value of "3". The application will then be installable on platforms such as Android 1.5 (API Level 3) and Android 1.6 (API Level 4), but not on the Android 1.1 (API Level 2) and Android 1.0 platforms (API Level 1).

For more information about how to specify an application's API Level requirements, see the [`<uses-sdk>`](#) section of the manifest file documentation.

## Development Considerations

The sections below provide information related to API level that you should consider when developing your application.

### Application forward compatibility

Android applications are generally forward-compatible with new versions of the Android platform.

Because almost all changes to the framework API are additive, an Android application developed using any given version of the API (as specified by its API Level) is forward-compatible with later versions of the Android platform and higher API levels. The application should be able to run on all later versions of the Android platform, except in isolated cases where the application uses a part of the API that is later removed for some reason.

Forward compatibility is important because many Android-powered devices receive over-the-air (OTA) system updates. The user may install your application and use it successfully, then later receive an OTA update to a new version of the Android platform. Once the update is installed, your application will run in a new run-time version of the environment, but one that has the API and system capabilities that your application depends on.

In some cases, changes *below* the API, such those in the underlying system itself, may affect your application when it is run in the new environment. For that reason it's important for you, as the application developer, to understand how the application will look and behave in each system environment. To help you test your application on various versions of the Android platform, the Android SDK includes multiple platforms that you can

download. Each platform includes a compatible system image that you can run in an AVD, to test your application.

## Application backward compatibility

Android applications are not necessarily backward compatible with versions of the Android platform older than the version against which they were compiled.

Each new version of the Android platform can include new framework APIs, such as those that give applications access to new platform capabilities or replace existing API parts. The new APIs are accessible to applications when running on the new platform and, as mentioned above, also when running on later versions of the platform, as specified by API Level. Conversely, because earlier versions of the platform do not include the new APIs, applications that use the new APIs are unable to run on those platforms.

Although it's unlikely that an Android-powered device would be downgraded to a previous version of the platform, it's important to realize that there are likely to be many devices in the field that run earlier versions of the platform. Even among devices that receive OTA updates, some might lag and might not receive an update for a significant amount of time.

## Selecting a platform version and API Level

When you are developing your application, you will need to choose the platform version against which you will compile the application. In general, you should compile your application against the lowest possible version of the platform that your application can support.

You can determine the lowest possible platform version by compiling the application against successively lower build targets. After you determine the lowest version, you should create an AVD using the corresponding platform version (and API Level) and fully test your application. Make sure to declare a `android:minSdkVersion` attribute in the application's manifest and set its value to the API Level of the platform version.

## Declaring a minimum API Level

If you build an application that uses APIs or system features introduced in the latest platform version, you should set the `android:minSdkVersion` attribute to the API Level of the latest platform version. This ensures that users will only be able to install your application if their devices are running a compatible version of the Android platform. In turn, this ensures that your application can function properly on their devices.

If your application uses APIs introduced in the latest platform version but does *not* declare a `android:minSdkVersion` attribute, then it will run properly on devices running the latest version of the platform, but *not* on devices running earlier versions of the platform. In the latter case, the application will crash at runtime when it tries to use APIs that don't exist on the earlier versions.

## Testing against higher API Levels

After compiling your application, you should make sure to test it on the platform specified in the application's `android:minSdkVersion` attribute. To do so, create an AVD that uses the platform version required by your application. Additionally, to ensure forward-compatibility, you should run and test the application on all platforms that use a higher API Level than that used by your application.

The Android SDK includes multiple platform versions that you can use, including the latest version, and provides an updater tool that you can use to download other platform versions as necessary.

To access the updater, use the `android` command-line tool, located in the `<sdk>/tools` directory. You can launch the SDK updater by executing `android sdk`. You can also simply double-click the `android.bat` (Windows) or `android` (OS X/Linux) file. In ADT, you can also access the updater by selecting **Window > Android SDK Manager**.

To run your application against different platform versions in the emulator, create an AVD for each platform version that you want to test. For more information about AVDs, see [Creating and Managing Virtual Devices](#). If you are using a physical device for testing, ensure that you know the API Level of the Android platform it runs. See the table at the top of this document for a list of platform versions and their API Levels.

## Using a Provisional API Level

In some cases, an "Early Look" Android SDK platform may be available. To let you begin developing on the platform although the APIs may not be final, the platform's API Level integer will not be specified. You must instead use the platform's *provisional API Level* in your application manifest, in order to build applications against the platform. A provisional API Level is not an integer, but a string matching the codename of the unreleased platform version. The provisional API Level will be specified in the release notes for the Early Look SDK release notes and is case-sensitive.

The use of a provisional API Level is designed to protect developers and device users from inadvertently publishing or installing applications based on the Early Look framework API, which may not run properly on actual devices running the final system image.

The provisional API Level will only be valid while using the Early Look SDK and can only be used to run applications in the emulator. An application using the provisional API Level can never be installed on an Android device. At the final release of the platform, you must replace any instances of the provisional API Level in your application manifest with the final platform's actual API Level integer.

## Filtering the Reference Documentation by API Level

Reference documentation pages on the Android Developers site offer a "Filter by API Level" control in the top-right area of each page. You can use the control to show documentation only for parts of the API that are actually accessible to your application, based on the API Level that it specifies in the `android:minSdkVersion` attribute of its manifest file.

To use filtering, select the checkbox to enable filtering, just below the page search box. Then set the "Filter by API Level" control to the same API Level as specified by your application. Notice that APIs introduced in a later API Level are then grayed out and their content is masked, since they would not be accessible to your application.

Filtering by API Level in the documentation does not provide a view of what is new or introduced in each API Level — it simply provides a way to view the entire API associated with a given API Level, while excluding API elements introduced in later API Levels.

If you decide that you don't want to filter the API documentation, just disable the feature using the checkbox. By default, API Level filtering is disabled, so that you can view the full framework API, regardless of API Level.

Also note that the reference documentation for individual API elements specifies the API Level at which each element was introduced. The API Level for packages and classes is specified as "Since <api level>" at the top-right corner of the content area on each documentation page. The API Level for class members is specified in their detailed description headers, at the right margin.

# <application>

**syntax:**

```
<application android:allowTaskReparenting=["true" | "false"]
    android:allowBackup=["true" | "false"]
    android:backupAgent="string"
    android:debuggable=["true" | "false"]
    android:description="string resource"
    android:enabled=["true" | "false"]
    android:hasCode=["true" | "false"]
    android:hardwareAccelerated=["true" | "false"]
    android:icon="drawable resource"
    android:killAfterRestore=["true" | "false"]
    android:largeHeap=["true" | "false"]
    android:label="string resource"
    android:logo="drawable resource"
    android:manageSpaceActivity="string"
    android:name="string"
    android:permission="string"
    android:persistent=["true" | "false"]
    android:process="string"
    android:restoreAnyVersion=["true" | "false"]
    android:requiredAccountType="string"
    android:restrictedAccountType="string"
    android:supportsRtl=["true" | "false"]
    android:taskAffinity="string"
    android:testOnly=["true" | "false"]
    android:theme="resource or theme"
    android:uiOptions=["none" | "splitActionBarWhenNarrow"]
    android:vmSafeMode=["true" | "false"] >
    . . .
</application>
```

**contained in:**

[<manifest>](#)

**can contain:**

[<activity>](#)  
[<activity-alias>](#)  
[<service>](#)  
[<receiver>](#)  
[<provider>](#)  
[<uses-library>](#)

**description:**

The declaration of the application. This element contains subelements that declare each of the application's components and has attributes that can affect all the components. Many of these attributes (such as icon, label, permission, process, taskAffinity, and allowTaskReparenting) set default values for corresponding attributes of the component elements. Others (such as debuggable, enabled, description, and allowClearUserData) set values for the application as a whole and cannot be overridden by the components.

## attributes

### **android:allowTaskReparenting**

Whether or not activities that the application defines can move from the task that started them to the task they have an affinity for when that task is next brought to the front — "true" if they can move, and "false" if they must remain with the task where they started. The default value is "false".

The [`<activity>`](#) element has its own [`allowTaskReparenting`](#) attribute that can override the value set here. See that attribute for more information.

### **android:allowBackup**

Whether to allow the application to participate in the backup and restore infrastructure. If this attribute is set to false, no backup or restore of the application will ever be performed, even by a full-system backup that would otherwise cause all application data to be saved via adb. The default value of this attribute is true.

### **android:backupAgent**

The name of the class that implement's the application's backup agent, a subclass of [`BackupAgent`](#). The attribute value should be a fully qualified class name (such as, "`com.example.project.MyBackupAgent`"). However, as a shorthand, if the first character of the name is a period (for example, "`.MyBackupAgent`"), it is appended to the package name specified in the [`<manifest>`](#) element.

There is no default. The name must be specified.

### **android:debuggable**

Whether or not the application can be debugged, even when running on a device in user mode — "true" if it can be, and "false" if not. The default value is "false".

### **android:description**

User-readable text about the application, longer and more descriptive than the application label. The value must be set as a reference to a string resource. Unlike the label, it cannot be a raw string. There is no default value.

### **android:enabled**

Whether or not the Android system can instantiate components of the application — "true" if it can, and "false" if not. If the value is "true", each component's `enabled` attribute determines whether that component is enabled or not. If the value is "false", it overrides the component-specific values; all components are disabled.

The default value is "true".

### **android:hasCode**

Whether or not the application contains any code — "true" if it does, and "false" if not. When the value is "false", the system does not try to load any application code when launching components. The default value is "true".

An application would not have any code of its own only if it's using nothing but built-in component classes, such as an activity that uses the [`AliasActivity`](#) class, a rare occurrence.

### **android:hardwareAccelerated**

Whether or not hardware-accelerated rendering should be enabled for all activities and views in this application — "true" if it should be enabled, and "false" if not. The default value is "true" if

you've set either `minSdkVersion` or `targetSdkVersion` to "14" or higher; otherwise, it's "false".

Starting from Android 3.0 (API level 11), a hardware-accelerated OpenGL renderer is available to applications, to improve performance for many common 2D graphics operations. When the hardware-accelerated renderer is enabled, most operations in Canvas, Paint, Xfermode, ColorFilter, Shader, and Camera are accelerated. This results in smoother animations, smoother scrolling, and improved responsiveness overall, even for applications that do not explicitly make use of the framework's OpenGL libraries.

Note that not all of the OpenGL 2D operations are accelerated. If you enable the hardware-accelerated renderer, test your application to ensure that it can make use of the renderer without errors.

For more information, read the [Hardware Acceleration](#) guide.

#### **android:icon**

An icon for the application as whole, and the default icon for each of the application's components. See the individual `icon` attributes for [`<activity>`](#), [`<activity-alias>`](#), [`<service>`](#), [`<receiver>`](#), and [`<provider>`](#) elements.

This attribute must be set as a reference to a drawable resource containing the image (for example "`@drawable/icon`"). There is no default icon.

#### **android:killAfterRestore**

Whether the application in question should be terminated after its settings have been restored during a full-system restore operation. Single-package restore operations will never cause the application to be shut down. Full-system restore operations typically only occur once, when the phone is first set up. Third-party applications will not normally need to use this attribute.

The default is `true`, which means that after the application has finished processing its data during a full-system restore, it will be terminated.

#### **android:largeHeap**

Whether your application's processes should be created with a large Dalvik heap. This applies to all processes created for the application. It only applies to the first application loaded into a process; if you're using a shared user ID to allow multiple applications to use a process, they all must use this option consistently or they will have unpredictable results.

Most apps should not need this and should instead focus on reducing their overall memory usage for improved performance. Enabling this also does not guarantee a fixed increase in available memory, because some devices are constrained by their total available memory.

To query the available memory size at runtime, use the methods [`getMemoryClass\(\)`](#) or [`getLargeMemoryClass\(\)`](#).

#### **android:label**

A user-readable label for the application as a whole, and a default label for each of the application's components. See the individual `label` attributes for [`<activity>`](#), [`<activity-alias>`](#), [`<service>`](#), [`<receiver>`](#), and [`<provider>`](#) elements.

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

## **android:logo**

A logo for the application as whole, and the default logo for activities.

This attribute must be set as a reference to a drawable resource containing the image (for example "@drawable/logo"). There is no default logo.

## **android:manageSpaceActivity**

The fully qualified name of an Activity subclass that the system can launch to let users manage the memory occupied by the application on the device. The activity should also be declared with an [`<activity>`](#) element.

## **android:name**

The fully qualified name of an [Application](#) subclass implemented for the application. When the application process is started, this class is instantiated before any of the application's components.

The subclass is optional; most applications won't need one. In the absence of a subclass, Android uses an instance of the base Application class.

## **android:permission**

The name of a permission that clients must have in order to interact with the application. This attribute is a convenient way to set a permission that applies to all of the application's components. It can be overwritten by setting the `permission` attributes of individual components.

For more information on permissions, see the [Permissions](#) section in the introduction and another document, [Security and Permissions](#).

## **android:persistent**

Whether or not the application should remain running at all times — "true" if it should, and "false" if not. The default value is "false". Applications should not normally set this flag; persistence mode is intended only for certain system applications.

## **android:process**

The name of a process where all components of the application should run. Each component can override this default by setting its own `process` attribute.

By default, Android creates a process for an application when the first of its components needs to run. All components then run in that process. The name of the default process matches the package name set by the [`<manifest>`](#) element.

By setting this attribute to a process name that's shared with another application, you can arrange for components of both applications to run in the same process — but only if the two applications also share a user ID and be signed with the same certificate.

If the name assigned to this attribute begins with a colon (:), a new process, private to the application, is created when it's needed. If the process name begins with a lowercase character, a global process of that name is created. A global process can be shared with other applications, reducing resource usage.

## **android:restoreAnyVersion**

Indicates that the application is prepared to attempt a restore of any backed-up data set, even if the backup was stored by a newer version of the application than is currently installed on the device. Setting this attribute to `true` will permit the Backup Manager to attempt restore even when a version mismatch suggests that the data are incompatible. *Use with caution!*

The default value of this attribute is `false`.

### **android:requiredAccountType**

Specifies the account type required by the application in order to function. If your app requires an [Account](#), the value for this attribute must correspond to the account authenticator type used by your app (as defined by [AuthenticatorDescription](#)), such as "com.google".

The default value is null and indicates that the application can work *without* any accounts.

Because restricted profiles currently cannot add accounts, specifying this attribute **makes your app unavailable from a restricted profile** unless you also declare [an-android:restrictedAccountType](#) with the same value.

**Caution:** If the account data may reveal personally identifiable information, it's important that you declare this attribute and leave [android:restrictedAccountType](#) null, so that restricted profiles cannot use your app to access personal information that belongs to the owner user.

This attribute was added in API level 18.

### **android:restrictedAccountType**

Specifies the account type required by this application and indicates that restricted profiles are allowed to access such accounts that belong to the owner user. If your app requires an [Account](#) and restricted profiles **are allowed to access** the primary user's accounts, the value for this attribute must correspond to the account authenticator type used by your app (as defined by [AuthenticatorDescription](#)), such as "com.google".

The default value is null and indicates that the application can work *without* any accounts.

**Caution:** Specifying this attribute allows restricted profiles to use your app with accounts that belong to the owner user, which may reveal personally identifiable information. If the account may reveal personal details, you **should not** use this attribute and you should instead declare the [an-android:requiredAccountType](#) attribute to make your app unavailable to restricted profiles.

This attribute was added in API level 18.

### **android:supportsRtl**

Declares whether your application is willing to support right-to-left (RTL) layouts.

If set to `true` and [targetSdkVersion](#) is set to 17 or higher, various RTL APIs will be activated and used by the system so your app can display RTL layouts. If set to `false` or if [targetSdkVersion](#) is set to 16 or lower, the RTL APIs will be ignored or will have no effect and your app will behave the same regardless of the layout direction associated to the user's Locale choice (your layouts will always be left-to-right).

The default value of this attribute is `false`.

This attribute was added in API level 17.

### **android:taskAffinity**

An affinity name that applies to all activities within the application, except for those that set a different affinity with their own [taskAffinity](#) attributes. See that attribute for more information.

By default, all activities within an application share the same affinity. The name of that affinity is the same as the package name set by the [<manifest>](#) element.

**android:testOnly**

Indicates whether this application is only for testing purposes. For example, it may expose functionality or data outside of itself that would cause a security hole, but is useful for testing. This kind of application can be installed only through adb.

**android:theme**

A reference to a style resource defining a default theme for all activities in the application. Individual activities can override the default by setting their own [theme](#) attributes. For more information, see the [Styles and Themes](#) developer guide.

**android:uiOptions**

Extra options for an activity's UI.

Must be one of the following values.

Value	Description
"none"	No extra UI options. This is the default.
"splitActionBarWhenNarrow"	Add a bar at the bottom of the screen to display action items in the <a href="#">ActionBar</a> , when constrained for horizontal space (such as when in portrait mode on a handset). Instead of a small number of action items appearing in the action bar at the top of the screen, the action bar is split into the top navigation section and the bottom bar for action items. This ensures a reasonable amount of space is made available not only for the action items, but also for navigation and title elements at the top. Menu items are not split across the two bars; they always appear together.

For more information about the action bar, see the [Action Bar](#) developer guide.

This attribute was added in API level 14.

**android:vmSafeMode**

Indicates whether the app would like the virtual machine (VM) to operate in safe mode. The default value is "false".

**introduced in:**

API Level 1

**see also:**

[`<activity>`](#)  
[`<service>`](#)  
[`<receiver>`](#)  
[`<provider>`](#)

# <activity>

syntax:

```
<activity android:allowTaskReparenting=["true" | "false"]
          android:alwaysRetainTaskState=["true" | "false"]
          android:clearTaskOnLaunch=["true" | "false"]
          android:configChanges=["mcc", "mnc", "locale",
                                 "touchscreen", "keyboard", "keyboardHidden",
                                 "navigation", "screenLayout", "fontScale",
                                 "orientation", "screenSize", "smallestScreen"
                                 "size", "density"]
          android:enabled=["true" | "false"]
          android:excludeFromRecents=["true" | "false"]
          android:exported=["true" | "false"]
          android:finishOnTaskLaunch=["true" | "false"]
          android:hardwareAccelerated=["true" | "false"]
          android:icon="drawable resource"
          android:label="string resource"
          android:launchMode=["multiple" | "singleTop" |
                             "singleTask" | "singleInstance"]
          android:multiprocess=["true" | "false"]
          android:name="string"
          android:noHistory=["true" | "false"]
          android:parentActivityName="string"
          android:permission="string"
          android:process="string"
          android:screenOrientation=["unspecified" | "behind" |
                                     "landscape" | "portrait" |
                                     "reverseLandscape" | "reversePortrait" |
                                     "sensorLandscape" | "sensorPortrait" |
                                     "userLandscape" | "userPortrait" |
                                     "sensor" | "fullSensor" | "nosensor" |
                                     "user" | "fullUser" | "locked"]
          android:stateNotNeeded=["true" | "false"]
          android:taskAffinity="string"
          android:theme="resource or theme"
          android:uiOptions=["none" | "splitActionBarWhenNarrow"]
          android:windowSoftInputMode=["stateUnspecified",
                                       "stateUnchanged", "stateHidden",
                                       "stateAlwaysHidden", "stateVisible",
                                       "stateAlwaysVisible", "adjustUnspecified",
                                       "adjustResize", "adjustPan"] >
    . . .
</activity>
```

contained in:

[<application>](#)

can contain:

[<intent-filter>](#)  
[<meta-data>](#)

## **description:**

Declares an activity (an [Activity](#) subclass) that implements part of the application's visual user interface. All activities must be represented by <activity> elements in the manifest file. Any that are not declared there will not be seen by the system and will never be run.

## **attributes:**

### **android:allowTaskReparenting**

Whether or not the activity can move from the task that started it to the task it has an affinity for when that task is next brought to the front — "true" if it can move, and "false" if it must remain with the task where it started.

If this attribute is not set, the value set by the corresponding [allowTaskReparenting](#) attribute of the [<application>](#) element applies to the activity. The default value is "false".

Normally when an activity is started, it's associated with the task of the activity that started it and it stays there for its entire lifetime. You can use this attribute to force it to be re-parented to the task it has an affinity for when its current task is no longer displayed. Typically, it's used to cause the activities of an application to move to the main task associated with that application.

For example, if an e-mail message contains a link to a web page, clicking the link brings up an activity that can display the page. That activity is defined by the browser application, but is launched as part of the e-mail task. If it's reparented to the browser task, it will be shown when the browser next comes to the front, and will be absent when the e-mail task again comes forward.

The affinity of an activity is defined by the [taskAffinity](#) attribute. The affinity of a task is determined by reading the affinity of its root activity. Therefore, by definition, a root activity is always in a task with the same affinity. Since activities with "singleTask" or "singleInstance" launch modes can only be at the root of a task, re-parenting is limited to the "standard" and "singleTop" modes. (See also the [launchMode](#) attribute.)

### **android:alwaysRetainTaskState**

Whether or not the state of the task that the activity is in will always be maintained by the system — "true" if it will be, and "false" if the system is allowed to reset the task to its initial state in certain situations. The default value is "false". This attribute is meaningful only for the root activity of a task; it's ignored for all other activities.

Normally, the system clears a task (removes all activities from the stack above the root activity) in certain situations when the user re-selects that task from the home screen. Typically, this is done if the user hasn't visited the task for a certain amount of time, such as 30 minutes.

However, when this attribute is "true", users will always return to the task in its last state, regardless of how they get there. This is useful, for example, in an application like the web browser where there is a lot of state (such as multiple open tabs) that users would not like to lose.

### **android:clearTaskOnLaunch**

Whether or not all activities will be removed from the task, except for the root activity, whenever it is re-launched from the home screen — "true" if the task is always stripped down to its root activity, and "false" if not. The default value is "false". This attribute is meaningful only for activities that start a new task (the root activity); it's ignored for all other activities in the task.

When the value is "true", every time users start the task again, they are brought to its root activity regardless of what they were last doing in the task and regardless of whether they used the *Back* or *Home* button to leave it. When the value is "false", the task may be cleared of activities in some situations (see the [alwaysRetainTaskState](#) attribute), but not always.

Suppose, for example, that someone launches activity P from the home screen, and from there goes to activity Q. The user next presses *Home*, and then returns to activity P. Normally, the user would see activity Q, since that is what they were last doing in P's task. However, if P set this flag to "true", all of the activities on top of it (Q in this case) were removed when the user pressed *Home* and the task went to the background. So the user sees only P when returning to the task.

If this attribute and [allowTaskReparenting](#) are both "true", any activities that can be re-parented are moved to the task they share an affinity with; the remaining activities are then dropped, as described above.

#### **android:configChanges**

Lists configuration changes that the activity will handle itself. When a configuration change occurs at runtime, the activity is shut down and restarted by default, but declaring a configuration with this attribute will prevent the activity from being restarted. Instead, the activity remains running and its [onConfigurationChanged\(\)](#) method is called.

**Note:** Using this attribute should be avoided and used only as a last resort. Please read [Handling Runtime Changes](#) for more information about how to properly handle a restart due to a configuration change.

Any or all of the following strings are valid values for this attribute. Multiple values are separated by '|' — for example, "locale|navigation|orientation".

Value	Description
"mcc"	The IMSI mobile country code (MCC) has changed — a SIM has been detected and updated the MCC.
"mnc"	The IMSI mobile network code (MNC) has changed — a SIM has been detected and updated the MNC.
"locale"	The locale has changed — the user has selected a new language that text should be displayed in.
"touchscreen"	The touchscreen has changed. (This should never normally happen.)
"keyboard"	The keyboard type has changed — for example, the user has plugged in an external keyboard.
"keyboardHidden"	The keyboard accessibility has changed — for example, the user has revealed the hardware keyboard.
"navigation"	The navigation type (trackball/dpad) has changed. (This should never normally happen.)
"screenLayout"	The screen layout has changed — this might be caused by a different display being activated.
"fontScale"	The font scaling factor has changed — the user has selected a new global font size.
"uiMode"	The user interface mode has changed — this can be caused when the user places the device into a desk/car dock or when the night mode changes. See <a href="#">UiModeManager</a> . <i>Added in API level 8</i> .
"orientation"	The screen orientation has changed — the user has rotated the device.

**Note:** If your application targets API level 13 or higher (as declared by the [minSdkVersion](#) and [targetSdkVersion](#) attributes), then you should also declare the " screenSize" configuration, because it also changes when a device switches between portrait and landscape orientations.

"screenSize"

The current available screen size has changed. This represents a change in the currently available size, relative to the current aspect ratio, so will change when the user switches between landscape and portrait. However, if your application targets API level 12 or lower, then your activity always handles this configuration change itself (this configuration change does not restart your activity, even when running on an Android 3.2 or higher device).

*Added in API level 13.*

"smallestScreenSize"

The physical screen size has changed. This represents a change in size regardless of orientation, so will only change when the actual physical screen size has changed such as switching to an external display. A change to this configuration corresponds to a change in the [smallest-Width configuration](#). However, if your application targets API level 12 or lower, then your activity always handles this configuration change itself (this configuration change does not restart your activity, even when running on an Android 3.2 or higher device).

*Added in API level 13.*

"layoutDirection"

The layout direction has changed. For example, changing from left-to-right (LTR) to right-to-left (RTL). *Added in API level 17.*

All of these configuration changes can impact the resource values seen by the application. Therefore, when [onConfigurationChanged\(\)](#) is called, it will generally be necessary to again retrieve all resources (including view layouts, drawables, and so on) to correctly handle the change.

#### **android:enabled**

Whether or not the activity can be instantiated by the system — "true" if it can be, and "false" if not. The default value is "true".

The [`<application>`](#) element has its own [enabled](#) attribute that applies to all application components, including activities. The [`<application>`](#) and [`<activity>`](#) attributes must both be "true" (as they both are by default) for the system to be able to instantiate the activity. If either is "false", it cannot be instantiated.

#### **android:excludeFromRecents**

Whether or not the task initiated by this activity should be excluded from the list of recently used applications ("recent apps"). That is, when this activity is the root activity of a new task, this attribute determines whether the task should not appear in the list of recent apps. Set "true" if the task should be *excluded* from the list; set "false" if it should be *included*. The default value is "false".

#### **android:exported**

Whether or not the activity can be launched by components of other applications — "true" if it can be, and "false" if not. If "false", the activity can be launched only by components of the same application or applications with the same user ID.

The default value depends on whether the activity contains intent filters. The absence of any filters means that the activity can be invoked only by specifying its exact class name. This implies that the activity is intended only for application-internal use (since others would not know the class name). So in this case, the default value is "false". On the other hand, the presence of at least one filter implies that the activity is intended for external use, so the default value is "true".

This attribute is not the only way to limit an activity's exposure to other applications. You can also use a permission to limit the external entities that can invoke the activity (see the [permission](#) attribute).

#### **android:finishOnTaskLaunch**

Whether or not an existing instance of the activity should be shut down (finished) whenever the user again launches its task (chooses the task on the home screen) — "true" if it should be shut down, and "false" if not. The default value is "false".

If this attribute and [allowTaskReparenting](#) are both "true", this attribute trumps the other. The affinity of the activity is ignored. The activity is not re-parented, but destroyed.

#### **android:hardwareAccelerated**

Whether or not hardware-accelerated rendering should be enabled for this Activity — "true" if it should be enabled, and "false" if not. The default value is "false".

Starting from Android 3.0, a hardware-accelerated OpenGL renderer is available to applications, to improve performance for many common 2D graphics operations. When the hardware-accelerated renderer is enabled, most operations in Canvas, Paint, Xfermode, ColorFilter, Shader, and Camera are accelerated. This results in smoother animations, smoother scrolling, and improved responsiveness overall, even for applications that do not explicitly make use of the framework's OpenGL libraries. Because of the increased resources required to enable hardware acceleration, your app will consume more RAM.

Note that not all of the OpenGL 2D operations are accelerated. If you enable the hardware-accelerated renderer, test your application to ensure that it can make use of the renderer without errors.

#### **android:icon**

An icon representing the activity. The icon is displayed to users when a representation of the activity is required on-screen. For example, icons for activities that initiate tasks are displayed in the launcher window. The icon is often accompanied by a label (see the [android:label](#) attribute).

This attribute must be set as a reference to a drawable resource containing the image definition. If it is not set, the icon specified for the application as a whole is used instead (see the [<application>](#) element's [icon](#) attribute).

The activity's icon — whether set here or by the [<application>](#) element — is also the default icon for all the activity's intent filters (see the [<intent-filter>](#) element's [icon](#) attribute).

#### **android:label**

A user-readable label for the activity. The label is displayed on-screen when the activity must be represented to the user. It's often displayed along with the activity icon.

If this attribute is not set, the label set for the application as a whole is used instead (see the [<application>](#) element's [label](#) attribute).

The activity's label — whether set here or by the [<application>](#) element — is also the default label for all the activity's intent filters (see the [<intent-filter>](#) element's [label](#) attribute).

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

## `android:launchMode`

An instruction on how the activity should be launched. There are four modes that work in conjunction with activity flags (`FLAG_ACTIVITY_*` constants) in [Intent](#) objects to determine what should happen when the activity is called upon to handle an intent. They are:

```
"standard"  
"singleTop"  
"singleTask"  
"singleInstance"
```

The default mode is "standard".

As shown in the table below, the modes fall into two main groups, with "standard" and "singleTop" activities on one side, and "singleTask" and "singleInstance" activities on the other. An activity with the "standard" or "singleTop" launch mode can be instantiated multiple times. The instances can belong to any task and can be located anywhere in the activity stack. Typically, they're launched into the task that called [startActivity\(\)](#) (unless the Intent object contains a `FLAG_ACTIVITY_NEW_TASK` instruction, in which case a different task is chosen — see the [taskAffinity](#) attribute).

In contrast, "singleTask" and "singleInstance" activities can only begin a task. They are always at the root of the activity stack. Moreover, the device can hold only one instance of the activity at a time — only one such task.

The "standard" and "singleTop" modes differ from each other in just one respect: Every time there's a new intent for a "standard" activity, a new instance of the class is created to respond to that intent. Each instance handles a single intent. Similarly, a new instance of a "singleTop" activity may also be created to handle a new intent. However, if the target task already has an existing instance of the activity at the top of its stack, that instance will receive the new intent (in an [onNewIntent\(\)](#) call); a new instance is not created. In other circumstances — for example, if an existing instance of the "singleTop" activity is in the target task, but not at the top of the stack, or if it's at the top of a stack, but not in the target task — a new instance would be created and pushed on the stack.

The "singleTask" and "singleInstance" modes also differ from each other in only one respect: A "singleTask" activity allows other activities to be part of its task. It's always at the root of its task, but other activities (necessarily "standard" and "singleTop" activities) can be launched into that task. A "singleInstance" activity, on the other hand, permits no other activities to be part of its task. It's the only activity in the task. If it starts another activity, that activity is assigned to a different task — as if `FLAG_ACTIVITY_NEW_TASK` was in the intent.

Use Cases	Launch Mode	Multiple Instances?	Comments
Normal launches for most activities	"standard"	Yes	Default. The system always creates a new instance of the activity in the target task and routes the intent to it.
	"singleTop"	Conditionally	If an instance of the activity already exists at the top of the target task, the system routes the intent to that instance through a call to its <a href="#">onNewIntent()</a> method, rather than creating a new instance of the activity.
Specialized launches	"singleTask"	No	The system creates the activity at the root of a new task and routes the intent to it. However, if an

*(not recommended for general use)*

"singleInstance" No

instance of the activity already exists, the system routes the intent to existing instance through a call to its [onNewIntent\(\)](#) method, rather than creating a new one.

Same as "singleTask", except that the system doesn't launch any other activities into the task holding the instance. The activity is always the single and only member of its task.

As shown in the table above, standard is the default mode and is appropriate for most types of activities. SingleTop is also a common and useful launch mode for many types of activities. The other modes — singleTask and singleInstance — are not appropriate for most applications, since they result in an interaction model that is likely to be unfamiliar to users and is very different from most other applications.

Regardless of the launch mode that you choose, make sure to test the usability of the activity during launch and when navigating back to it from other activities and tasks using the *Back* button.

For more information on launch modes and their interaction with Intent flags, see the [Tasks and Back Stack](#) document.

#### **android:multiprocess**

Whether an instance of the activity can be launched into the process of the component that started it — "true" if it can be, and "false" if not. The default value is "false".

Normally, a new instance of an activity is launched into the process of the application that defined it, so all instances of the activity run in the same process. However, if this flag is set to "true", instances of the activity can run in multiple processes, allowing the system to create instances wherever they are used (provided permissions allow it), something that is almost never necessary or desirable.

#### **android:name**

The name of the class that implements the activity, a subclass of [Activity](#). The attribute value should be a fully qualified class name (such as, "com.example.project.ExtracurricularActivity"). However, as a shorthand, if the first character of the name is a period (for example, ".ExtracurricularActivity"), it is appended to the package name specified in the [<manifest>](#) element.

Once you publish your application, you [should not change this name](#) (unless you've set [android:exported="false"](#)).

There is no default. The name must be specified.

#### **android:noHistory**

Whether or not the activity should be removed from the activity stack and finished (its [finish\(\)](#) method called) when the user navigates away from it and it's no longer visible on screen — "true" if it should be finished, and "false" if not. The default value is "false".

A value of "true" means that the activity will not leave a historical trace. It will not remain in the activity stack for the task, so the user will not be able to return to it.

This attribute was introduced in API Level 3.

### **android:parentActivityName**

The class name of the logical parent of the activity. The name here must match the class name given to the corresponding `<activity>` element's [android:name](#) attribute.

The system reads this attribute to determine which activity should be started when the user presses the Up button in the action bar. The system can also use this information to synthesize a back stack of activities with [TaskStackBuilder](#).

To support API levels 4 - 16, you can also declare the parent activity with a `<meta-data>` element that specifies a value for "android.support.PARENT\_ACTIVITY". For example:

```
<activity
    android:name="com.example.app.ChildActivity"
    android:label="@string/title_child_activity"
    android:parentActivityName="com.example.myfirstapp.MainActivity" >
    <!-- Parent activity meta-data to support API level 4+ -->
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value="com.example.app.MainActivity" />
</activity>
```

For more information about declaring the parent activity to support Up navigation, read [Providing Up Navigation](#).

This attribute was introduced in API Level 16.

### **android:permission**

The name of a permission that clients must have to launch the activity or otherwise get it to respond to an intent. If a caller of [startActivity\(\)](#) or [startActivityForResult\(\)](#) has not been granted the specified permission, its intent will not be delivered to the activity.

If this attribute is not set, the permission set by the `<application>` element's [permission](#) attribute applies to the activity. If neither attribute is set, the activity is not protected by a permission.

For more information on permissions, see the [Permissions](#) section in the introduction and another document, [Security and Permissions](#).

### **android:process**

The name of the process in which the activity should run. Normally, all components of an application run in a default process name created for the application and you do not need to use this attribute. But if necessary, you can override the default process name with this attribute, allowing you to spread your app components across multiple processes.

If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the activity runs in that process. If the process name begins with a lowercase character, the activity will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage.

The `<application>` element's [process](#) attribute can set a different default process name for all components.

### **android:screenOrientation**

The orientation of the activity's display on the device.

The value can be any one of the following strings:

"unspecified"	The default value. The system chooses the orientation. The policy it uses, and therefore the choices made in specific contexts, may differ from device to device.
"behind"	The same orientation as the activity that's immediately beneath it in the activity stack.
"landscape"	Landscape orientation (the display is wider than it is tall).
"portrait"	Portrait orientation (the display is taller than it is wide).
"reverseLandscape"	Landscape orientation in the opposite direction from normal landscape. <i>Added in API level 9.</i>
"reversePortrait"	Portrait orientation in the opposite direction from normal portrait. <i>Added in API level 9.</i>
"sensorLandscape"	Landscape orientation, but can be either normal or reverse landscape based on the device sensor. <i>Added in API level 9.</i>
"sensorPortrait"	Portrait orientation, but can be either normal or reverse portrait based on the device sensor. <i>Added in API level 9.</i>
"userLandscape"	Landscape orientation, but can be either normal or reverse landscape based on the device sensor and the user's sensor preference. If the user has locked sensor-based rotation, this behaves the same as <code>landscape</code> , otherwise it behaves the same as <code>sensorLandscape</code> . <i>Added in API level 18.</i>
"userPortrait"	Portrait orientation, but can be either normal or reverse portrait based on the device sensor and the user's sensor preference. If the user has locked sensor-based rotation, this behaves the same as <code>portrait</code> , otherwise it behaves the same as <code>sensorPortrait</code> . <i>Added in API level 18.</i>
"sensor"	The orientation is determined by the device orientation sensor. The orientation of the display depends on how the user is holding the device; it changes when the user rotates the device. Some devices, though, will not rotate to all four possible orientations, by default. To allow all four orientations, use " <code>fullSensor</code> ".
"fullSensor"	The orientation is determined by the device orientation sensor for any of the 4 orientations. This is similar to " <code>sensor</code> " except this allows any of the 4 possible screen orientations, regardless of what the device will normally do (for example, some devices won't normally use reverse portrait or reverse landscape, but this enables those). <i>Added in API level 9.</i>
"nosensor"	The orientation is determined without reference to a physical orientation sensor. The sensor is ignored, so the display will not rotate based on how the user moves the device. Except for this distinction, the system chooses the orientation using the same policy as for the " <code>unspecified</code> " setting.
"user"	The user's current preferred orientation.
"fullUser"	If the user has locked sensor-based rotation, this behaves the same as <code>user</code> , otherwise it behaves the same as <code>fullSensor</code> and allows any of the 4 possible screen orientations. <i>Added in API level 18.</i>
"locked"	Locks the orientation to its current rotation, whatever that is. <i>Added in API level 18.</i>

**Note:** When you declare one of the landscape or portrait values, it is considered a hard requirement for the orientation in which the activity runs. As such, the value you declare enables filtering by services such as Google Play so your application is available only to devices that support the orientation required by your activities. For example, if you declare either "`landscape`", "`reverseLandscape`", or "`sensorLandscape`", then your application will be available only to devices that support landscape orientation. However, you should also explicitly declare that your application re-

quires either portrait or landscape orientation with the `<uses-feature>` element. For example, `<uses-feature android:name="android.hardware.screen.portrait"/>`. This is purely a filtering behavior provided by Google Play (and other services that support it) and the platform itself does not control whether your app can be installed when a device supports only certain orientations.

#### **android:stateNotNeeded**

Whether or not the activity can be killed and successfully restarted without having saved its state — "true" if it can be restarted without reference to its previous state, and "false" if its previous state is required. The default value is "false".

Normally, before an activity is temporarily shut down to save resources, its [onSaveInstanceState\(\)](#) method is called. This method stores the current state of the activity in a [Bundle](#) object, which is then passed to [onCreate\(\)](#) when the activity is restarted. If this attribute is set to "true", `onSaveInstanceState()` may not be called and `onCreate()` will be passed `null` instead of the [Bundle](#) — just as it was when the activity started for the first time.

A "true" setting ensures that the activity can be restarted in the absence of retained state. For example, the activity that displays the home screen uses this setting to make sure that it does not get removed if it crashes for some reason.

#### **android:taskAffinity**

The task that the activity has an affinity for. Activities with the same affinity conceptually belong to the same task (to the same "application" from the user's perspective). The affinity of a task is determined by the affinity of its root activity.

The affinity determines two things — the task that the activity is re-parented to (see the [allowTaskReparenting](#) attribute) and the task that will house the activity when it is launched with the [FLAG\\_ACTIVITY\\_NEW\\_TASK](#) flag.

By default, all activities in an application have the same affinity. You can set this attribute to group them differently, and even place activities defined in different applications within the same task. To specify that the activity does not have an affinity for any task, set it to an empty string.

If this attribute is not set, the activity inherits the affinity set for the application (see the [<application>](#) element's [taskAffinity](#) attribute). The name of the default affinity for an application is the package name set by the [<manifest>](#) element.

#### **android:theme**

A reference to a style resource defining an overall theme for the activity. This automatically sets the activity's context to use this theme (see [setTheme\(\)](#), and may also cause "starting" animations prior to the activity being launched (to better match what the activity actually looks like).

If this attribute is not set, the activity inherits the theme set for the application as a whole — from the [<application>](#) element's [theme](#) attribute. If that attribute is also not set, the default system theme is used. For more information, see the [Styles and Themes](#) developer guide.

#### **android:uiOptions**

Extra options for an activity's UI.

Must be one of the following values.

Value	Description
-------	-------------

"none"	No extra UI options. This is the default.
"splitActionBarWhenNarrow"	Add a bar at the bottom of the screen to display action items in the <a href="#">ActionBar</a> , when constrained for horizontal space (such as when in portrait mode on a handset). Instead of a small number of action items appearing in the action bar at the top of the screen, the action bar is split into the top navigation section and the bottom bar for action items. This ensures a reasonable amount of space is made available not only for the action items, but also for navigation and title elements at the top. Menu items are not split across the two bars; they always appear together.

For more information about the action bar, see the [Action Bar](#) developer guide.

This attribute was added in API level 14.

#### **android:windowSoftInputMode**

How the main window of the activity interacts with the window containing the on-screen soft keyboard. The setting for this attribute affects two things:

- The state of the soft keyboard — whether it is hidden or visible — when the activity becomes the focus of user attention.
- The adjustment made to the activity's main window — whether it is resized smaller to make room for the soft keyboard or whether its contents pan to make the current focus visible when part of the window is covered by the soft keyboard.

The setting must be one of the values listed in the following table, or a combination of one "state..." value plus one "adjust..." value. Setting multiple values in either group — multiple "state..." values, for example — has undefined results. Individual values are separated by a vertical bar (|). For example:

```
<activity android:windowSoftInputMode="stateVisible|adjustResize" . . . >
```

Values set here (other than "stateUnspecified" and "adjustUnspecified") override values set in the theme.

<b>Value</b>	<b>Description</b>
"stateUnspecified"	The state of the soft keyboard (whether it is hidden or visible) is not specified. The system will choose an appropriate state or rely on the setting in the theme.
	This is the default setting for the behavior of the soft keyboard.
"stateUnchanged"	The soft keyboard is kept in whatever state it was last in, whether visible or hidden, when the activity comes to the fore.
"stateHidden"	The soft keyboard is hidden when the user chooses the activity — that is, when the user affirmatively navigates forward to the activity, rather than backs into it because of leaving another activity.
"stateAlwaysHidden"	The soft keyboard is always hidden when the activity's main window has input focus.
"stateVisible"	The soft keyboard is visible when that's normally appropriate (when the user is navigating forward to the activity's main window).

The soft keyboard is made visible when the user chooses the activity — "stateAlwaysVisible" — that is, when the user affirmatively navigates forward to the activity, rather than backs into it because of leaving another activity.

It is unspecified whether the activity's main window resizes to make room for the soft keyboard, or whether the contents of the window pan to make the current focus visible on-screen. The system will automatically select one of these modes depending on whether the content of the window has any layout views that can scroll their contents. If there is such a view, the window will be resized, on the assumption that scrolling can make all of the window's contents visible within a smaller area.

This is the default setting for the behavior of the main window.

"adjustUnspecified" The activity's main window is always resized to make room for the soft keyboard on screen.

"adjustResize" The activity's main window is not resized to make room for the soft keyboard. Rather, the contents of the window are automatically panned so that the current focus is never obscured by the keyboard and users can always see what they are typing. This is generally less desirable than resizing, because the user may need to close the soft keyboard to get at and interact with obscured parts of the window.

This attribute was introduced in API Level 3.

#### **introduced in:**

API Level 1 for all attributes except for [noHistory](#) and [windowSoftInputMode](#), which were added in API Level 3.

#### **see also:**

[<application>](#)  
[<activity-alias>](#)

# <activity>

syntax:

```
<activity android:allowTaskReparenting=["true" | "false"]
          android:alwaysRetainTaskState=["true" | "false"]
          android:clearTaskOnLaunch=["true" | "false"]
          android:configChanges=["mcc", "mnc", "locale",
                                 "touchscreen", "keyboard", "keyboardHidden",
                                 "navigation", "screenLayout", "fontScale",
                                 "orientation", "screenSize", "smallestScreen"
                                 "size", "density"]
          android:enabled=["true" | "false"]
          android:excludeFromRecents=["true" | "false"]
          android:exported=["true" | "false"]
          android:finishOnTaskLaunch=["true" | "false"]
          android:hardwareAccelerated=["true" | "false"]
          android:icon="drawable resource"
          android:label="string resource"
          android:launchMode=["multiple" | "singleTop" |
                             "singleTask" | "singleInstance"]
          android:multiprocess=["true" | "false"]
          android:name="string"
          android:noHistory=["true" | "false"]
          android:parentActivityName="string"
          android:permission="string"
          android:process="string"
          android:screenOrientation=["unspecified" | "behind" |
                                     "landscape" | "portrait" |
                                     "reverseLandscape" | "reversePortrait" |
                                     "sensorLandscape" | "sensorPortrait" |
                                     "userLandscape" | "userPortrait" |
                                     "sensor" | "fullSensor" | "nosensor" |
                                     "user" | "fullUser" | "locked"]
          android:stateNotNeeded=["true" | "false"]
          android:taskAffinity="string"
          android:theme="resource or theme"
          android:uiOptions=["none" | "splitActionBarWhenNarrow"]
          android:windowSoftInputMode=["stateUnspecified",
                                       "stateUnchanged", "stateHidden",
                                       "stateAlwaysHidden", "stateVisible",
                                       "stateAlwaysVisible", "adjustUnspecified",
                                       "adjustResize", "adjustPan"] >
    . . .
</activity>
```

contained in:

[<application>](#)

can contain:

[<intent-filter>](#)  
[<meta-data>](#)

## **description:**

Declares an activity (an [Activity](#) subclass) that implements part of the application's visual user interface. All activities must be represented by <activity> elements in the manifest file. Any that are not declared there will not be seen by the system and will never be run.

## **attributes:**

### **android:allowTaskReparenting**

Whether or not the activity can move from the task that started it to the task it has an affinity for when that task is next brought to the front — "true" if it can move, and "false" if it must remain with the task where it started.

If this attribute is not set, the value set by the corresponding [allowTaskReparenting](#) attribute of the [<application>](#) element applies to the activity. The default value is "false".

Normally when an activity is started, it's associated with the task of the activity that started it and it stays there for its entire lifetime. You can use this attribute to force it to be re-parented to the task it has an affinity for when its current task is no longer displayed. Typically, it's used to cause the activities of an application to move to the main task associated with that application.

For example, if an e-mail message contains a link to a web page, clicking the link brings up an activity that can display the page. That activity is defined by the browser application, but is launched as part of the e-mail task. If it's reparented to the browser task, it will be shown when the browser next comes to the front, and will be absent when the e-mail task again comes forward.

The affinity of an activity is defined by the [taskAffinity](#) attribute. The affinity of a task is determined by reading the affinity of its root activity. Therefore, by definition, a root activity is always in a task with the same affinity. Since activities with "singleTask" or "singleInstance" launch modes can only be at the root of a task, re-parenting is limited to the "standard" and "singleTop" modes. (See also the [launchMode](#) attribute.)

### **android:alwaysRetainTaskState**

Whether or not the state of the task that the activity is in will always be maintained by the system — "true" if it will be, and "false" if the system is allowed to reset the task to its initial state in certain situations. The default value is "false". This attribute is meaningful only for the root activity of a task; it's ignored for all other activities.

Normally, the system clears a task (removes all activities from the stack above the root activity) in certain situations when the user re-selects that task from the home screen. Typically, this is done if the user hasn't visited the task for a certain amount of time, such as 30 minutes.

However, when this attribute is "true", users will always return to the task in its last state, regardless of how they get there. This is useful, for example, in an application like the web browser where there is a lot of state (such as multiple open tabs) that users would not like to lose.

### **android:clearTaskOnLaunch**

Whether or not all activities will be removed from the task, except for the root activity, whenever it is re-launched from the home screen — "true" if the task is always stripped down to its root activity, and "false" if not. The default value is "false". This attribute is meaningful only for activities that start a new task (the root activity); it's ignored for all other activities in the task.

When the value is "true", every time users start the task again, they are brought to its root activity regardless of what they were last doing in the task and regardless of whether they used the *Back* or *Home* button to leave it. When the value is "false", the task may be cleared of activities in some situations (see the [alwaysRetainTaskState](#) attribute), but not always.

Suppose, for example, that someone launches activity P from the home screen, and from there goes to activity Q. The user next presses *Home*, and then returns to activity P. Normally, the user would see activity Q, since that is what they were last doing in P's task. However, if P set this flag to "true", all of the activities on top of it (Q in this case) were removed when the user pressed *Home* and the task went to the background. So the user sees only P when returning to the task.

If this attribute and [allowTaskReparenting](#) are both "true", any activities that can be re-parented are moved to the task they share an affinity with; the remaining activities are then dropped, as described above.

#### **android:configChanges**

Lists configuration changes that the activity will handle itself. When a configuration change occurs at runtime, the activity is shut down and restarted by default, but declaring a configuration with this attribute will prevent the activity from being restarted. Instead, the activity remains running and its [onConfigurationChanged\(\)](#) method is called.

**Note:** Using this attribute should be avoided and used only as a last resort. Please read [Handling Runtime Changes](#) for more information about how to properly handle a restart due to a configuration change.

Any or all of the following strings are valid values for this attribute. Multiple values are separated by '|' — for example, "locale|navigation|orientation".

Value	Description
"mcc"	The IMSI mobile country code (MCC) has changed — a SIM has been detected and updated the MCC.
"mnc"	The IMSI mobile network code (MNC) has changed — a SIM has been detected and updated the MNC.
"locale"	The locale has changed — the user has selected a new language that text should be displayed in.
"touchscreen"	The touchscreen has changed. (This should never normally happen.)
"keyboard"	The keyboard type has changed — for example, the user has plugged in an external keyboard.
"keyboardHidden"	The keyboard accessibility has changed — for example, the user has revealed the hardware keyboard.
"navigation"	The navigation type (trackball/dpad) has changed. (This should never normally happen.)
"screenLayout"	The screen layout has changed — this might be caused by a different display being activated.
"fontScale"	The font scaling factor has changed — the user has selected a new global font size.
"uiMode"	The user interface mode has changed — this can be caused when the user places the device into a desk/car dock or when the night mode changes. See <a href="#">UiModeManager</a> . <i>Added in API level 8</i> .
"orientation"	The screen orientation has changed — the user has rotated the device.

**Note:** If your application targets API level 13 or higher (as declared by the [minSdkVersion](#) and [targetSdkVersion](#) attributes), then you should also declare the " screenSize" configuration, because it also changes when a device switches between portrait and landscape orientations.

"screenSize"

The current available screen size has changed. This represents a change in the currently available size, relative to the current aspect ratio, so will change when the user switches between landscape and portrait. However, if your application targets API level 12 or lower, then your activity always handles this configuration change itself (this configuration change does not restart your activity, even when running on an Android 3.2 or higher device).

*Added in API level 13.*

"smallestScreenSize"

The physical screen size has changed. This represents a change in size regardless of orientation, so will only change when the actual physical screen size has changed such as switching to an external display. A change to this configuration corresponds to a change in the [smallest-Width configuration](#). However, if your application targets API level 12 or lower, then your activity always handles this configuration change itself (this configuration change does not restart your activity, even when running on an Android 3.2 or higher device).

*Added in API level 13.*

"layoutDirection"

The layout direction has changed. For example, changing from left-to-right (LTR) to right-to-left (RTL). *Added in API level 17.*

All of these configuration changes can impact the resource values seen by the application. Therefore, when [onConfigurationChanged\(\)](#) is called, it will generally be necessary to again retrieve all resources (including view layouts, drawables, and so on) to correctly handle the change.

#### **android:enabled**

Whether or not the activity can be instantiated by the system — "true" if it can be, and "false" if not. The default value is "true".

The [`<application>`](#) element has its own [enabled](#) attribute that applies to all application components, including activities. The [`<application>`](#) and [`<activity>`](#) attributes must both be "true" (as they both are by default) for the system to be able to instantiate the activity. If either is "false", it cannot be instantiated.

#### **android:excludeFromRecents**

Whether or not the task initiated by this activity should be excluded from the list of recently used applications ("recent apps"). That is, when this activity is the root activity of a new task, this attribute determines whether the task should not appear in the list of recent apps. Set "true" if the task should be *excluded* from the list; set "false" if it should be *included*. The default value is "false".

#### **android:exported**

Whether or not the activity can be launched by components of other applications — "true" if it can be, and "false" if not. If "false", the activity can be launched only by components of the same application or applications with the same user ID.

The default value depends on whether the activity contains intent filters. The absence of any filters means that the activity can be invoked only by specifying its exact class name. This implies that the activity is intended only for application-internal use (since others would not know the class name). So in this case, the default value is "false". On the other hand, the presence of at least one filter implies that the activity is intended for external use, so the default value is "true".

This attribute is not the only way to limit an activity's exposure to other applications. You can also use a permission to limit the external entities that can invoke the activity (see the [permission](#) attribute).

#### **android:finishOnTaskLaunch**

Whether or not an existing instance of the activity should be shut down (finished) whenever the user again launches its task (chooses the task on the home screen) — "true" if it should be shut down, and "false" if not. The default value is "false".

If this attribute and [allowTaskReparenting](#) are both "true", this attribute trumps the other. The affinity of the activity is ignored. The activity is not re-parented, but destroyed.

#### **android:hardwareAccelerated**

Whether or not hardware-accelerated rendering should be enabled for this Activity — "true" if it should be enabled, and "false" if not. The default value is "false".

Starting from Android 3.0, a hardware-accelerated OpenGL renderer is available to applications, to improve performance for many common 2D graphics operations. When the hardware-accelerated renderer is enabled, most operations in Canvas, Paint, Xfermode, ColorFilter, Shader, and Camera are accelerated. This results in smoother animations, smoother scrolling, and improved responsiveness overall, even for applications that do not explicitly make use of the framework's OpenGL libraries. Because of the increased resources required to enable hardware acceleration, your app will consume more RAM.

Note that not all of the OpenGL 2D operations are accelerated. If you enable the hardware-accelerated renderer, test your application to ensure that it can make use of the renderer without errors.

#### **android:icon**

An icon representing the activity. The icon is displayed to users when a representation of the activity is required on-screen. For example, icons for activities that initiate tasks are displayed in the launcher window. The icon is often accompanied by a label (see the [android:label](#) attribute).

This attribute must be set as a reference to a drawable resource containing the image definition. If it is not set, the icon specified for the application as a whole is used instead (see the [<application>](#) element's [icon](#) attribute).

The activity's icon — whether set here or by the [<application>](#) element — is also the default icon for all the activity's intent filters (see the [<intent-filter>](#) element's [icon](#) attribute).

#### **android:label**

A user-readable label for the activity. The label is displayed on-screen when the activity must be represented to the user. It's often displayed along with the activity icon.

If this attribute is not set, the label set for the application as a whole is used instead (see the [<application>](#) element's [label](#) attribute).

The activity's label — whether set here or by the [<application>](#) element — is also the default label for all the activity's intent filters (see the [<intent-filter>](#) element's [label](#) attribute).

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

## `android:launchMode`

An instruction on how the activity should be launched. There are four modes that work in conjunction with activity flags (`FLAG_ACTIVITY_*` constants) in [Intent](#) objects to determine what should happen when the activity is called upon to handle an intent. They are:

```
"standard"  
"singleTop"  
"singleTask"  
"singleInstance"
```

The default mode is "standard".

As shown in the table below, the modes fall into two main groups, with "standard" and "singleTop" activities on one side, and "singleTask" and "singleInstance" activities on the other. An activity with the "standard" or "singleTop" launch mode can be instantiated multiple times. The instances can belong to any task and can be located anywhere in the activity stack. Typically, they're launched into the task that called [startActivity\(\)](#) (unless the Intent object contains a `FLAG_ACTIVITY_NEW_TASK` instruction, in which case a different task is chosen — see the [taskAffinity](#) attribute).

In contrast, "singleTask" and "singleInstance" activities can only begin a task. They are always at the root of the activity stack. Moreover, the device can hold only one instance of the activity at a time — only one such task.

The "standard" and "singleTop" modes differ from each other in just one respect: Every time there's a new intent for a "standard" activity, a new instance of the class is created to respond to that intent. Each instance handles a single intent. Similarly, a new instance of a "singleTop" activity may also be created to handle a new intent. However, if the target task already has an existing instance of the activity at the top of its stack, that instance will receive the new intent (in an [onNewIntent\(\)](#) call); a new instance is not created. In other circumstances — for example, if an existing instance of the "singleTop" activity is in the target task, but not at the top of the stack, or if it's at the top of a stack, but not in the target task — a new instance would be created and pushed on the stack.

The "singleTask" and "singleInstance" modes also differ from each other in only one respect: A "singleTask" activity allows other activities to be part of its task. It's always at the root of its task, but other activities (necessarily "standard" and "singleTop" activities) can be launched into that task. A "singleInstance" activity, on the other hand, permits no other activities to be part of its task. It's the only activity in the task. If it starts another activity, that activity is assigned to a different task — as if `FLAG_ACTIVITY_NEW_TASK` was in the intent.

Use Cases	Launch Mode	Multiple Instances?	Comments
Normal launches for most activities	"standard"	Yes	Default. The system always creates a new instance of the activity in the target task and routes the intent to it.
	"singleTop"	Conditionally	If an instance of the activity already exists at the top of the target task, the system routes the intent to that instance through a call to its <a href="#">onNewIntent()</a> method, rather than creating a new instance of the activity.
Specialized launches	"singleTask"	No	The system creates the activity at the root of a new task and routes the intent to it. However, if an

*(not recommended for general use)*

"singleInstance" No

instance of the activity already exists, the system routes the intent to existing instance through a call to its [onNewIntent\(\)](#) method, rather than creating a new one.

Same as "singleTask", except that the system doesn't launch any other activities into the task holding the instance. The activity is always the single and only member of its task.

As shown in the table above, standard is the default mode and is appropriate for most types of activities. SingleTop is also a common and useful launch mode for many types of activities. The other modes — singleTask and singleInstance — are not appropriate for most applications, since they result in an interaction model that is likely to be unfamiliar to users and is very different from most other applications.

Regardless of the launch mode that you choose, make sure to test the usability of the activity during launch and when navigating back to it from other activities and tasks using the *Back* button.

For more information on launch modes and their interaction with Intent flags, see the [Tasks and Back Stack](#) document.

#### **android:multiprocess**

Whether an instance of the activity can be launched into the process of the component that started it — "true" if it can be, and "false" if not. The default value is "false".

Normally, a new instance of an activity is launched into the process of the application that defined it, so all instances of the activity run in the same process. However, if this flag is set to "true", instances of the activity can run in multiple processes, allowing the system to create instances wherever they are used (provided permissions allow it), something that is almost never necessary or desirable.

#### **android:name**

The name of the class that implements the activity, a subclass of [Activity](#). The attribute value should be a fully qualified class name (such as, "com.example.project.ExtracurricularActivity"). However, as a shorthand, if the first character of the name is a period (for example, ".ExtracurricularActivity"), it is appended to the package name specified in the [<manifest>](#) element.

Once you publish your application, you [should not change this name](#) (unless you've set [android:exported="false"](#)).

There is no default. The name must be specified.

#### **android:noHistory**

Whether or not the activity should be removed from the activity stack and finished (its [finish\(\)](#) method called) when the user navigates away from it and it's no longer visible on screen — "true" if it should be finished, and "false" if not. The default value is "false".

A value of "true" means that the activity will not leave a historical trace. It will not remain in the activity stack for the task, so the user will not be able to return to it.

This attribute was introduced in API Level 3.

## **android:parentActivityName**

The class name of the logical parent of the activity. The name here must match the class name given to the corresponding `<activity>` element's [android:name](#) attribute.

The system reads this attribute to determine which activity should be started when the user presses the Up button in the action bar. The system can also use this information to synthesize a back stack of activities with [TaskStackBuilder](#).

To support API levels 4 - 16, you can also declare the parent activity with a `<meta-data>` element that specifies a value for "android.support.PARENT\_ACTIVITY". For example:

```
<activity
    android:name="com.example.app.ChildActivity"
    android:label="@string/title_child_activity"
    android:parentActivityName="com.example.myfirstapp.MainActivity" >
    <!-- Parent activity meta-data to support API level 4+ -->
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value="com.example.app.MainActivity" />
</activity>
```

For more information about declaring the parent activity to support Up navigation, read [Providing Up Navigation](#).

This attribute was introduced in API Level 16.

## **android:permission**

The name of a permission that clients must have to launch the activity or otherwise get it to respond to an intent. If a caller of [startActivity\(\)](#) or [startActivityForResult\(\)](#) has not been granted the specified permission, its intent will not be delivered to the activity.

If this attribute is not set, the permission set by the `<application>` element's [permission](#) attribute applies to the activity. If neither attribute is set, the activity is not protected by a permission.

For more information on permissions, see the [Permissions](#) section in the introduction and another document, [Security and Permissions](#).

## **android:process**

The name of the process in which the activity should run. Normally, all components of an application run in a default process name created for the application and you do not need to use this attribute. But if necessary, you can override the default process name with this attribute, allowing you to spread your app components across multiple processes.

If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the activity runs in that process. If the process name begins with a lowercase character, the activity will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage.

The `<application>` element's [process](#) attribute can set a different default process name for all components.

## **android:screenOrientation**

The orientation of the activity's display on the device.

The value can be any one of the following strings:

"unspecified"	The default value. The system chooses the orientation. The policy it uses, and therefore the choices made in specific contexts, may differ from device to device.
"behind"	The same orientation as the activity that's immediately beneath it in the activity stack.
"landscape"	Landscape orientation (the display is wider than it is tall).
"portrait"	Portrait orientation (the display is taller than it is wide).
"reverseLandscape"	Landscape orientation in the opposite direction from normal landscape. <i>Added in API level 9.</i>
"reversePortrait"	Portrait orientation in the opposite direction from normal portrait. <i>Added in API level 9.</i>
"sensorLandscape"	Landscape orientation, but can be either normal or reverse landscape based on the device sensor. <i>Added in API level 9.</i>
"sensorPortrait"	Portrait orientation, but can be either normal or reverse portrait based on the device sensor. <i>Added in API level 9.</i>
"userLandscape"	Landscape orientation, but can be either normal or reverse landscape based on the device sensor and the user's sensor preference. If the user has locked sensor-based rotation, this behaves the same as <code>landscape</code> , otherwise it behaves the same as <code>sensorLandscape</code> . <i>Added in API level 18.</i>
"userPortrait"	Portrait orientation, but can be either normal or reverse portrait based on the device sensor and the user's sensor preference. If the user has locked sensor-based rotation, this behaves the same as <code>portrait</code> , otherwise it behaves the same as <code>sensorPortrait</code> . <i>Added in API level 18.</i>
"sensor"	The orientation is determined by the device orientation sensor. The orientation of the display depends on how the user is holding the device; it changes when the user rotates the device. Some devices, though, will not rotate to all four possible orientations, by default. To allow all four orientations, use " <code>fullSensor</code> ".
"fullSensor"	The orientation is determined by the device orientation sensor for any of the 4 orientations. This is similar to " <code>sensor</code> " except this allows any of the 4 possible screen orientations, regardless of what the device will normally do (for example, some devices won't normally use reverse portrait or reverse landscape, but this enables those). <i>Added in API level 9.</i>
"nosensor"	The orientation is determined without reference to a physical orientation sensor. The sensor is ignored, so the display will not rotate based on how the user moves the device. Except for this distinction, the system chooses the orientation using the same policy as for the " <code>unspecified</code> " setting.
"user"	The user's current preferred orientation.
"fullUser"	If the user has locked sensor-based rotation, this behaves the same as <code>user</code> , otherwise it behaves the same as <code>fullSensor</code> and allows any of the 4 possible screen orientations. <i>Added in API level 18.</i>
"locked"	Locks the orientation to its current rotation, whatever that is. <i>Added in API level 18.</i>

**Note:** When you declare one of the landscape or portrait values, it is considered a hard requirement for the orientation in which the activity runs. As such, the value you declare enables filtering by services such as Google Play so your application is available only to devices that support the orientation required by your activities. For example, if you declare either "`landscape`", "`reverseLandscape`", or "`sensorLandscape`", then your application will be available only to devices that support landscape orientation. However, you should also explicitly declare that your application re-

quires either portrait or landscape orientation with the `<uses-feature>` element. For example, `<uses-feature android:name="android.hardware.screen.portrait"/>`. This is purely a filtering behavior provided by Google Play (and other services that support it) and the platform itself does not control whether your app can be installed when a device supports only certain orientations.

#### **android:stateNotNeeded**

Whether or not the activity can be killed and successfully restarted without having saved its state — "true" if it can be restarted without reference to its previous state, and "false" if its previous state is required. The default value is "false".

Normally, before an activity is temporarily shut down to save resources, its [onSaveInstanceState\(\)](#) method is called. This method stores the current state of the activity in a [Bundle](#) object, which is then passed to [onCreate\(\)](#) when the activity is restarted. If this attribute is set to "true", `onSaveInstanceState()` may not be called and `onCreate()` will be passed `null` instead of the [Bundle](#) — just as it was when the activity started for the first time.

A "true" setting ensures that the activity can be restarted in the absence of retained state. For example, the activity that displays the home screen uses this setting to make sure that it does not get removed if it crashes for some reason.

#### **android:taskAffinity**

The task that the activity has an affinity for. Activities with the same affinity conceptually belong to the same task (to the same "application" from the user's perspective). The affinity of a task is determined by the affinity of its root activity.

The affinity determines two things — the task that the activity is re-parented to (see the [allowTaskReparenting](#) attribute) and the task that will house the activity when it is launched with the [FLAG\\_ACTIVITY\\_NEW\\_TASK](#) flag.

By default, all activities in an application have the same affinity. You can set this attribute to group them differently, and even place activities defined in different applications within the same task. To specify that the activity does not have an affinity for any task, set it to an empty string.

If this attribute is not set, the activity inherits the affinity set for the application (see the [<application>](#) element's [taskAffinity](#) attribute). The name of the default affinity for an application is the package name set by the [<manifest>](#) element.

#### **android:theme**

A reference to a style resource defining an overall theme for the activity. This automatically sets the activity's context to use this theme (see [setTheme\(\)](#), and may also cause "starting" animations prior to the activity being launched (to better match what the activity actually looks like).

If this attribute is not set, the activity inherits the theme set for the application as a whole — from the [<application>](#) element's [theme](#) attribute. If that attribute is also not set, the default system theme is used. For more information, see the [Styles and Themes](#) developer guide.

#### **android:uiOptions**

Extra options for an activity's UI.

Must be one of the following values.

Value	Description
-------	-------------

"none"	No extra UI options. This is the default.
"splitActionBarWhenNarrow"	Add a bar at the bottom of the screen to display action items in the <a href="#">ActionBar</a> , when constrained for horizontal space (such as when in portrait mode on a handset). Instead of a small number of action items appearing in the action bar at the top of the screen, the action bar is split into the top navigation section and the bottom bar for action items. This ensures a reasonable amount of space is made available not only for the action items, but also for navigation and title elements at the top. Menu items are not split across the two bars; they always appear together.

For more information about the action bar, see the [Action Bar](#) developer guide.

This attribute was added in API level 14.

#### **android:windowSoftInputMode**

How the main window of the activity interacts with the window containing the on-screen soft keyboard. The setting for this attribute affects two things:

- The state of the soft keyboard — whether it is hidden or visible — when the activity becomes the focus of user attention.
- The adjustment made to the activity's main window — whether it is resized smaller to make room for the soft keyboard or whether its contents pan to make the current focus visible when part of the window is covered by the soft keyboard.

The setting must be one of the values listed in the following table, or a combination of one "state..." value plus one "adjust..." value. Setting multiple values in either group — multiple "state..." values, for example — has undefined results. Individual values are separated by a vertical bar (|). For example:

```
<activity android:windowSoftInputMode="stateVisible|adjustResize" . . . >
```

Values set here (other than "stateUnspecified" and "adjustUnspecified") override values set in the theme.

<b>Value</b>	<b>Description</b>
"stateUnspecified"	The state of the soft keyboard (whether it is hidden or visible) is not specified. The system will choose an appropriate state or rely on the setting in the theme.
	This is the default setting for the behavior of the soft keyboard.
"stateUnchanged"	The soft keyboard is kept in whatever state it was last in, whether visible or hidden, when the activity comes to the fore.
"stateHidden"	The soft keyboard is hidden when the user chooses the activity — that is, when the user affirmatively navigates forward to the activity, rather than backs into it because of leaving another activity.
"stateAlwaysHidden"	The soft keyboard is always hidden when the activity's main window has input focus.
"stateVisible"	The soft keyboard is visible when that's normally appropriate (when the user is navigating forward to the activity's main window).

The soft keyboard is made visible when the user chooses the activity — "stateAlwaysVisible" — that is, when the user affirmatively navigates forward to the activity, rather than backs into it because of leaving another activity.

It is unspecified whether the activity's main window resizes to make room for the soft keyboard, or whether the contents of the window pan to make the current focus visible on-screen. The system will automatically select one of these modes depending on whether the content of the window has any layout views that can scroll their contents. If there is such a view, the window will be resized, on the assumption that scrolling can make all of the window's contents visible within a smaller area.

This is the default setting for the behavior of the main window.

"adjustUnspecified" The activity's main window is always resized to make room for the soft keyboard on screen.

"adjustResize" The activity's main window is not resized to make room for the soft keyboard. Rather, the contents of the window are automatically panned so that the current focus is never obscured by the keyboard and users can always see what they are typing. This is generally less desirable than resizing, because the user may need to close the soft keyboard to get at and interact with obscured parts of the window.

This attribute was introduced in API Level 3.

#### **introduced in:**

API Level 1 for all attributes except for [noHistory](#) and [windowSoftInputMode](#), which were added in API Level 3.

#### **see also:**

[<application>](#)  
[<activity-alias>](#)

# <service>

**syntax:**

```
<service android:enabled=["true" | "false"]  
        android:exported=["true" | "false"]  
        android:icon="drawable resource"  
        android:isolatedProcess=["true" | "false"]  
        android:label="string resource"  
        android:name="string"  
        android:permission="string"  
        android:process="string" >  
        . . .  
</service>
```

**contained in:**

[<application>](#)

**can contain:**

[<intent-filter>](#)  
[<meta-data>](#)

**description:**

Declares a service (a [Service](#) subclass) as one of the application's components. Unlike activities, services lack a visual user interface. They're used to implement long-running background operations or a rich communications API that can be called by other applications.

All services must be represented by `<service>` elements in the manifest file. Any that are not declared there will not be seen by the system and will never be run.

**attributes:**

**android:enabled**

Whether or not the service can be instantiated by the system — "true" if it can be, and "false" if not. The default value is "true".

The [<application>](#) element has its own `enabled` attribute that applies to all application components, including services. The [<application>](#) and `<service>` attributes must both be "true" (as they both are by default) for the service to be enabled. If either is "false", the service is disabled; it cannot be instantiated.

**android:exported**

Whether or not components of other applications can invoke the service or interact with it — "true" if they can, and "false" if not. When the value is "false", only components of the same application or applications with the same user ID can start the service or bind to it.

The default value depends on whether the service contains intent filters. The absence of any filters means that it can be invoked only by specifying its exact class name. This implies that the service is intended only for application-internal use (since others would not know the class name). So in this case, the default value is "false". On the other hand, the presence of at least one filter implies that the service is intended for external use, so the default value is "true".

This attribute is not the only way to limit the exposure of a service to other applications. You can also use a permission to limit the external entities that can interact with the service (see the [permission](#) attribute).

#### **android:icon**

An icon representing the service. This attribute must be set as a reference to a drawable resource containing the image definition. If it is not set, the icon specified for the application as a whole is used instead (see the [application](#) element's [icon](#) attribute).

The service's icon — whether set here or by the [application](#) element — is also the default icon for all the service's intent filters (see the [intent-filter](#) element's [icon](#) attribute).

#### **android:isolatedProcess**

If set to true, this service will run under a special process that is isolated from the rest of the system and has no permissions of its own. The only communication with it is through the Service API (binding and starting).

#### **android:label**

A name for the service that can be displayed to users. If this attribute is not set, the label set for the application as a whole is used instead (see the [application](#) element's [label](#) attribute).

The service's label — whether set here or by the [application](#) element — is also the default label for all the service's intent filters (see the [intent-filter](#) element's [label](#) attribute).

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

#### **android:name**

The name of the [Service](#) subclass that implements the service. This should be a fully qualified class name (such as, "com.example.project.RoomService"). However, as a shorthand, if the first character of the name is a period (for example, ".RoomService"), it is appended to the package name specified in the [manifest](#) element.

Once you publish your application, you [should not change this name](#) (unless you've set [android:exported="false"](#)).

There is no default. The name must be specified.

#### **android:permission**

The name of a permission that that an entity must have in order to launch the service or bind to it. If a caller of [startService\(\)](#), [bindService\(\)](#), or [stopService\(\)](#), has not been granted this permission, the method will not work and the Intent object will not be delivered to the service.

If this attribute is not set, the permission set by the [application](#) element's [permission](#) attribute applies to the service. If neither attribute is set, the service is not protected by a permission.

For more information on permissions, see the [Permissions](#) section in the introduction and a separate document, [Security and Permissions](#).

#### **android:process**

The name of the process where the service is to run. Normally, all components of an application run in the default process created for the application. It has the same name as the application package. The [application](#) element's [process](#) attribute can set a different default for all components. But

component can override the default with its own `process` attribute, allowing you to spread your application across multiple processes.

If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the service runs in that process. If the process name begins with a lowercase character, the service will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage.

**see also:**

[<application>](#)  
[<activity>](#)

**introduced in:**

API Level 1

# <receiver>

**syntax:**

```
<receiver android:enabled=["true" | "false"]  
         android:exported=["true" | "false"]  
         android:icon="drawable resource"  
         android:label="string resource"  
         android:name="string"  
         android:permission="string"  
         android:process="string" >  
         . . .  
</receiver>
```

**contained in:**

[<application>](#)

**can contain:**

[<intent-filter>](#)  
[<meta-data>](#)

**description:**

Declares a broadcast receiver (a [BroadcastReceiver](#) subclass) as one of the application's components. Broadcast receivers enable applications to receive intents that are broadcast by the system or by other applications, even when other components of the application are not running.

There are two ways to make a broadcast receiver known to the system: One is declare it in the manifest file with this element. The other is to create the receiver dynamically in code and register it with the [Context.registerReceiver\(\)](#) method. See the [BroadcastReceiver](#) class description for more on dynamically created receivers.

**attributes:**

**android:enabled**

Whether or not the broadcast receiver can be instantiated by the system — "true" if it can be, and "false" if not. The default value is "true".

The [<application>](#) element has its own [enabled](#) attribute that applies to all application components, including broadcast receivers. The [<application>](#) and [<receiver>](#) attributes must both be "true" for the broadcast receiver to be enabled. If either is "false", it is disabled; it cannot be instantiated.

**android:exported**

Whether or not the broadcast receiver can receive messages from sources outside its application — "true" if it can, and "false" if not. If "false", the only messages the broadcast receiver can receive are those sent by components of the same application or applications with the same user ID.

The default value depends on whether the broadcast receiver contains intent filters. The absence of any filters means that it can be invoked only by Intent objects that specify its exact class name. This implies that the receiver is intended only for application-internal use (since others would not normally know the class name). So in this case, the default value is "false". On the other hand, the presence of at least one filter implies that the broadcast receiver is intended to receive intents broadcast by the system or other applications, so the default value is "true".

This attribute is not the only way to limit a broadcast receiver's external exposure. You can also use a permission to limit the external entities that can send it messages (see the [permission](#) attribute).

#### **android:icon**

An icon representing the broadcast receiver. This attribute must be set as a reference to a drawable resource containing the image definition. If it is not set, the icon specified for the application as a whole is used instead (see the [application](#) element's [icon](#) attribute).

The broadcast receiver's icon — whether set here or by the [application](#) element — is also the default icon for all the receiver's intent filters (see the [intent-filter](#) element's [icon](#) attribute).

#### **android:label**

A user-readable label for the broadcast receiver. If this attribute is not set, the label set for the application as a whole is used instead (see the [application](#) element's [label](#) attribute).

The broadcast receiver's label — whether set here or by the [application](#) element — is also the default label for all the receiver's intent filters (see the [intent-filter](#) element's [label](#) attribute).

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

#### **android:name**

The name of the class that implements the broadcast receiver, a subclass of [BroadcastReceiver](#). This should be a fully qualified class name (such as, "com.example.project.ReportReceiver"). However, as a shorthand, if the first character of the name is a period (for example, ". ReportReceiver"), it is appended to the package name specified in the [manifest](#) element.

Once you publish your application, you [should not change this name](#) (unless you've set [android:exported="false"](#)).

There is no default. The name must be specified.

#### **android:permission**

The name of a permission that broadcasters must have to send a message to the broadcast receiver. If this attribute is not set, the permission set by the [application](#) element's [permission](#) attribute applies to the broadcast receiver. If neither attribute is set, the receiver is not protected by a permission.

For more information on permissions, see the [Permissions](#) section in the introduction and a separate document, [Security and Permissions](#).

#### **android:process**

The name of the process in which the broadcast receiver should run. Normally, all components of an application run in the default process created for the application. It has the same name as the application package. The [application](#) element's [process](#) attribute can set a different default for all components. But each component can override the default with its own [process](#) attribute, allowing you to spread your application across multiple processes.

If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the broadcast receiver runs in that process. If the process name begins

with a lowercase character, the receiver will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage.

**introduced in:**

API Level 1

# <provider>

syntax:

```
<provider android:authorities="list"
          android:enabled=["true" | "false"]
          android:exported=["true" | "false"]
          android:grantUriPermissions=["true" | "false"]
          android:icon="drawable resource"
          android:initOrder=integer"
          android:label=string resource"
          android:multiprocess=["true" | "false"]
          android:name=string"
          android:permission=string"
          android:process=string"
          android:readPermission=string"
          android:syncable=["true" | "false"]
          android:writePermission=string" >
        .
        .
        .
</provider>
```

contained in:

[application](#)

can contain:

[meta-data](#)  
[grant-uri-permission](#)  
[path-permission](#)

description:

Declares a content provider component. A content provider is a subclass of [ContentProvider](#) that supplies structured access to data managed by the application. All content providers in your application must be defined in a <provider> element in the manifest file; otherwise, the system is unaware of them and doesn't run them.

You only declare content providers that are part of your application. Content providers in other applications that you use in your application should not be declared.

The Android system stores references to content providers according to an **authority** string, part of the provider's **content URI**. For example, suppose you want to access a content provider that stores information about health care professionals. To do this, you call the method [ContentResolver.query\(\)](#), which among other arguments takes a URI that identifies the provider:

```
content://com.example.project.healthcareprovider/nurses/rn
```

The **content : scheme** identifies the URI as a content URI pointing to an Android content provider. The **authority** `com.example.project.healthcareprovider` identifies the provider itself; the Android system looks up the authority in its list of known providers and their authorities. The substring `nurses/rn` is a **path**, which the content provider can use to identify subsets of the provider data.

Notice that when you define your provider in the <provider> element, you don't include the scheme or the path in the `android:name` argument, only the authority.

For information on using and developing content providers, see the API Guide, [Content Providers](#).

## attributes:

### **android:authorities**

A list of one or more URI authorities that identify data offered by the content provider. Multiple authorities are listed by separating their names with a semicolon. To avoid conflicts, authority names should use a Java-style naming convention (such as `com.example.provider.cartoonprovider`). Typically, it's the name of the [Content-Provider](#) subclass that implements the provider

There is no default. At least one authority must be specified.

### **android:enabled**

Whether or not the content provider can be instantiated by the system — "true" if it can be, and "false" if not. The default value is "true".

The [`<application>`](#) element has its own [enabled](#) attribute that applies to all application components, including content providers. The [`<application>`](#) and [`<provider>`](#) attributes must both be "true" (as they both are by default) for the content provider to be enabled. If either is "false", the provider is disabled; it cannot be instantiated.

### **android:exported**

Whether the content provider is available for other applications to use:

- true: The provider is available to other applications. Any application can use the provider's content URI to access it, subject to the permissions specified for the provider.
- false: The provider is not available to other applications. Set `android:exported="false"` to limit access to the provider to your applications. Only applications that have the same user ID (UID) as the provider will have access to it.

The default value is "true" for applications that set either [android:minSdkVersion](#) or [android:targetSdkVersion](#) to "16" or lower. For applications that set either of these attributes to "17" or higher, the default is "false".

You can set `android:exported="false"` and still limit access to your provider by setting permissions with the [permission](#) attribute.

### **android:grantUriPermissions**

Whether or not those who ordinarily would not have permission to access the content provider's data can be granted permission to do so, temporarily overcoming the restriction imposed by the [readPermission](#), [writePermission](#), and [permission](#) attributes — "true" if permission can be granted, and "false" if not. If "true", permission can be granted to any of the content provider's data. If "false", permission can be granted only to the data subsets listed in [`<grant-uri-permission>`](#) subelements, if any. The default value is "false".

Granting permission is a way of giving an application component one-time access to data protected by a permission. For example, when an e-mail message contains an attachment, the mail application may call upon the appropriate viewer to open it, even though the viewer doesn't have general permission to look at all the content provider's data.

In such cases, permission is granted by [FLAG\\_GRANT\\_READ\\_URI\\_PERMISSION](#) and [FLAG\\_GRANT\\_WRITE\\_URI\\_PERMISSION](#) flags in the Intent object that activates the component. For example, the mail application might put `FLAG_GRANT_READ_URI_PERMISSION` in the Intent passed to `Context.startActivity()`. The permission is specific to the URI in the Intent.

If you enable this feature, either by setting this attribute to "true" or by defining [`<grant-uri-permission>`](#) subelements, you must call [`Context.revokeUriPermission\(\)`](#) when a covered URI is deleted from the provider.

See also the [`<grant-uri-permission>`](#) element.

#### **android:icon**

An icon representing the content provider. This attribute must be set as a reference to a drawable resource containing the image definition. If it is not set, the icon specified for the application as a whole is used instead (see the [`<application>`](#) element's [`icon`](#) attribute).

#### **android:initOrder**

The order in which the content provider should be instantiated, relative to other content providers hosted by the same process. When there are dependencies among content providers, setting this attribute for each of them ensures that they are created in the order required by those dependencies. The value is a simple integer, with higher numbers being initialized first.

#### **android:label**

A user-readable label for the content provided. If this attribute is not set, the label set for the application as a whole is used instead (see the [`<application>`](#) element's [`label`](#) attribute).

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

#### **android:multiprocess**

Whether or not an instance of the content provider can be created in every client process — "true" if instances can run in multiple processes, and "false" if not. The default value is "false".

Normally, a content provider is instantiated in the process of the application that defined it. However, if this flag is set to "true", the system can create an instance in every process where there's a client that wants to interact with it, thus avoiding the overhead of interprocess communication.

#### **android:name**

The name of the class that implements the content provider, a subclass of [`ContentProvider`](#). This should be a fully qualified class name (such as, "com.example.project.TransportationProvider"). However, as a shorthand, if the first character of the name is a period, it is appended to the package name specified in the [`<manifest>`](#) element.

There is no default. The name must be specified.

#### **android:permission**

The name of a permission that clients must have to read or write the content provider's data. This attribute is a convenient way of setting a single permission for both reading and writing. However, the [`readPermission`](#) and [`writePermission`](#) attributes take precedence over this one. If the [`readPermission`](#) attribute is also set, it controls access for querying the content provider. And if the [`writePermission`](#) attribute is set, it controls access for modifying the provider's data.

For more information on permissions, see the [`Permissions`](#) section in the introduction and a separate document, [`Security and Permissions`](#).

## **android:process**

The name of the process in which the content provider should run. Normally, all components of an application run in the default process created for the application. It has the same name as the application package. The [`<application>`](#) element's `process` attribute can set a different default for all components. But each component can override the default with its own `process` attribute, allowing you to spread your application across multiple processes.

If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the activity runs in that process. If the process name begins with a lowercase character, the activity will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage.

## **android:readPermission**

A permission that clients must have to query the content provider. See also the [`permission`](#) and [`writePermission`](#) attributes.

## **android:syncable**

Whether or not the data under the content provider's control is to be synchronized with data on a server—"true" if it is to be synchronized, and "false" if not.

## **android:writePermission**

A permission that clients must have to make changes to the data controlled by the content provider. See also the [`permission`](#) and [`readPermission`](#) attributes.

### **introduced in:**

API Level 1

### **see also:**

[Content Providers](#)

# The AndroidManifest.xml File

## In this document

1. [Structure of the Manifest File](#)
2. [File Conventions](#)
3. [File Features](#)
  1. [Intent Filters](#)
  2. [Icons and Labels](#)
  3. [Permissions](#)
  4. [Libraries](#)

Every application must have an `AndroidManifest.xml` file (with precisely that name) in its root directory. The manifest presents essential information about the application to the Android system, information the system must have before it can run any of the application's code. Among other things, the manifest does the following:

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It describes the components of the application — the activities, services, broadcast receivers, and content providers that the application is composed of. It names the classes that implement each of the components and publishes their capabilities (for example, which [Intent](#) messages they can handle). These declarations let the Android system know what the components are and under what conditions they can be launched.
- It determines which processes will host application components.
- It declares which permissions the application must have in order to access protected parts of the API and interact with other applications.
- It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the [Instrumentation](#) classes that provide profiling and other information as the application is running. These declarations are present in the manifest only while the application is being developed and tested; they're removed before the application is published.
- It declares the minimum level of the Android API that the application requires.
- It lists the libraries that the application must be linked against.

## Structure of the Manifest File

The diagram below shows the general structure of the manifest file and every element that it can contain. Each element, along with all of its attributes, is documented in full in a separate file. To view detailed information about any element, click on the element name in the diagram, in the alphabetical list of elements that follows the diagram, or on any other mention of the element name.

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

<manifest>

    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
```

```

<uses-feature />
<supports-screens />
<compatible-screens />
<supports-gl-texture />

<application>

    <activity>
        <intent-filter>
            <action />
            <category />
            <data />
        </intent-filter>
        <meta-data />
    </activity>

    <activity-alias>
        <intent-filter> . . .
        <meta-data />
    </activity-alias>

    <service>
        <intent-filter> . . .
        <meta-data/>
    </service>

    <receiver>
        <intent-filter> . . .
        <meta-data />
    </receiver>

    <provider>
        <grant-uri-permission />
        <meta-data />
        <path-permission />
    </provider>

    <uses-library />

</application>

</manifest>

```

All the elements that can appear in the manifest file are listed below in alphabetical order. These are the only legal elements; you cannot add your own elements or attributes.

```

<action>
<activity>
<activity-alias>
<application>
<category>
<data>
<grant-uri-permission>
<instrumentation>

```

```
<intent-filter>
<manifest>
<meta-data>
<permission>
<permission-group>
<permission-tree>
<provider>
<receiver>
<service>
<supports-screens>
<uses-configuration>
<uses-feature>
<uses-library>
<uses-permission>
<uses-sdk>
```

## File Conventions

Some conventions and rules apply generally to all elements and attributes in the manifest:

### Elements

Only the [<manifest>](#) and [<application>](#) elements are required, they each must be present and can occur only once. Most of the others can occur many times or not at all — although at least some of them must be present for the manifest to accomplish anything meaningful.

If an element contains anything at all, it contains other elements. All values are set through attributes, not as character data within an element.

Elements at the same level are generally not ordered. For example, [<activity>](#), [<provider>](#), and [<service>](#) elements can be intermixed in any sequence. (An [<activity-alias>](#) element is the exception to this rule: It must follow the [<activity>](#) it is an alias for.)

### Attributes

In a formal sense, all attributes are optional. However, there are some that must be specified for an element to accomplish its purpose. Use the documentation as a guide. For truly optional attributes, it mentions a default value or states what happens in the absence of a specification.

Except for some attributes of the root [<manifest>](#) element, all attribute names begin with an android: prefix — for example, android:alwaysRetainTaskState. Because the prefix is universal, the documentation generally omits it when referring to attributes by name.

### Declaring class names

Many elements correspond to Java objects, including elements for the application itself (the [<application>](#) element) and its principal components — activities ([<activity>](#)), services ([<service>](#)), broadcast receivers ([<receiver>](#)), and content providers ([<provider>](#)).

If you define a subclass, as you almost always would for the component classes ([Activity](#), [Service](#), [BroadcastReceiver](#), and [ContentProvider](#)), the subclass is declared through a name attribute. The name must include the full package designation. For example, an [Service](#) subclass might be declared as follows:

```
<manifest . . . >
    <application . . . >
        <service android:name="com.example.project.SecretService" . . . >
            . . .
        </service>
        . . .
    </application>
</manifest>
```

However, as a shorthand, if the first character of the string is a period, the string is appended to the application's package name (as specified by the [`<manifest>`](#) element's [`package`](#) attribute). The following assignment is the same as the one above:

```
<manifest package="com.example.project" . . . >
    <application . . . >
        <service android:name=".SecretService" . . . >
            . . .
        </service>
        . . .
    </application>
</manifest>
```

When starting a component, Android creates an instance of the named subclass. If a subclass isn't specified, it creates an instance of the base class.

## Multiple values

If more than one value can be specified, the element is almost always repeated, rather than listing multiple values within a single element. For example, an intent filter can list several actions:

```
<intent-filter . . . >
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.INSERT" />
    <action android:name="android.intent.action.DELETE" />
    . . .
</intent-filter>
```

## Resource values

Some attributes have values that can be displayed to users — for example, a label and an icon for an activity. The values of these attributes should be localized and therefore set from a resource or theme. Resource values are expressed in the following format,

`@ [package:] type:name`

where the *package* name can be omitted if the resource is in the same package as the application, *type* is a type of resource — such as "string" or "drawable" — and *name* is the name that identifies the specific resource. For example:

```
<activity android:icon="@drawable/smallPic" . . . >
```

Values from a theme are expressed in a similar manner, but with an initial '?' rather than '@':

`? [package:] type:name`

## String values

Where an attribute value is a string, double backslashes ('\\') must be used to escape characters — for example, '\\n' for a newline or '\\xxxxx' for a Unicode character.

# File Features

The following sections describe how some Android features are reflected in the manifest file.

## Intent Filters

The core components of an application (its activities, services, and broadcast receivers) are activated by *intents*. An intent is a bundle of information (an [Intent](#) object) describing a desired action — including the data to be acted upon, the category of component that should perform the action, and other pertinent instructions. Android locates an appropriate component to respond to the intent, launches a new instance of the component if one is needed, and passes it the Intent object.

Components advertise their capabilities — the kinds of intents they can respond to — through *intent filters*. Since the Android system must learn which intents a component can handle before it launches the component, intent filters are specified in the manifest as [`<intent-filter>`](#) elements. A component may have any number of filters, each one describing a different capability.

An intent that explicitly names a target component will activate that component; the filter doesn't play a role. But an intent that doesn't specify a target by name can activate a component only if it can pass through one of the component's filters.

For information on how Intent objects are tested against intent filters, see a separate document, [Intents and Intent Filters](#).

## Icons and Labels

A number of elements have `icon` and `label` attributes for a small icon and a text label that can be displayed to users. Some also have a `description` attribute for longer explanatory text that can also be shown on-screen. For example, the [`<permission>`](#) element has all three of these attributes, so that when the user is asked whether to grant the permission to an application that has requested it, an icon representing the permission, the name of the permission, and a description of what it entails can all be presented to the user.

In every case, the icon and label set in a containing element become the default `icon` and `label` settings for all of the container's subelements. Thus, the icon and label set in the [`<application>`](#) element are the default icon and label for each of the application's components. Similarly, the icon and label set for a component — for example, an [`<activity>`](#) element — are the default settings for each of the component's [`<intent-filter>`](#) elements. If an [`<application>`](#) element sets a label, but an activity and its intent filter do not, the application label is treated as the label for both the activity and the intent filter.

The icon and label set for an intent filter are used to represent a component whenever the component is presented to the user as fulfilling the function advertised by the filter. For example, a filter with "android.intent.action.MAIN" and "android.intent.category.LAUNCHER" settings advertises an activity as one that initiates an application — that is, as one that should be displayed in the application launcher. The icon and label set in the filter are therefore the ones displayed in the launcher.

## Permissions

A *permission* is a restriction limiting access to a part of the code or to data on the device. The limitation is imposed to protect critical data and code that could be misused to distort or damage the user experience.

Each permission is identified by a unique label. Often the label indicates the action that's restricted. For example, here are some permissions defined by Android:

```
android.permission.CALL_EMERGENCY_NUMBERS  
android.permission.READ_OWNER_DATA  
android.permission.SET_WALLPAPER  
android.permission.DEVICE_POWER
```

A feature can be protected by at most one permission.

If an application needs access to a feature protected by a permission, it must declare that it requires that permission with a [`<uses-permission>`](#) element in the manifest. Then, when the application is installed on the device, the installer determines whether or not to grant the requested permission by checking the authorities that signed the application's certificates and, in some cases, asking the user. If the permission is granted, the application is able to use the protected features. If not, its attempts to access those features will simply fail without any notification to the user.

An application can also protect its own components (activities, services, broadcast receivers, and content providers) with permissions. It can employ any of the permissions defined by Android (listed in [`an-  
droid.Manifest.permission`](#)) or declared by other applications. Or it can define its own. A new permission is declared with the [`<permission>`](#) element. For example, an activity could be protected as follows:

```
<manifest . . . >  
    <permission android:name="com.example.project.DEBIT_ACCT" . . . />  
    <uses-permission android:name="com.example.project.DEBIT_ACCT" />  
    . . .  
    <application . . . >  
        <activity android:name="com.example.project.FreneticActivity"  
            android:permission="com.example.project.DEBIT_ACCT"  
            . . . >  
            . . .  
        </activity>  
    </application>  
</manifest>
```

Note that, in this example, the DEBIT\_ACCT permission is not only declared with the [`<permission>`](#) element, its use is also requested with the [`<uses-permission>`](#) element. Its use must be requested in order for other components of the application to launch the protected activity, even though the protection is imposed by the application itself.

If, in the same example, the `permission` attribute was set to a permission declared elsewhere (such as `an-  
droid.permission.CALL_EMERGENCY_NUMBERS`, it would not have been necessary to declare it again with a [`<permission>`](#) element. However, it would still have been necessary to request its use with [`<uses-  
permission>`](#).

The [`<permission-tree>`](#) element declares a namespace for a group of permissions that will be defined in code. And [`<permission-group>`](#) defines a label for a set of permissions (both those declared in the manifest with [`<permission>`](#) elements and those declared elsewhere). It affects only how the permissions are grouped when presented to the user. The [`<permission-group>`](#) element does not specify which permissions belong to the group; it just gives the group a name. A permission is placed in the group by assigning the group name to the [`<permission>`](#) element's `permissionGroup` attribute.

## Libraries

Every application is linked against the default Android library, which includes the basic packages for building applications (with common classes such as Activity, Service, Intent, View, Button, Application, ContentProvider, and so on).

However, some packages reside in their own libraries. If your application uses code from any of these packages, it must explicitly ask to be linked against them. The manifest must contain a separate [`<uses-library>`](#) element to name each of the libraries. (The library name can be found in the documentation for the package.)

# <intent-filter>

## syntax:

```
<intent-filter android:icon="drawable resource"  
            android:label="string resource"  
            android:priority="integer" >  
    . . .  
</intent-filter>
```

## contained in:

[<activity>](#)  
[<activity-alias>](#)  
[<service>](#)  
[<receiver>](#)

## must contain:

[<action>](#)

## can contain:

[<category>](#)  
[<data>](#)

## description:

Specifies the types of intents that an activity, service, or broadcast receiver can respond to. An intent filter declares the capabilities of its parent component — what an activity or service can do and what types of broadcasts a receiver can handle. It opens the component to receiving intents of the advertised type, while filtering out those that are not meaningful for the component.

Most of the contents of the filter are described by its [<action>](#), [<category>](#), and [<data>](#) subelements.

For a more detailed discussion of filters, see the separate [Intents and Intent Filters](#) document, as well as the [Intents Filters](#) section in the introduction.

## attributes:

### **android:icon**

An icon that represents the parent activity, service, or broadcast receiver when that component is presented to the user as having the capability described by the filter.

This attribute must be set as a reference to a drawable resource containing the image definition. The default value is the icon set by the parent component's `icon` attribute. If the parent does not specify an icon, the default is the icon set by the [<application>](#) element.

For more on intent filter icons, see [Icons and Labels](#) in the introduction.

### **android:label**

A user-readable label for the parent component. This label, rather than the one set by the parent component, is used when the component is presented to the user as having the capability described by the filter.

The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface. However, as a convenience while you're developing the application, it can also be set as a raw string.

The default value is the label set by the parent component. If the parent does not specify a label, the default is the label set by the [`<application>`](#) element's [`label`](#) attribute.

For more on intent filter labels, see [Icons and Labels](#) in the introduction.

#### **android:priority**

The priority that should be given to the parent component with regard to handling intents of the type described by the filter. This attribute has meaning for both activities and broadcast receivers:

- It provides information about how able an activity is to respond to an intent that matches the filter, relative to other activities that could also respond to the intent. When an intent could be handled by multiple activities with different priorities, Android will consider only those with higher priority values as potential targets for the intent.
- It controls the order in which broadcast receivers are executed to receive broadcast messages. Those with higher priority values are called before those with lower values. (The order applies only to synchronous messages; it's ignored for asynchronous messages.)

Use this attribute only if you really need to impose a specific order in which the broadcasts are received, or want to force Android to prefer one activity over others.

The value must be an integer, such as "100". Higher numbers have a higher priority. The default value is 0. The value must be greater than -1000 and less than 1000.

Also see [setPriority\(\)](#).

#### **introduced in:**

API Level 1

#### **see also:**

[`<action>`](#)  
[`<category>`](#)  
[`<data>`](#)

# Intents and Intent Filters

## In this document

1. [Intent Objects](#)
2. [Intent Resolution](#)
3. [Intent filters](#)
4. [Common cases](#)
5. [Using intent matching](#)
6. [Note Pad Example](#)

## Key classes

1. [Intent](#)
2. [IntentFilter](#)
3. [BroadcastReceiver](#)
4. [PackageManager](#)

Three of the core components of an application — activities, services, and broadcast receivers — are activated through messages, called *intents*. Intent messaging is a facility for late run-time binding between components in the same or different applications. The intent itself, an [Intent](#) object, is a passive data structure holding an abstract description of an operation to be performed — or, often in the case of broadcasts, a description of something that has happened and is being announced. There are separate mechanisms for delivering intents to each type of component:

- An Intent object is passed to [Context.startActivity\(\)](#) or [Activity.startActivityForResult\(\)](#) to launch an activity or get an existing activity to do something new. (It can also be passed to [Activity.setResult\(\)](#) to return information to the activity that called `startActivityForResult()`.)
- An Intent object is passed to [Context.startService\(\)](#) to initiate a service or deliver new instructions to an ongoing service. Similarly, an intent can be passed to [Context.bindService\(\)](#) to establish a connection between the calling component and a target service. It can optionally initiate the service if it's not already running.
- Intent objects passed to any of the broadcast methods (such as [Context.sendBroadcast\(\)](#), [Context.sendOrderedBroadcast\(\)](#), or [Context.sendStickyBroadcast\(\)](#)) are delivered to all interested broadcast receivers. Many kinds of broadcasts originate in system code.

In each case, the Android system finds the appropriate activity, service, or set of broadcast receivers to respond to the intent, instantiating them if necessary. There is no overlap within these messaging systems: Broadcast intents are delivered only to broadcast receivers, never to activities or services. An intent passed to `startActivity()` is delivered only to an activity, never to a service or broadcast receiver, and so on.

This document begins with a description of Intent objects. It then describes the rules Android uses to map intents to components — how it resolves which component should receive an intent message. For intents that don't explicitly name a target component, this process involves testing the Intent object against *intent filters* associated with potential targets.

# Intent Objects

An [Intent](#) object is a bundle of information. It contains information of interest to the component that receives the intent (such as the action to be taken and the data to act on) plus information of interest to the Android system (such as the category of component that should handle the intent and instructions on how to launch a target activity). Principally, it can contain the following:

## Component name

The name of the component that should handle the intent. This field is a [ComponentName](#) object — a combination of the fully qualified class name of the target component (for example "com.example.project.app.FreneticActivity") and the package name set in the manifest file of the application where the component resides (for example, "com.example.project"). The package part of the component name and the package name set in the manifest do not necessarily have to match.

The component name is optional. If it is set, the Intent object is delivered to an instance of the designated class. If it is not set, Android uses other information in the Intent object to locate a suitable target — see [Intent Resolution](#), later in this document.

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

## Action

A string naming the action to be performed — or, in the case of broadcast intents, the action that took place and is being reported. The Intent class defines a number of action constants, including these:

Constant	Target component	Action
ACTION_CALL	activity	Initiate a phone call.
ACTION_EDIT	activity	Display data for the user to edit.
ACTION_MAIN	activity	Start up as the initial activity of a task, with no data input and no returned output.
ACTION_SYNC	activity	Synchronize data on a server with data on the mobile device.
ACTION_BATTERY_LOW	broadcast receiver	A warning that the battery is low.
ACTION_HEADSET_PLUG	broadcast receiver	A headset has been plugged into the device, or unplugged from it.
ACTION_SCREEN_ON	broadcast receiver	The screen has been turned on.
ACTION_TIMEZONE_CHANGED	broadcast receiver	The setting for the time zone has changed.

See the [Intent](#) class description for a list of pre-defined constants for generic actions. Other actions are defined elsewhere in the Android API. You can also define your own action strings for activating the components in your application. Those you invent should include the application package as a prefix — for example: "com.example.project.SHOW\_COLOR".

The action largely determines how the rest of the intent is structured — particularly the [data](#) and [extras](#) fields — much as a method name determines a set of arguments and a return value. For this reason, it's a good idea to use action names that are as specific as possible, and to couple them tightly to the other fields

of the intent. In other words, instead of defining an action in isolation, define an entire protocol for the Intent objects your components can handle.

The action in an Intent object is set by the [setAction\(\)](#) method and read by [getAction\(\)](#).

## Data

The URI of the data to be acted on and the MIME type of that data. Different actions are paired with different kinds of data specifications. For example, if the action field is ACTION\_EDIT, the data field would contain the URI of the document to be displayed for editing. If the action is ACTION\_CALL, the data field would be a tel: URI with the number to call. Similarly, if the action is ACTION\_VIEW and the data field is an http: URI, the receiving activity would be called upon to download and display whatever data the URI refers to.

When matching an intent to a component that is capable of handling the data, it's often important to know the type of data (its MIME type) in addition to its URI. For example, a component able to display image data should not be called upon to play an audio file.

In many cases, the data type can be inferred from the URI — particularly content: URIs, which indicate that the data is located on the device and controlled by a content provider (see the [separate discussion on content providers](#)). But the type can also be explicitly set in the Intent object. The [setData\(\)](#) method specifies data only as a URI, [setType\(\)](#) specifies it only as a MIME type, and [setDataAndType\(\)](#) specifies it as both a URI and a MIME type. The URI is read by [getData\(\)](#) and the type by [getType\(\)](#).

## Category

A string containing additional information about the kind of component that should handle the intent. Any number of category descriptions can be placed in an Intent object. As it does for actions, the Intent class defines several category constants, including these:

Constant	Meaning
CATEGORY_BROWSABLE	The target activity can be safely invoked by the browser to display data referenced by a link — for example, an image or an e-mail message.
CATEGORY_GADGET	The activity can be embedded inside of another activity that hosts gadgets.
CATEGORY_HOME	The activity displays the home screen, the first screen the user sees when the device is turned on or when the <i>Home</i> button is pressed.
CATEGORY_LAUNCHER	The activity can be the initial activity of a task and is listed in the top-level application launcher.
CATEGORY_PREFERENCE	The target activity is a preference panel.

See the [Intent](#) class description for the full list of categories.

The [addCategory\(\)](#) method places a category in an Intent object, [removeCategory\(\)](#) deletes a category previously added, and [getCategories\(\)](#) gets the set of all categories currently in the object.

## Extras

Key-value pairs for additional information that should be delivered to the component handling the intent. Just as some actions are paired with particular kinds of data URIs, some are paired with particular extras. For example, an ACTION\_TIMEZONE\_CHANGED intent has a "time-zone" extra that identifies the new time zone, and ACTION\_HEADSET\_PLUG has a "state" extra indicating whether the headset is now plugged in or unplugged, as well as a "name" extra for the type of headset. If you were to invent a SHOW\_COLOR action, the color value would be set in an extra key-value pair.

The Intent object has a series of `put...()` methods for inserting various types of extra data and a similar set of `get...()` methods for reading the data. These methods parallel those for [Bundle](#) objects. In fact, the extras can be installed and read as a Bundle using the [putExtras\(\)](#) and [getExtras\(\)](#) methods.

## Flags

Flags of various sorts. Many instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's launched (for example, whether it belongs in the list of recent activities). All these flags are defined in the Intent class.

The Android system and the applications that come with the platform employ Intent objects both to send out system-originated broadcasts and to activate system-defined components. To see how to structure an intent to activate a system component, consult the [list of intents](#) in the reference.

## Intent Resolution

Intents can be divided into two groups:

- *Explicit intents* designate the target component by its name (the [component name field](#), mentioned earlier, has a value set). Since component names would generally not be known to developers of other applications, explicit intents are typically used for application-internal messages — such as an activity starting a subordinate service or launching a sister activity.
- *Implicit intents* do not name a target (the field for the component name is blank). Implicit intents are often used to activate components in other applications.

Android delivers an explicit intent to an instance of the designated target class. Nothing in the Intent object other than the component name matters for determining which component should get the intent.

A different strategy is needed for implicit intents. In the absence of a designated target, the Android system must find the best component (or components) to handle the intent — a single activity or service to perform the requested action or the set of broadcast receivers to respond to the broadcast announcement. It does so by comparing the contents of the Intent object to *intent filters*, structures associated with components that can potentially receive intents. Filters advertise the capabilities of a component and delimit the intents it can handle. They open the component to the possibility of receiving implicit intents of the advertised type. If a component does not have any intent filters, it can receive only explicit intents. A component with filters can receive both explicit and implicit intents.

Only three aspects of an Intent object are consulted when the object is tested against an intent filter:

action  
data (both URI and data type)  
category

The extras and flags play no part in resolving which component receives an intent.

## Intent filters

To inform the system which implicit intents they can handle, activities, services, and broadcast receivers can have one or more intent filters. Each filter describes a capability of the component, a set of intents that the component is willing to receive. It, in effect, filters in intents of a desired type, while filtering out unwanted intents — but only unwanted implicit intents (those that don't name a target class). An explicit intent is always delivered to its target, no matter what it contains; the filter is not consulted. But an implicit intent is delivered to a component only if it can pass through one of the component's filters.

A component has separate filters for each job it can do, each face it can present to the user. For example, the NoteEditor activity of the sample Note Pad application has two filters — one for starting up with a specific note that the user can view or edit, and another for starting with a new, blank note that the user can fill in and save. (All of Note Pad's filters are described in the [Note Pad Example](#) section, later.)

## Filters and security

An intent filter cannot be relied on for security. While it opens a component to receiving only certain kinds of implicit intents, it does nothing to prevent explicit intents from targeting the component. Even though a filter restricts the intents a component will be asked to handle to certain actions and data sources, someone could always put together an explicit intent with a different action and data source, and name the component as the target.

An intent filter is an instance of the [IntentFilter](#) class. However, since the Android system must know about the capabilities of a component before it can launch that component, intent filters are generally not set up in Java code, but in the application's manifest file (AndroidManifest.xml) as [`<intent-filter>`](#) elements. (The one exception would be filters for broadcast receivers that are registered dynamically by calling [`Context.registerReceiver\(\)`](#); they are directly created as IntentFilter objects.)

A filter has fields that parallel the action, data, and category fields of an Intent object. An implicit intent is tested against the filter in all three areas. To be delivered to the component that owns the filter, it must pass all three tests. If it fails even one of them, the Android system won't deliver it to the component — at least not on the basis of that filter. However, since a component can have multiple intent filters, an intent that does not pass through one of a component's filters might make it through on another.

Each of the three tests is described in detail below:

### Action test

An [`<intent-filter>`](#) element in the manifest file lists actions as [`<action>`](#) subelements. For example:

```
<intent-filter . . . >
    <action android:name="com.example.project.SHOW_CURRENT" />
    <action android:name="com.example.project.SHOW_RECENT" />
    <action android:name="com.example.project.SHOW_PENDING" />
    . . .
</intent-filter>
```

As the example shows, while an Intent object names just a single action, a filter may list more than one. The list cannot be empty; a filter must contain at least one [`<action>`](#) element, or it will block all intents.

To pass this test, the action specified in the Intent object must match one of the actions listed in the filter. If the object or the filter does not specify an action, the results are as follows:

- If the filter fails to list any actions, there is nothing for an intent to match, so all intents fail the test. No intents can get through the filter.
- On the other hand, an Intent object that doesn't specify an action automatically passes the test — as long as the filter contains at least one action.

### Category test

An [`<intent-filter>`](#) element also lists categories as subelements. For example:

```
<intent-filter . . . >
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    . . .
</intent-filter>
```

Note that the constants described earlier for actions and categories are not used in the manifest file. The full string values are used instead. For instance, the "android.intent.category.BROWSABLE" string in the example above corresponds to the `CATEGORY_BROWSABLE` constant mentioned earlier in this document. Similarly, the string "android.intent.action.EDIT" corresponds to the `ACTION_EDIT` constant.

For an intent to pass the category test, every category in the Intent object must match a category in the filter. The filter can list additional categories, but it cannot omit any that are in the intent.

In principle, therefore, an Intent object with no categories should always pass this test, regardless of what's in the filter. That's mostly true. However, with one exception, Android treats all implicit intents passed to [startActivity\(\)](#) as if they contained at least one category: "android.intent.category.DEFAULT" (the `CATEGORY_DEFAULT` constant). Therefore, activities that are willing to receive implicit intents must include "android.intent.category.DEFAULT" in their intent filters. (Filters with "android.intent.action.MAIN" and "android.intent.category.LAUNCHER" settings are the exception. They mark activities that begin new tasks and that are represented on the launcher screen. They can include "android.intent.category.DEFAULT" in the list of categories, but don't need to.) See [Using intent matching](#), later, for more on these filters.)

## Data test

Like the action and categories, the data specification for an intent filter is contained in a subelement. And, as in those cases, the subelement can appear multiple times, or not at all. For example:

```
<intent-filter . . . >
    <data android:mimeType="video/mpeg" android:scheme="http" . . . />
    <data android:mimeType="audio/mpeg" android:scheme="http" . . . />
    . . .
</intent-filter>
```

Each [`<data>`](#) element can specify a URI and a data type (MIME media type). There are separate attributes — `scheme`, `host`, `port`, and `path` — for each part of the URI:

`scheme://host:port/path`

For example, in the following URI,

`content://com.example.project:200/folder/subfolder/etc`

the scheme is "content", the host is "com.example.project", the port is "200", and the path is "folder/subfolder/etc". The host and port together constitute the URI *authority*; if a host is not specified, the port is ignored.

Each of these attributes is optional, but they are not independent of each other: For an authority to be meaningful, a scheme must also be specified. For a path to be meaningful, both a scheme and an authority must be specified.

When the URI in an Intent object is compared to a URI specification in a filter, it's compared only to the parts of the URI actually mentioned in the filter. For example, if a filter specifies only a scheme, all URIs with that scheme match the filter. If a filter specifies a scheme and an authority but no path, all URIs with the same scheme and authority match, regardless of their paths. If a filter specifies a scheme, an authority, and a path, only URIs with the same scheme, authority, and path match. However, a path specification in the filter can contain wildcards to require only a partial match of the path.

The `type` attribute of a `<data>` element specifies the MIME type of the data. It's more common in filters than a URI. Both the Intent object and the filter can use a "\*" wildcard for the subtype field — for example, "text/\*" or "audio/\*" — indicating any subtype matches.

The data test compares both the URI and the data type in the Intent object to a URI and data type specified in the filter. The rules are as follows:

- a. An Intent object that contains neither a URI nor a data type passes the test only if the filter likewise does not specify any URIs or data types.
- b. An Intent object that contains a URI but no data type (and a type cannot be inferred from the URI) passes the test only if its URI matches a URI in the filter and the filter likewise does not specify a type. This will be the case only for URIs like `mailto:` and `tel:` that do not refer to actual data.
- c. An Intent object that contains a data type but not a URI passes the test only if the filter lists the same data type and similarly does not specify a URI.
- d. An Intent object that contains both a URI and a data type (or a data type can be inferred from the URI) passes the data type part of the test only if its type matches a type listed in the filter. It passes the URI part of the test either if its URI matches a URI in the filter or if it has a `content:` or `file:` URI and the filter does not specify a URI. In other words, a component is presumed to support `content:` and `file:` data if its filter lists only a data type.

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

## Common cases

The last rule shown above for the data test, rule (d), reflects the expectation that components are able to get local data from a file or content provider. Therefore, their filters can list just a data type and do not need to explicitly name the `content:` and `file:` schemes. This is a typical case. A `<data>` element like the following, for example, tells Android that the component can get image data from a content provider and display it:

```
<data android:mimeType="image/*" />
```

Since most available data is dispensed by content providers, filters that specify a data type but not a URI are perhaps the most common.

Another common configuration is filters with a scheme and a data type. For example, a `<data>` element like the following tells Android that the component can get video data from the network and display it:

```
<data android:scheme="http" android:type="video/*" />
```

Consider, for example, what the browser application does when the user follows a link on a web page. It first tries to display the data (as it could if the link was to an HTML page). If it can't display the data, it puts together an implicit intent with the scheme and data type and tries to start an activity that can do the job. If there are no

takers, it asks the download manager to download the data. That puts it under the control of a content provider, so a potentially larger pool of activities (those with filters that just name a data type) can respond.

Most applications also have a way to start fresh, without a reference to any particular data. Activities that can initiate applications have filters with "android.intent.action.MAIN" specified as the action. If they are to be represented in the application launcher, they also specify the "android.intent.category.LAUNCHER" category:

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

## Using intent matching

Intents are matched against intent filters not only to discover a target component to activate, but also to discover something about the set of components on the device. For example, the Android system populates the application launcher, the top-level screen that shows the applications that are available for the user to launch, by finding all the activities with intent filters that specify the "android.intent.action.MAIN" action and "android.intent.category.LAUNCHER" category (as illustrated in the previous section). It then displays the icons and labels of those activities in the launcher. Similarly, it discovers the home screen by looking for the activity with "android.intent.category.HOME" in its filter.

Your application can use intent matching in a similar way. The [PackageManager](#) has a set of `query...` methods that return all components that can accept a particular intent, and a similar series of `resolve...` methods that determine the best component to respond to an intent. For example, [queryIntentActivities\(\)](#) returns a list of all activities that can perform the intent passed as an argument, and [queryIntentServices\(\)](#) returns a similar list of services. Neither method activates the components; they just list the ones that can respond. There's a similar method, [queryBroadcastReceivers\(\)](#), for broadcast receivers.

## Note Pad Example

The Note Pad sample application enables users to browse through a list of notes, view details about individual items in the list, edit the items, and add a new item to the list. This section looks at the intent filters declared in its manifest file. (If you're working offline in the SDK, you can find all the source files for this sample application, including its manifest file, at `<sdk>/samples/NotePad/index.html`. If you're viewing the documentation online, the source files are in the [Tutorials and Sample Code](#) section [here](#).)

In its manifest file, the Note Pad application declares three activities, each with at least one intent filter. It also declares a content provider that manages the note data. Here is the manifest file in its entirety:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.notepad">
    <application android:icon="@drawable/app_notes"
        android:label="@string/app_name" >

        <provider android:name="NotePadProvider"
            android:authorities="com.google.provider.NotePad" />

        <activity android:name="NotesList" android:label="@string/title_notes_1"
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="android.intent.action.PICK" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.GET_CONTENT" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
</activity>

<activity android:name="NoteEditor"
          android:theme="@android:style/Theme.Light"
          android:label="@string/title_note" >
    <intent-filter android:label="@string/resolve_edit">
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="com.android.notepad.action.EDIT_NOTE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.INSERT" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
    </intent-filter>
</activity>

<activity android:name="TitleEditor"
          android:label="@string/title_edit_title"
          android:theme="@android:style/Theme.Dialog">
    <intent-filter android:label="@string/resolve_title">
        <action android:name="com.android.notepad.action.EDIT_TITLE" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.ALTERNATIVE" />
        <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
        <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
    </intent-filter>
</activity>

</application>
</manifest>

```

The first activity, NotesList, is distinguished from the other activities by the fact that it operates on a directory of notes (the note list) rather than on a single note. It would generally serve as the initial user interface into the application. It can do three things as described by its three intent filters:

1. <intent-filter>
 

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

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

This filter declares the main entry point into the Note Pad application. The standard `MAIN` action is an entry point that does not require any other information in the Intent (no data specification, for example), and the `LAUNCHER` category says that this entry point should be listed in the application launcher.

2. 

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.PICK" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

This filter declares the things that the activity can do on a directory of notes. It can allow the user to view or edit the directory (via the `VIEW` and `EDIT` actions), or to pick a particular note from the directory (via the `PICK` action).

The `mimeType` attribute of the [`<data>`](#) element specifies the kind of data that these actions operate on. It indicates that the activity can get a Cursor over zero or more items (`vnd.android.cursor.dir`) from a content provider that holds Note Pad data (`vnd.google.note`). The Intent object that launches the activity would include a `content:` URI specifying the exact data of this type that the activity should open.

Note also the `DEFAULT` category supplied in this filter. It's there because the [`Context.startActivity\(\)`](#) and [`Activity.startActivityForResult\(\)`](#) methods treat all intents as if they contained the `DEFAULT` category — with just two exceptions:

- Intents that explicitly name the target activity
- Intents consisting of the `MAIN` action and `LAUNCHER` category

Therefore, the `DEFAULT` category is *required* for all filters — except for those with the `MAIN` action and `LAUNCHER` category. (Intent filters are not consulted for explicit intents.)

3. 

```
<intent-filter>
    <action android:name="android.intent.action.GET_CONTENT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

This filter describes the activity's ability to return a note selected by the user without requiring any specification of the directory the user should choose from. The `GET_CONTENT` action is similar to the `PICK` action. In both cases, the activity returns the URI for a note selected by the user. (In each case, it's returned to the activity that called [`startActivityForResult\(\)`](#) to start the `NoteList` activity.) Here, however, the caller specifies the type of data desired instead of the directory of data the user will be picking from.

The data type, `vnd.android.cursor.item/vnd.google.note`, indicates the type of data the activity can return — a URI for a single note. From the returned URI, the caller can get a Cursor for exactly one item (`vnd.android.cursor.item`) from the content provider that holds Note Pad data (`vnd.google.note`).

In other words, for the `PICK` action in the previous filter, the data type indicates the type of data the activity could display to the user. For the `GET_CONTENT` filter, it indicates the type of data the activity can return to the caller.

Given these capabilities, the following intents will resolve to the `NotesList` activity:

**action: `android.intent.action.MAIN`**

Launches the activity with no data specified.

**action: `android.intent.action.MAIN`**

**category: `android.intent.category.LAUNCHER`**

Launches the activity with no data selected specified. This is the actual intent used by the Launcher to populate its top-level list. All activities with filters that match this action and category are added to the list.

**action: `android.intent.action.VIEW`**

**data: `content://com.google.provider.NotePad/notes`**

Asks the activity to display a list of all the notes under `content://com.google.provider.NotePad/notes`. The user can then browse through the list and get information about the items in it.

**action: `android.intent.action.PICK`**

**data: `content://com.google.provider.NotePad/notes`**

Asks the activity to display a list of the notes under `content://com.google.provider.NotePad/notes`. The user can then pick a note from the list, and the activity will return the URI for that item back to the activity that started the `NoteList` activity.

**action: `android.intent.action.GET_CONTENT`**

**data type: `vnd.android.cursor.item/vnd.google.note`**

Asks the activity to supply a single item of Note Pad data.

The second activity, `NoteEditor`, shows users a single note entry and allows them to edit it. It can do two things as described by its two intent filters:

1. 

```
<intent-filter android:label="@string/resolve_edit">
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="com.android.notepad.action.EDIT_NOTE" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

The first, primary, purpose of this activity is to enable the user to interact with a single note — to either `VIEW` the note or `EDIT` it. (The `EDIT_NOTE` category is a synonym for `EDIT`.) The intent would contain the URI for data matching the MIME type `vnd.android.cursor.item/vnd.google.note` — that is, the URI for a single, specific note. It would typically be a URI that was returned by the `PICK` or `GET_CONTENT` actions of the `NoteList` activity.

As before, this filter lists the `DEFAULT` category so that the activity can be launched by intents that don't explicitly specify the `NoteEditor` class.

2. 

```
<intent-filter>
    <action android:name="android.intent.action.INSERT" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

```
<data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

The secondary purpose of this activity is to enable the user to create a new note, which it will `INSERT` into an existing directory of notes. The intent would contain the URI for data matching the MIME type `vnd.android.cursor.dir/vnd.google.note` — that is, the URI for the directory where the note should be placed.

Given these capabilities, the following intents will resolve to the `NoteEditor` activity:

**action: `android.intent.action.VIEW`**

**data: `content://com.google.provider.NotePad/notes/ID`**

Asks the activity to display the content of the note identified by *ID*. (For details on how `content:` URIs specify individual members of a group, see [Content Providers](#).)

**action: `android.intent.action.EDIT`**

**data: `content://com.google.provider.NotePad/notes/ID`**

Asks the activity to display the content of the note identified by *ID*, and to let the user edit it. If the user saves the changes, the activity updates the data for the note in the content provider.

**action: `android.intent.action.INSERT`**

**data: `content://com.google.provider.NotePad/notes`**

Asks the activity to create a new, empty note in the notes list at `content://com.google.provider.NotePad/notes` and allow the user to edit it. If the user saves the note, its URI is returned to the caller.

The last activity, `TitleEditor`, enables the user to edit the title of a note. This could be implemented by directly invoking the activity (by explicitly setting its component name in the Intent), without using an intent filter. But here we take the opportunity to show how to publish alternative operations on existing data:

```
<intent-filter android:label="@string/resolve_title">
    <action android:name="com.android.notepad.action.EDIT_TITLE" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.ALTERNATIVE" />
    <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

The single intent filter for this activity uses a custom action called `"com.android.notepad.action.EDIT_TITLE"`. It must be invoked on a specific note (data type `vnd.android.cursor.item/vnd.google.note`), like the previous `VIEW` and `EDIT` actions. However, here the activity displays the title contained in the note data, not the content of the note itself.

In addition to supporting the usual `DEFAULT` category, the title editor also supports two other standard categories: `ALTERNATIVE` and `SELECTED_ALTERNATIVE`. These categories identify activities that can be presented to users in a menu of options (much as the `LAUNCHER` category identifies activities that should be presented to user in the application launcher). Note that the filter also supplies an explicit label (via `android:label="@string/resolve_title"`) to better control what users see when presented with this activity as an alternative action to the data they are currently viewing. (For more information on these categories and building options menus, see the [PackageManager.queryIntentActivityOptions\(\)](#) and [Menu.addIntentOptions\(\)](#) methods.)

Given these capabilities, the following intent will resolve to the `TitleEditor` activity:

**action: com.android.notePad.action.EDIT\_TITLE**

**data: content://com.google.provider.NotePad/notes/*ID***

Asks the activity to display the title associated with note *ID*, and allow the user to edit the title.

# In this document

1. [What is API Level?](#)
2. [Uses of API Level in Android](#)
3. [Development Considerations](#)
  1. [Application forward compatibility](#)
  2. [Application backward compatibility](#)
  3. [Selecting a platform version and API Level](#)
  4. [Declaring a minimum API Level](#)
  5. [Testing against higher API Levels](#)
4. [Using a Provisional API Level](#)
5. [Filtering the Reference Documentation by API Level](#)



## Google Play Filtering

Google Play uses the `<uses-sdk>` attributes declared in your app manifest to filter your app from devices that do not meet its platform version requirements. Before setting these attributes, make sure that you understand [Google Play filters](#).

### syntax:

```
<uses-sdk android:minSdkVersion="integer"  
        android:targetSdkVersion="integer"  
        android:maxSdkVersion="integer" />
```

### contained in:

[`<manifest>`](#)

### description:

Lets you express an application's compatibility with one or more versions of the Android platform, by means of an API Level integer. The API Level expressed by an application will be compared to the API Level of a given Android system, which may vary among different Android devices.

Despite its name, this element is used to specify the API Level, *not* the version number of the SDK (software development kit) or Android platform. The API Level is always a single integer. You cannot derive the API Level from its associated Android version number (for example, it is not the same as the major version or the sum of the major and minor versions).

Also read the document about [Versioning Your Applications](#).

### attributes:

`android:minSdkVersion`

An integer designating the minimum API Level required for the application to run. The Android system will prevent the user from installing the application if the system's API Level is lower than the value specified in this attribute. You should always declare this attribute.

**Caution:** If you do not declare this attribute, the system assumes a default value of "1", which indicates that your application is compatible with all versions of Android. If your application is *not* compatible with all versions (for instance, it uses APIs introduced in API Level 3) and you have not declared the proper `minSdkVersion`, then when installed on a system with an API Level less than 3,

the application will crash during runtime when attempting to access the unavailable APIs. For this reason, be certain to declare the appropriate API Level in the `minSdkVersion` attribute.

#### **android:targetSdkVersion**

An integer designating the API Level that the application targets. If not set, the default value equals that given to `minSdkVersion`.

This attribute informs the system that you have tested against the target version and the system should not enable any compatibility behaviors to maintain your app's forward-compatibility with the target version. The application is still able to run on older versions (down to `minSdkVersion`).

As Android evolves with each new version, some behaviors and even appearances might change. However, if the API level of the platform is higher than the version declared by your app's `targetSdkVersion`, the system may enable compatibility behaviors to ensure that your app continues to work the way you expect. You can disable such compatibility behaviors by specifying `targetSdkVersion` to match the API level of the platform on which it's running. For example, setting this value to "11" or higher allows the system to apply a new default theme (Holo) to your app when running on Android 3.0 or higher and also disables [screen compatibility mode](#) when running on larger screens (because support for API level 11 implicitly supports larger screens).

There are many compatibility behaviors that the system may enable based on the value you set for this attribute. Several of these behaviors are described by the corresponding platform versions in the [Build.VERSION\\_CODES](#) reference.

To maintain your application along with each Android release, you should increase the value of this attribute to match the latest API level, then thoroughly test your application on the corresponding platform version.

Introduced in: API Level 4

#### **android:maxSdkVersion**

An integer designating the maximum API Level on which the application is designed to run.

In Android 1.5, 1.6, 2.0, and 2.0.1, the system checks the value of this attribute when installing an application and when re-validating the application after a system update. In either case, if the application's `maxSdkVersion` attribute is lower than the API Level used by the system itself, then the system will not allow the application to be installed. In the case of re-validation after system update, this effectively removes your application from the device.

To illustrate how this attribute can affect your application after system updates, consider the following example:

An application declaring `maxSdkVersion="5"` in its manifest is published on Google Play. A user whose device is running Android 1.6 (API Level 4) downloads and installs the app. After a few weeks, the user receives an over-the-air system update to Android 2.0 (API Level 5). After the update is installed, the system checks the application's `maxSdkVersion` and successfully re-validates it. The application functions as normal. However, some time later, the device receives another system update, this time to Android 2.0.1 (API Level 6). After the update, the system can no longer re-validate the application because the system's own API Level (6) is now higher than the maximum supported by the application (5). The system prevents the application from being visible to the user, in effect removing it from the device.

**Warning:** Declaring this attribute is not recommended. First, there is no need to set the attribute as means of blocking deployment of your application onto new versions of the Android platform as they

are released. By design, new versions of the platform are fully backward-compatible. Your application should work properly on new versions, provided it uses only standard APIs and follows development best practices. Second, note that in some cases, declaring the attribute can **result in your application being removed from users' devices after a system update** to a higher API Level. Most devices on which your application is likely to be installed will receive periodic system updates over the air, so you should consider their effect on your application before setting this attribute.

Introduced in: API Level 4

Future versions of Android (beyond Android 2.0.1) will no longer check or enforce the `maxSdkVersion` attribute during installation or re-validation. Google Play will continue to use the attribute as a filter, however, when presenting users with applications available for download.

**introduced in:**

API Level 1

## What is API Level?

API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.

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

- A core set of packages and classes
- A set of XML elements and attributes for declaring a manifest file
- A set of XML elements and attributes for declaring and accessing resources
- A set of Intents
- A set of permissions that applications can request, as well as permission enforcements included in the system

Each successive version of the Android platform can include updates to the Android application framework API that it delivers.

Updates to the framework API are designed so that the new API remains compatible with earlier versions of the API. That is, most changes in the API are additive and introduce new or replacement functionality. As parts of the API are upgraded, the older replaced parts are deprecated but are not removed, so that existing applications can still use them. In a very small number of cases, parts of the API may be modified or removed, although typically such changes are only needed to ensure API robustness and application or system security. All other API parts from earlier revisions are carried forward without modification.

The framework API that an Android platform delivers is specified using an integer identifier called "API Level". Each Android platform version supports exactly one API Level, although support is implicit for all earlier API Levels (down to API Level 1). The initial release of the Android platform provided API Level 1 and subsequent releases have incremented the API Level.

The table below specifies the API Level supported by each version of the Android platform. For information about the relative numbers of devices that are running each version, see the [Platform Versions dashboards page](#).

Platform Version	API Level	VERSION_CODE	Notes
<a href="#">Android 4.3</a>	<a href="#">18</a>	<a href="#">JELLY_BEAN_MR2</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.2, 4.2.2</a>	<a href="#">17</a>	<a href="#">JELLY_BEAN_MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.1, 4.1.1</a>	<a href="#">16</a>	<a href="#">JELLY_BEAN</a>	<a href="#">Platform Highlights</a>

<a href="#">Android 4.0.3, 4.0.4</a>	<a href="#">15</a>	<a href="#">ICE CREAM SANDWICH MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.0, 4.0.1, 4.0.2</a>	<a href="#">14</a>	<a href="#">ICE CREAM SANDWICH</a>	
<a href="#">Android 3.2</a>	<a href="#">13</a>	<a href="#">HONEYCOMB MR2</a>	
<a href="#">Android 3.1.x</a>	<a href="#">12</a>	<a href="#">HONEYCOMB MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 3.0.x</a>	<a href="#">11</a>	<a href="#">HONEYCOMB</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.4</a>	<a href="#">10</a>	<a href="#">GINGERBREAD MR1</a>	
<a href="#">Android 2.3.3</a>			
<a href="#">Android 2.3.2</a>			<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.1</a>	<a href="#">9</a>	<a href="#">GINGERBREAD</a>	
<a href="#">Android 2.3</a>			
<a href="#">Android 2.2.x</a>	<a href="#">8</a>	<a href="#">FROYO</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.1.x</a>	<a href="#">7</a>	<a href="#">ECLAIR_MR1</a>	
<a href="#">Android 2.0.1</a>	<a href="#">6</a>	<a href="#">ECLAIR_0_1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.0</a>	<a href="#">5</a>	<a href="#">ECLAIR</a>	
<a href="#">Android 1.6</a>	<a href="#">4</a>	<a href="#">DONUT</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.5</a>	<a href="#">3</a>	<a href="#">CUPCAKE</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.1</a>	<a href="#">2</a>	<a href="#">BASE_1_1</a>	
Android 1.0	<a href="#">1</a>	<a href="#">BASE</a>	

## Uses of API Level in Android

The API Level identifier serves a key role in ensuring the best possible experience for users and application developers:

- It lets the Android platform describe the maximum framework API revision that it supports
- It lets applications describe the framework API revision that they require
- It lets the system negotiate the installation of applications on the user's device, such that version-incompatible applications are not installed.

Each Android platform version stores its API Level identifier internally, in the Android system itself.

Applications can use a manifest element provided by the framework API — `<uses-sdk>` — to describe the minimum and maximum API Levels under which they are able to run, as well as the preferred API Level that they are designed to support. The element offers three key attributes:

- `android:minSdkVersion` — Specifies the minimum API Level on which the application is able to run. The default value is "1".
- `android:targetSdkVersion` — Specifies the API Level on which the application is designed to run. In some cases, this allows the application to use manifest elements or behaviors defined in the target API Level, rather than being restricted to using only those defined for the minimum API Level.
- `android:maxSdkVersion` — Specifies the maximum API Level on which the application is able to run. **Important:** Please read the [`<uses-sdk>`](#) documentation before using this attribute.

For example, to specify the minimum system API Level that an application requires in order to run, the application would include in its manifest a `<uses-sdk>` element with a `android:minSdkVersion` attribute. The value of `android:minSdkVersion` would be the integer corresponding to the API Level of the earliest version of the Android platform under which the application can run.

When the user attempts to install an application, or when revalidating an application after a system update, the Android system first checks the `<uses-sdk>` attributes in the application's manifest and compares the values against its own internal API Level. The system allows the installation to begin only if these conditions are met:

- If a `android:minSdkVersion` attribute is declared, its value must be less than or equal to the system's API Level integer. If not declared, the system assumes that the application requires API Level 1.
- If a `android:maxSdkVersion` attribute is declared, its value must be equal to or greater than the system's API Level integer. If not declared, the system assumes that the application has no maximum API Level. Please read the [`<uses-sdk>`](#) documentation for more information about how the system handles this attribute.

When declared in an application's manifest, a `<uses-sdk>` element might look like this:

```
<manifest>
  <uses-sdk android:minSdkVersion="5" />
  ...
</manifest>
```

The principal reason that an application would declare an API Level in `android:minSdkVersion` is to tell the Android system that it is using APIs that were *introduced* in the API Level specified. If the application were to be somehow installed on a platform with a lower API Level, then it would crash at run-time when it tried to access APIs that don't exist. The system prevents such an outcome by not allowing the application to be installed if the lowest API Level it requires is higher than that of the platform version on the target device.

For example, the [`android.appwidget`](#) package was introduced with API Level 3. If an application uses that API, it must declare a `android:minSdkVersion` attribute with a value of "3". The application will then be installable on platforms such as Android 1.5 (API Level 3) and Android 1.6 (API Level 4), but not on the Android 1.1 (API Level 2) and Android 1.0 platforms (API Level 1).

For more information about how to specify an application's API Level requirements, see the [`<uses-sdk>`](#) section of the manifest file documentation.

## Development Considerations

The sections below provide information related to API level that you should consider when developing your application.

### Application forward compatibility

Android applications are generally forward-compatible with new versions of the Android platform.

Because almost all changes to the framework API are additive, an Android application developed using any given version of the API (as specified by its API Level) is forward-compatible with later versions of the Android platform and higher API levels. The application should be able to run on all later versions of the Android platform, except in isolated cases where the application uses a part of the API that is later removed for some reason.

Forward compatibility is important because many Android-powered devices receive over-the-air (OTA) system updates. The user may install your application and use it successfully, then later receive an OTA update to a new version of the Android platform. Once the update is installed, your application will run in a new run-time version of the environment, but one that has the API and system capabilities that your application depends on.

In some cases, changes *below* the API, such those in the underlying system itself, may affect your application when it is run in the new environment. For that reason it's important for you, as the application developer, to understand how the application will look and behave in each system environment. To help you test your application on various versions of the Android platform, the Android SDK includes multiple platforms that you can

download. Each platform includes a compatible system image that you can run in an AVD, to test your application.

## Application backward compatibility

Android applications are not necessarily backward compatible with versions of the Android platform older than the version against which they were compiled.

Each new version of the Android platform can include new framework APIs, such as those that give applications access to new platform capabilities or replace existing API parts. The new APIs are accessible to applications when running on the new platform and, as mentioned above, also when running on later versions of the platform, as specified by API Level. Conversely, because earlier versions of the platform do not include the new APIs, applications that use the new APIs are unable to run on those platforms.

Although it's unlikely that an Android-powered device would be downgraded to a previous version of the platform, it's important to realize that there are likely to be many devices in the field that run earlier versions of the platform. Even among devices that receive OTA updates, some might lag and might not receive an update for a significant amount of time.

## Selecting a platform version and API Level

When you are developing your application, you will need to choose the platform version against which you will compile the application. In general, you should compile your application against the lowest possible version of the platform that your application can support.

You can determine the lowest possible platform version by compiling the application against successively lower build targets. After you determine the lowest version, you should create an AVD using the corresponding platform version (and API Level) and fully test your application. Make sure to declare a `android:minSdkVersion` attribute in the application's manifest and set its value to the API Level of the platform version.

## Declaring a minimum API Level

If you build an application that uses APIs or system features introduced in the latest platform version, you should set the `android:minSdkVersion` attribute to the API Level of the latest platform version. This ensures that users will only be able to install your application if their devices are running a compatible version of the Android platform. In turn, this ensures that your application can function properly on their devices.

If your application uses APIs introduced in the latest platform version but does *not* declare a `android:minSdkVersion` attribute, then it will run properly on devices running the latest version of the platform, but *not* on devices running earlier versions of the platform. In the latter case, the application will crash at runtime when it tries to use APIs that don't exist on the earlier versions.

## Testing against higher API Levels

After compiling your application, you should make sure to test it on the platform specified in the application's `android:minSdkVersion` attribute. To do so, create an AVD that uses the platform version required by your application. Additionally, to ensure forward-compatibility, you should run and test the application on all platforms that use a higher API Level than that used by your application.

The Android SDK includes multiple platform versions that you can use, including the latest version, and provides an updater tool that you can use to download other platform versions as necessary.

To access the updater, use the `android` command-line tool, located in the `<sdk>/tools` directory. You can launch the SDK updater by executing `android sdk`. You can also simply double-click the `android.bat` (Windows) or `android` (OS X/Linux) file. In ADT, you can also access the updater by selecting **Window > Android SDK Manager**.

To run your application against different platform versions in the emulator, create an AVD for each platform version that you want to test. For more information about AVDs, see [Creating and Managing Virtual Devices](#). If you are using a physical device for testing, ensure that you know the API Level of the Android platform it runs. See the table at the top of this document for a list of platform versions and their API Levels.

## Using a Provisional API Level

In some cases, an "Early Look" Android SDK platform may be available. To let you begin developing on the platform although the APIs may not be final, the platform's API Level integer will not be specified. You must instead use the platform's *provisional API Level* in your application manifest, in order to build applications against the platform. A provisional API Level is not an integer, but a string matching the codename of the unreleased platform version. The provisional API Level will be specified in the release notes for the Early Look SDK release notes and is case-sensitive.

The use of a provisional API Level is designed to protect developers and device users from inadvertently publishing or installing applications based on the Early Look framework API, which may not run properly on actual devices running the final system image.

The provisional API Level will only be valid while using the Early Look SDK and can only be used to run applications in the emulator. An application using the provisional API Level can never be installed on an Android device. At the final release of the platform, you must replace any instances of the provisional API Level in your application manifest with the final platform's actual API Level integer.

## Filtering the Reference Documentation by API Level

Reference documentation pages on the Android Developers site offer a "Filter by API Level" control in the top-right area of each page. You can use the control to show documentation only for parts of the API that are actually accessible to your application, based on the API Level that it specifies in the `android:minSdkVersion` attribute of its manifest file.

To use filtering, select the checkbox to enable filtering, just below the page search box. Then set the "Filter by API Level" control to the same API Level as specified by your application. Notice that APIs introduced in a later API Level are then grayed out and their content is masked, since they would not be accessible to your application.

Filtering by API Level in the documentation does not provide a view of what is new or introduced in each API Level — it simply provides a way to view the entire API associated with a given API Level, while excluding API elements introduced in later API Levels.

If you decide that you don't want to filter the API documentation, just disable the feature using the checkbox. By default, API Level filtering is disabled, so that you can view the full framework API, regardless of API Level.

Also note that the reference documentation for individual API elements specifies the API Level at which each element was introduced. The API Level for packages and classes is specified as "Since <api level>" at the top-right corner of the content area on each documentation page. The API Level for class members is specified in their detailed description headers, at the right margin.

# <supports-screens>

## syntax:

```
<supports-screens android:resizeable=["true" | "false"]  
                  android:smallScreens=["true" | "false"]  
                  android:normalScreens=["true" | "false"]  
                  android:largeScreens=["true" | "false"]  
                  android:xlargeScreens=["true" | "false"]  
                  android:anyDensity=["true" | "false"]  
                  android:requiresSmallestWidthDp="integer"  
                  android:compatibleWidthLimitDp="integer"  
                  android:largestWidthLimitDp="integer"/>
```

## contained in:

[<manifest>](#)

## description:

Lets you specify the screen sizes your application supports and enable [screen compatibility mode](#) for screens larger than what your application supports. It's important that you always use this element in your application to specify the screen sizes your application supports.

An application "supports" a given screen size if it resizes properly to fill the entire screen. Normal resizing applied by the system works well for most applications and you don't have to do any extra work to make your application work on screens larger than a handset device. However, it's often important that you optimize your application's UI for different screen sizes by providing [alternative layout resources](#). For instance, you might want to modify the layout of an activity when it is on a tablet compared to when running on a handset device.

However, if your application does not work well when resized to fit different screen sizes, you can use the attributes of the <supports-screens> element to control whether your application should be distributed to smaller screens or have its UI scaled up ("zoomed") to fit larger screens using the system's [screen compatibility mode](#). When you have not designed for larger screen sizes and the normal resizing does not achieve the appropriate results, screen compatibility mode will scale your UI by emulating a *normal* size screen and medium density, then zooming in so that it fills the entire screen. Beware that this causes pixelation and blurring of your UI, so it's better if you optimize your UI for large screens.

**Note:** Android 3.2 introduces new attributes: `android:requiresSmallestWidthDp`, `android:compatibleWidthLimitDp`, and `android:largestWidthLimitDp`. If you're developing your application for Android 3.2 and higher, you should use these attributes to declare your screen size support, instead of the attributes based on generalized screen sizes.

For more information about how to properly support different screen sizes so that you can avoid using screen compatibility mode with your application, read [Supporting Multiple Screens](#).

## attributes:

### `android:resizeable`

Indicates whether the application is resizeable for different screen sizes. This attribute is true, by default. If set false, the system will run your application in [screen compatibility mode](#) on large screens.

**This attribute is deprecated.** It was introduced to help applications transition from Android 1.5 to 1.6, when support for multiple screens was first introduced. You should not use it.

#### **android:smallScreens**

Indicates whether the application supports smaller screen form-factors. A small screen is defined as one with a smaller aspect ratio than the "normal" (traditional HVGA) screen. An application that does not support small screens *will not be available* for small screen devices from external services (such as Google Play), because there is little the platform can do to make such an application work on a smaller screen. This is "true" by default.

#### **android:normalScreens**

Indicates whether an application supports the "normal" screen form-factors. Traditionally this is an HVGA medium density screen, but WQVGA low density and WVGA high density are also considered to be normal. This attribute is "true" by default.

#### **android:largeScreens**

Indicates whether the application supports larger screen form-factors. A large screen is defined as a screen that is significantly larger than a "normal" handset screen, and thus might require some special care on the application's part to make good use of it, though it may rely on resizing by the system to fill the screen.

The default value for this actually varies between some versions, so it's better if you explicitly declare this attribute at all times. Beware that setting it "false" will generally enable [screen compatibility mode](#).

#### **android:xlargeScreens**

Indicates whether the application supports extra large screen form-factors. An xlarge screen is defined as a screen that is significantly larger than a "large" screen, such as a tablet (or something larger) and may require special care on the application's part to make good use of it, though it may rely on resizing by the system to fill the screen.

The default value for this actually varies between some versions, so it's better if you explicitly declare this attribute at all times. Beware that setting it "false" will generally enable [screen compatibility mode](#).

This attribute was introduced in API level 9.

#### **android:anyDensity**

Indicates whether the application includes resources to accommodate any screen density.

For applications that support Android 1.6 (API level 4) and higher, this is "true" by default and **you should not set it "false"** unless you're absolutely certain that it's necessary for your application to work. The only time it might be necessary to disable this is if your app directly manipulates bitmaps (see the [Supporting Multiple Screens](#) document for more information).

#### **android:requiresSmallestWidthDp**

Specifies the minimum smallestWidth required. The smallestWidth is the shortest dimension of the screen space (in dp units) that must be available to your application UI—that is, the shortest of the available screen's two dimensions. So, in order for a device to be considered compatible with your application, the device's smallestWidth must be equal to or greater than this value. (Usually, the value you supply for this is the "smallest width" that your layout supports, regardless of the screen's current orientation.)

For example, a typical handset screen has a smallestWidth of 320dp, a 7" tablet has a smallestWidth of 600dp, and a 10" tablet has a smallestWidth of 720dp. These values are generally the smallestWidth because they are the shortest dimension of the screen's available space.

The size against which your value is compared takes into account screen decorations and system UI. For example, if the device has some persistent UI elements on the display, the system declares the device's smallestWidth as one that is smaller than the actual screen size, accounting for these UI elements because those are screen pixels not available for your UI. Thus, the value you use should be the minimum width required by your layout, regardless of the screen's current orientation.

If your application properly resizes for smaller screen sizes (down to the *small* size or a minimum width of 320dp), you do not need to use this attribute. Otherwise, you should use a value for this attribute that matches the smallest value used by your application for the [smallest screen width qualifier](#) (`sw<N>dp`).

**Caution:** The Android system does not pay attention to this attribute, so it does not affect how your application behaves at runtime. Instead, it is used to enable filtering for your application on services such as Google Play. However, **Google Play currently does not support this attribute for filtering** (on Android 3.2), so you should continue using the other size attributes if your application does not support small screens.

This attribute was introduced in API level 13.

#### **`android:compatibleWidthLimitDp`**

This attribute allows you to enable [screen compatibility mode](#) as a user-optimal feature by specifying the maximum "smallest screen width" for which your application is designed. If the smallest side of a device's available screen is greater than your value here, users can still install your application, but are offered to run it in screen compatibility mode. By default, screen compatibility mode is disabled and your layout is resized to fit the screen as usual, but a button is available in the system bar that allows the user to toggle screen compatibility mode on and off.

If your application is compatible with all screen sizes and its layout properly resizes, you do not need to use this attribute.

**Note:** Currently, screen compatibility mode emulates only handset screens with a 320dp width, so screen compatibility mode is not applied if your value for `android:compatibleWidthLimitDp` is larger than 320.

This attribute was introduced in API level 13.

#### **`android:largestWidthLimitDp`**

This attribute allows you to force-enable [screen compatibility mode](#) by specifying the maximum "smallest screen width" for which your application is designed. If the smallest side of a device's available screen is greater than your value here, the application runs in screen compatibility mode with no way for the user to disable it.

If your application is compatible with all screen sizes and its layout properly resizes, you do not need to use this attribute. Otherwise, you should first consider using the [`android:compatibleWidthLimitDp`](#) attribute. You should use the `android:largestWidthLimitDp` attribute only when your application is functionally broken when resized for larger screens and screen compatibility mode is the only way that users should use your application.

**Note:** Currently, screen compatibility mode emulates only handset screens with a 320dp width, so screen compatibility mode is not applied if your value for `android:largestWidthLimitDp` is larger than 320.

This attribute was introduced in API level 13.

**introduced in:**

API Level 4

**see also:**

- [Supporting Multiple Screens](#)
- [DisplayMetrics](#)

# Supporting Multiple Screens

## Quickview

- Android runs on devices that have different screen sizes and densities.
- The screen on which your application is displayed can affect its user interface.
- The system handles most of the work of adapting your app to the current screen.
- You should create screen-specific resources for precise control of your UI.

## In this document

1. [Overview of Screen Support](#)
  1. [Terms and concepts](#)
  2. [Range of screens supported](#)
  3. [Density independence](#)
2. [How to Support Multiple Screens](#)
  1. [Using configuration qualifiers](#)
  2. [Designing alternative layouts and drawables](#)
3. [Declaring Tablet Layouts for Android 3.2](#) new!
  1. [Using new size qualifiers](#)
  2. [Configuration examples](#)
  3. [Declaring screen size support](#)
4. [Best Practices](#)
5. [Additional Density Considerations](#)
  1. [Scaling Bitmap objects created at runtime](#)
  2. [Converting dp units to pixel units](#)
6. [How to Test Your Application on Multiple Screens](#)

## Related samples

1. [Multiple Resolutions](#)

## See also

1. [Thinking Like a Web Designer](#)
2. [Providing Alternative Resources](#)
3. [Icon Design Guidelines](#)
4. [Managing Virtual Devices](#)

Android runs on a variety of devices that offer different screen sizes and densities. For applications, the Android system provides a consistent development environment across devices and handles most of the work to adjust each application's user interface to the screen on which it is displayed. At the same time, the system provides APIs that allow you to control your application's UI for specific screen sizes and densities, in order to optimize your UI design for different screen configurations. For example, you might want a UI for tablets that's different from the UI for handsets.

Although the system performs scaling and resizing to make your application work on different screens, you should make the effort to optimize your application for different screen sizes and densities. In doing so, you maximize the user experience for all devices and your users believe that your application was actually designed for *their* devices—rather than simply stretched to fit the screen on their devices.

By following the practices described in this document, you can create an application that displays properly and provides an optimized user experience on all supported screen configurations, using a single .apk file.

**Note:** The information in this document assumes that your application is designed for Android 1.6 (API Level 4) or higher. If your application supports Android 1.5 or lower, please first read [Strategies for Android 1.5](#).

Also, be aware that **Android 3.2 has introduced new APIs** that allow you to more precisely control the layout resources your application uses for different screen sizes. These new features are especially important if you're developing an application that's optimized for tablets. For details, see the section about [Declaring Tablet Layouts for Android 3.2](#).

## Overview of Screens Support

This section provides an overview of Android's support for multiple screens, including: an introduction to the terms and concepts used in this document and in the API, a summary of the screen configurations that the system supports, and an overview of the API and underlying screen-compatibility features.

### Terms and concepts

#### *Screen size*

Actual physical size, measured as the screen's diagonal.

For simplicity, Android groups all actual screen sizes into four generalized sizes: small, normal, large, and extra large.

#### *Screen density*

The quantity of pixels within a physical area of the screen; usually referred to as dpi (dots per inch). For example, a "low" density screen has fewer pixels within a given physical area, compared to a "normal" or "high" density screen.

For simplicity, Android groups all actual screen densities into four generalized densities: low, medium, high, and extra high.

#### *Orientation*

The orientation of the screen from the user's point of view. This is either landscape or portrait, meaning that the screen's aspect ratio is either wide or tall, respectively. Be aware that not only do different devices operate in different orientations by default, but the orientation can change at runtime when the user rotates the device.

#### *Resolution*

The total number of physical pixels on a screen. When adding support for multiple screens, applications do not work directly with resolution; applications should be concerned only with screen size and density, as specified by the generalized size and density groups.

#### *Density-independent pixel (dp)*

A virtual pixel unit that you should use when defining UI layout, to express layout dimensions or position in a density-independent way.

The density-independent pixel is equivalent to one physical pixel on a 160 dpi screen, which is the baseline density assumed by the system for a "medium" density screen. At runtime, the system transparently handles any scaling of the dp units, as necessary, based on the actual density of the screen in use. The conversion of dp units to screen pixels is simple:  $\text{px} = \text{dp} * (\text{dpi} / 160)$ . For example, on a 240 dpi screen, 1 dp equals 1.5 physical pixels. You should always use dp units when defining your application's UI, to ensure proper display of your UI on screens with different densities.

## Range of screens supported

Starting with Android 1.6 (API Level 4), Android provides support for multiple screen sizes and densities, reflecting the many different screen configurations that a device may have. You can use features of the Android system to optimize your application's user interface for each screen configuration and ensure that your application not only renders properly, but provides the best user experience possible on each screen.

To simplify the way that you design your user interfaces for multiple screens, Android divides the range of actual screen sizes and densities into:

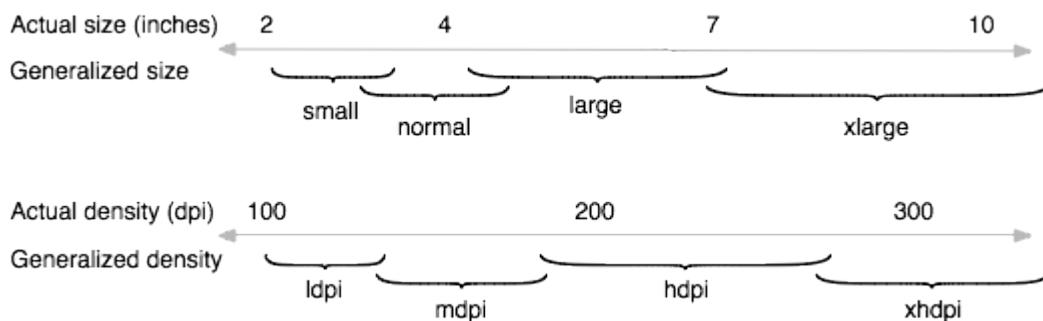
- A set of four generalized **sizes**: *small*, *normal*, *large*, and *xlarge*

**Note:** Beginning with Android 3.2 (API level 13), these size groups are deprecated in favor of a new technique for managing screen sizes based on the available screen width. If you're developing for Android 3.2 and greater, see [Declaring Tablet Layouts for Android 3.2](#) for more information.

- A set of four generalized **densities**: *ldpi* (low), *mdpi* (medium), *hdpi* (high), and *xhdpi* (extra high)

The generalized sizes and densities are arranged around a baseline configuration that is a *normal* size and *mdpi* (medium) density. This baseline is based upon the screen configuration for the first Android-powered device, the T-Mobile G1, which has an HVGA screen (until Android 1.6, this was the only screen configuration that Android supported).

Each generalized size and density spans a range of actual screen sizes and densities. For example, two devices that both report a screen size of *normal* might have actual screen sizes and aspect ratios that are slightly different when measured by hand. Similarly, two devices that report a screen density of *hdpi* might have real pixel densities that are slightly different. Android makes these differences abstract to applications, so you can provide UI designed for the generalized sizes and densities and let the system handle any final adjustments as necessary. Figure 1 illustrates how different sizes and densities are roughly categorized into the different size and density groups.



**Figure 1.** Illustration of how Android roughly maps actual sizes and densities to generalized sizes and densities (figures are not exact).

As you design your UI for different screen sizes, you'll discover that each design requires a minimum amount of space. So, each generalized screen size above has an associated minimum resolution that's defined by the system. These minimum sizes are in "dp" units—the same units you should use when defining your layouts—which allows the system to avoid worrying about changes in screen density.

- *xlarge* screens are at least 960dp x 720dp
- *large* screens are at least 640dp x 480dp
- *normal* screens are at least 470dp x 320dp
- *small* screens are at least 426dp x 320dp

**Note:** These minimum screen sizes were not as well defined prior to Android 3.0, so you may encounter some devices that are mis-classified between normal and large. These are also based on the physical resolution of the screen, so may vary across devices—for example a 1024x720 tablet with a system bar actually has a bit less space available to the application due to it being used by the system bar.

To optimize your application's UI for the different screen sizes and densities, you can provide [alternative resources](#) for any of the generalized sizes and densities. Typically, you should provide alternative layouts for some of the different screen sizes and alternative bitmap images for different screen densities. At runtime, the system uses the appropriate resources for your application, based on the generalized size or density of the current device screen.

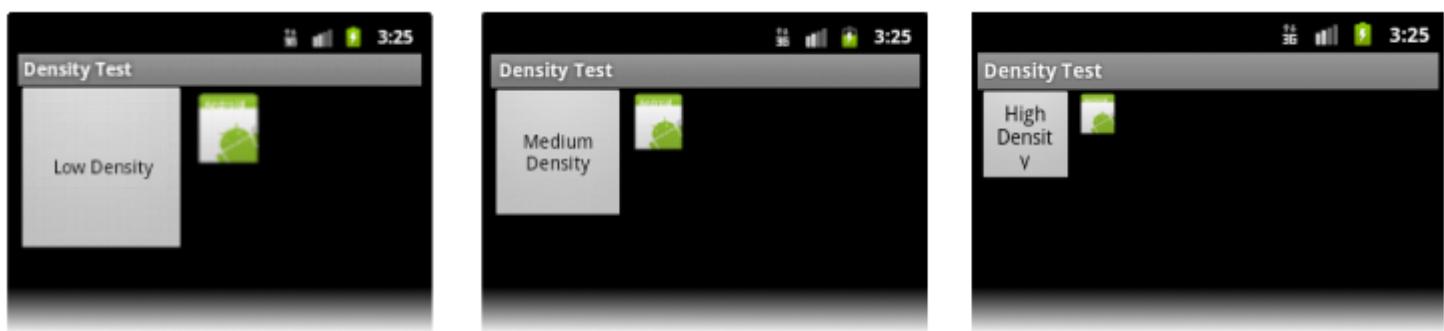
You do not need to provide alternative resources for every combination of screen size and density. The system provides robust compatibility features that can handle most of the work of rendering your application on any device screen, provided that you've implemented your UI using techniques that allow it to gracefully resize (as described in the [Best Practices](#), below).

**Note:** The characteristics that define a device's generalized screen size and density are independent from each other. For example, a WVGA high-density screen is considered a normal size screen because its physical size is about the same as the T-Mobile G1 (Android's first device and baseline screen configuration). On the other hand, a WVGA medium-density screen is considered a large size screen. Although it offers the same resolution (the same number of pixels), the WVGA medium-density screen has a lower screen density, meaning that each pixel is physically larger and, thus, the entire screen is larger than the baseline (normal size) screen.

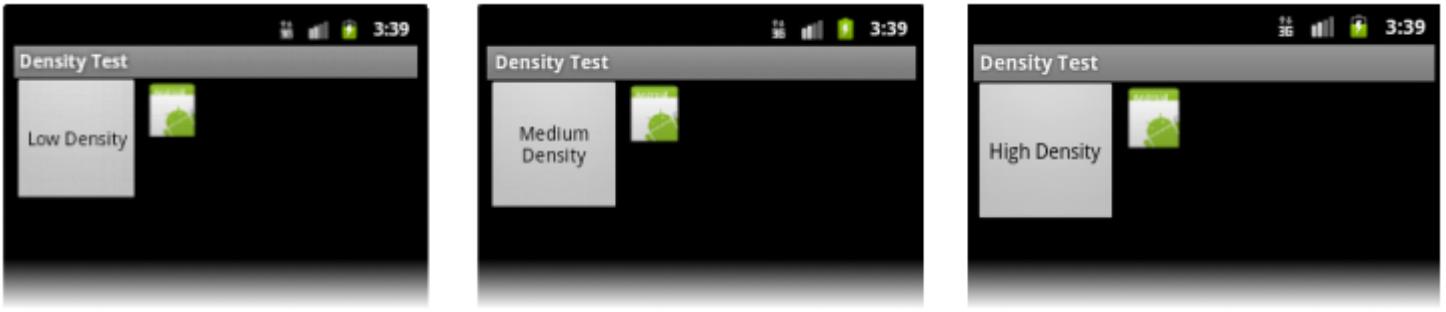
## Density independence

Your application achieves "density independence" when it preserves the physical size (from the user's point of view) of user interface elements when displayed on screens with different densities.

Maintaining density independence is important because, without it, a UI element (such as a button) appears physically larger on a low density screen and smaller on a high density screen. Such density-related size changes can cause problems in your application layout and usability. Figures 2 and 3 show the difference between an application when it does not provide density independence and when it does, respectively.



**Figure 2.** Example application without support for different densities, as shown on low, medium, and high density screens.



**Figure 3.** Example application with good support for different densities (it's density independent), as shown on low, medium, and high density screens.

The Android system helps your application achieve density independence in two ways:

- The system scales dp units as appropriate for the current screen density
- The system scales drawable resources to the appropriate size, based on the current screen density, if necessary

In figure 2, the text view and bitmap drawable have dimensions specified in pixels (px units), so the views are physically larger on a low density screen and smaller on a high density screen. This is because although the actual screen sizes may be the same, the high density screen has more pixels per inch (the same amount of pixels fit in a smaller area). In figure 3, the layout dimensions are specified in density-independent pixels (dp units). Because the baseline for density-independent pixels is a medium-density screen, the device with a medium-density screen looks the same as it does in figure 2. For the low-density and high-density screens, however, the system scales the density-independent pixel values down and up, respectively, to fit the screen as appropriate.

In most cases, you can ensure density independence in your application simply by specifying all layout dimension values in density-independent pixels (dp units) or with "wrap\_content", as appropriate. The system then scales bitmap drawables as appropriate in order to display at the appropriate size, based on the appropriate scaling factor for the current screen's density.

However, bitmap scaling can result in blurry or pixelated bitmaps, which you might notice in the above screenshots. To avoid these artifacts, you should provide alternative bitmap resources for different densities. For example, you should provide higher-resolution bitmaps for high-density screens and the system will use those instead of resizing the bitmap designed for medium-density screens. The following section describes more about how to supply alternative resources for different screen configurations.

## How to Support Multiple Screens

The foundation of Android's support for multiple screens is its ability to manage the rendering of an application's layout and bitmap drawables in an appropriate way for the current screen configuration. The system handles most of the work to render your application properly on each screen configuration by scaling layouts to fit the screen size/density and scaling bitmap drawables for the screen density, as appropriate. To more gracefully handle different screen configurations, however, you should also:

- **Explicitly declare in the manifest which screen sizes your application supports**

By declaring which screen sizes your application supports, you can ensure that only devices with the screens you support can download your application. Declaring support for different screen sizes can also affect how the system draws your application on larger screens—specifically, whether your application runs in [screen compatibility mode](#).

To declare the screen sizes your application supports, you should include the [`<supports-screens>`](#) element in your manifest file.

- **Provide different layouts for different screen sizes**

By default, Android resizes your application layout to fit the current device screen. In most cases, this works fine. In other cases, your UI might not look as good and might need adjustments for different screen sizes. For example, on a larger screen, you might want to adjust the position and size of some elements to take advantage of the additional screen space, or on a smaller screen, you might need to adjust sizes so that everything can fit on the screen.

The configuration qualifiers you can use to provide size-specific resources are `small`, `normal`, `large`, and `xlarge`. For example, layouts for an extra large screen should go in `layout-xlarge/`.

Beginning with Android 3.2 (API level 13), the above size groups are deprecated and you should instead use the `sw<N>dp` configuration qualifier to define the smallest available width required by your layout resources. For example, if your multi-pane tablet layout requires at least 600dp of screen width, you should place it in `layout-sw600dp/`. Using the new techniques for declaring layout resources is discussed further in the section about [Declaring Tablet Layouts for Android 3.2](#).

- **Provide different bitmap drawables for different screen densities**

By default, Android scales your bitmap drawables (`.png`, `.jpg`, and `.gif` files) and Nine-Patch drawables (`.9.png` files) so that they render at the appropriate physical size on each device. For example, if your application provides bitmap drawables only for the baseline, medium screen density (`mdpi`), then the system scales them up when on a high-density screen, and scales them down when on a low-density screen. This scaling can cause artifacts in the bitmaps. To ensure your bitmaps look their best, you should include alternative versions at different resolutions for different screen densities.

The configuration qualifiers you can use for density-specific resources are `ldpi` (low), `mdpi` (medium), `hdpi` (high), and `xhdpi` (extra high). For example, bitmaps for high-density screens should go in `drawable-hdpi/`.

The size and density configuration qualifiers correspond to the generalized sizes and densities described in [Range of screens supported](#), above.

**Note:** If you're not familiar with configuration qualifiers and how the system uses them to apply alternative resources, read [Providing Alternative Resources](#) for more information.

At runtime, the system ensures the best possible display on the current screen with the following procedure for any given resource:

1. The system uses the appropriate alternative resource

Based on the size and density of the current screen, the system uses any size- and density-specific resource provided in your application. For example, if the device has a high-density screen and the application requests a drawable resource, the system looks for a drawable resource directory that best matches the device configuration. Depending on the other alternative resources available, a resource directory with the `hdpi` qualifier (such as `drawable-hdpi/`) might be the best match, so the system uses the drawable resource from this directory.

2. If no matching resource is available, the system uses the default resource and scales it up or down as needed to match the current screen size and density

The "default" resources are those that are not tagged with a configuration qualifier. For example, the resources in `drawable/` are the default drawable resources. The system assumes that default resources are designed for the baseline screen size and density, which is a normal screen size and a medium density. As such, the system scales default density resources up for high-density screens and down for low-density screens, as appropriate.

However, when the system is looking for a density-specific resource and does not find it in the density-specific directory, it won't always use the default resources. The system may instead use one of the other density-specific resources in order to provide better results when scaling. For example, when looking for a low-density resource and it is not available, the system prefers to scale-down the high-density version of the resource, because the system can easily scale a high-density resource down to low-density by a factor of 0.5, with fewer artifacts, compared to scaling a medium-density resource by a factor of 0.75.

For more information about how Android selects alternative resources by matching configuration qualifiers to the device configuration, read [How Android Finds the Best-matching Resource](#).

## Using configuration qualifiers

Android supports several configuration qualifiers that allow you to control how the system selects your alternative resources based on the characteristics of the current device screen. A configuration qualifier is a string that you can append to a resource directory in your Android project and specifies the configuration for which the resources inside are designed.

To use a configuration qualifier:

1. Create a new directory in your project's `res/` directory and name it using the format:

```
<resources_name>-<qualifier>
  • <resources_name> is the standard resource name (such as drawable or layout).
  • <qualifier> is a configuration qualifier from table 1, below, specifying the screen configuration for which these resources are to be used (such as hdpi or xlarge).
```

You can use more than one `<qualifier>` at a time—simply separate each qualifier with a dash.

2. Save the appropriate configuration-specific resources in this new directory. The resource files must be named exactly the same as the default resource files.

For example, `xlarge` is a configuration qualifier for extra large screens. When you append this string to a resource directory name (such as `layout-xlarge`), it indicates to the system that these resources are to be used on devices that have an extra large screen.

**Table 1.** Configuration qualifiers that allow you to provide special resources for different screen configurations.

Screen characteristic	Qualifier	Description
Size	small	Resources for <i>small</i> size screens.
	normal	Resources for <i>normal</i> size screens. (This is the baseline size.)
	large	Resources for <i>large</i> size screens.
	xlarge	Resources for <i>extra large</i> size screens.
Density	ldpi	Resources for low-density ( <i>ldpi</i> ) screens (~120dpi).
	mdpi	Resources for medium-density ( <i>mdpi</i> ) screens (~160dpi). (This is the baseline density)
	hdpi	Resources for high-density ( <i>hdpi</i> ) screens (~240dpi).

	xhdpi	Resources for extra high-density ( <i>xhdpi</i> ) screens (~320dpi).
	nodpi	Resources for all densities. These are density-independent resources. The system does not scale resources tagged with this qualifier, regardless of the current screen's density.
	tvdpi	Resources for screens somewhere between mdpi and hdpi; approximately 213dpi. This is not considered a "primary" density group. It is mostly intended for televisions and most apps shouldn't need it—providing mdpi and hdpi resources is sufficient for most apps and the system will scale them as appropriate. If you find it necessary to provide tvdpi resources, you should size them at a factor of 1.33*mdpi. For example, a 100px x 100px image for mdpi screens should be 133px x 133px for tvdpi.
Orientation	land	Resources for screens in the landscape orientation (wide aspect ratio).
	port	Resources for screens in the portrait orientation (tall aspect ratio).
Aspect ratio	long	Resources for screens that have a significantly taller or wider aspect ratio (when in portrait or landscape orientation, respectively) than the baseline screen configuration.
	notlong	Resources for use screens that have an aspect ratio that is similar to the baseline screen configuration.

**Note:** If you're developing your application for Android 3.2 and higher, see the section about [Declaring Tablet Layouts for Android 3.2](#) for information about new configuration qualifiers that you should use when declaring layout resources for specific screen sizes (instead of using the size qualifiers in table 1).

For more information about how these qualifiers roughly correspond to real screen sizes and densities, see [Range of Screens Supported](#), earlier in this document.

For example, the following is a list of resource directories in an application that provides different layout designs for different screen sizes and different bitmap drawables for medium, high, and extra high density screens.

```
res/layout/my_layout.xml                      // layout for normal screen size ("default")
res/layout-small/my_layout.xml                 // layout for small screen size
res/layout-large/my_layout.xml                // layout for large screen size
res/layout-xlarge/my_layout.xml               // layout for extra large screen size
res/layout-xlarge-land/my_layout.xml          // layout for extra large in landscape orientation

res/drawable-mdpi/my_icon.png                // bitmap for medium density
res/drawable-hdpi/my_icon.png                // bitmap for high density
res/drawable-xhdpi/my_icon.png               // bitmap for extra high density
```

For more information about how to use alternative resources and a complete list of configuration qualifiers (not just for screen configurations), see [Providing Alternative Resources](#).

Be aware that, when the Android system picks which resources to use at runtime, it uses certain logic to determine the "best matching" resources. That is, the qualifiers you use don't have to exactly match the current screen configuration in all cases in order for the system to use them. Specifically, when selecting resources based on the size qualifiers, the system will use resources designed for a screen smaller than the current screen if there are no resources that better match (for example, a large-size screen will use normal-size screen resources if necessary). However, if the only available resources are *larger* than the current screen, the system will not use them and your application will crash if no other resources match the device configuration (for example, if all layout resources are tagged with the *xlarge* qualifier, but the device is a normal-size screen). For more information about how the system selects resources, read [How Android Finds the Best-matching Resource](#).

**Tip:** If you have some drawable resources that the system should never scale (perhaps because you perform some adjustments to the image yourself at runtime), you should place them in a directory with the *nodpi* con-

figuration qualifier. Resources with this qualifier are considered density-agnostic and the system will not scale them.

## Designing alternative layouts and drawables

The types of alternative resources you should create depends on your application's needs. Usually, you should use the size and orientation qualifiers to provide alternative layout resources and use the density qualifiers to provide alternative bitmap drawable resources.

The following sections summarize how you might want to use the size and density qualifiers to provide alternative layouts and drawables, respectively.

### Alternative layouts

Generally, you'll know whether you need alternative layouts for different screen sizes once you test your application on different screen configurations. For example:

- When testing on a small screen, you might discover that your layout doesn't quite fit on the screen. For example, a row of buttons might not fit within the width of the screen on a small screen device. In this case you should provide an alternative layout for small screens that adjusts the size or position of the buttons.
- When testing on an extra large screen, you might realize that your layout doesn't make efficient use of the big screen and is obviously stretched to fill it. In this case, you should provide an alternative layout for extra large screens that provides a redesigned UI that is optimized for bigger screens such as tablets.

Although your application should work fine without an alternative layout on big screens, it's quite important to users that your application looks as though it's designed specifically for their devices. If the UI is obviously stretched, users are more likely to be unsatisfied with the application experience.

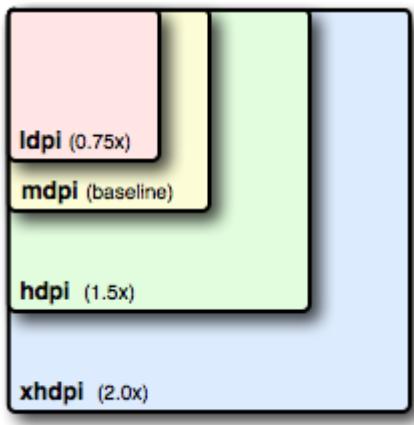
- And, when testing in the landscape orientation compared to the portrait orientation, you might notice that UI elements placed at the bottom of the screen for the portrait orientation should instead be on the right side of the screen in landscape orientation.

To summarize, you should be sure that your application layout:

- Fits on small screens (so users can actually use your application)
- Is optimized for bigger screens to take advantage of the additional screen space
- Is optimized for both landscape and portrait orientations

If your UI uses bitmaps that need to fit the size of a view even after the system scales the layout (such as the background image for a button), you should use [Nine-Patch](#) bitmap files. A Nine-Patch file is basically a PNG file in which you specific two-dimensional regions that are stretchable. When the system needs to scale the view in which the bitmap is used, the system stretches the Nine-Patch bitmap, but stretches only the specified regions. As such, you don't need to provide different drawables for different screen sizes, because the Nine-Patch bitmap can adjust to any size. You should, however, provide alternate versions of your Nine-Patch files for different screen densities.

## Alternative drawables



**Figure 4.** Relative sizes for bitmap drawables that support each density.

Almost every application should have alternative drawable resources for different screen densities, because almost every application has a launcher icon and that icon should look good on all screen densities. Likewise, if you include other bitmap drawables in your application (such as for menu icons or other graphics in your application), you should provide alternative versions of each one, for different densities.

**Note:** You only need to provide density-specific drawables for bitmap files (`.png`, `.jpg`, or `.gif`) and Nine-Path files (`.9.png`). If you use XML files to define shapes, colors, or other [drawable resources](#), you should put one copy in the default drawable directory (`drawable/`).

To create alternative bitmap drawables for different densities, you should follow the **3:4:6:8 scaling ratio** between the four generalized densities. For example, if you have a bitmap drawable that's 48x48 pixels for medium-density screen (the size for a launcher icon), all the different sizes should be:

- 36x36 for low-density
- 48x48 for medium-density
- 72x72 for high-density
- 96x96 for extra high-density

For more information about designing icons, see the [Icon Design Guidelines](#), which includes size information for various bitmap drawables, such as launcher icons, menu icons, status bar icons, tab icons, and more.

## Declaring Tablet Layouts for Android 3.2

For the first generation of tablets running Android 3.0, the proper way to declare tablet layouts was to put them in a directory with the `xlarge` configuration qualifier (for example, `res/layout-xlarge/`). In order to accommodate other types of tablets and screen sizes—in particular, 7" tablets—Android 3.2 introduces a new way to specify resources for more discrete screen sizes. The new technique is based on the amount of space your layout needs (such as 600dp of width), rather than trying to make your layout fit the generalized size groups (such as `large` or `xlarge`).

The reason designing for 7" tablets is tricky when using the generalized size groups is that a 7" tablet is technically in the same group as a 5" handset (the `large` group). While these two devices are seemingly close to each other in size, the amount of space for an application's UI is significantly different, as is the style of user interaction. Thus, a 7" and 5" screen should not always use the same layout. To make it possible for you to provide different layouts for these two kinds of screens, Android now allows you to specify your layout resources based on the width and/or height that's actually available for your application's layout, specified in dp units.

For example, after you've designed the layout you want to use for tablet-style devices, you might determine that the layout stops working well when the screen is less than 600dp wide. This threshold thus becomes the minimum size that you require for your tablet layout. As such, you can now specify that these layout resources should be used only when there is at least 600dp of width available for your application's UI.

You should either pick a width and design to it as your minimum size, or test what is the smallest width your layout supports once it's complete.

**Note:** Remember that all the figures used with these new size APIs are density-independent pixel (dp) values and your layout dimensions should also always be defined using dp units, because what you care about is the amount of screen space available after the system accounts for screen density (as opposed to using raw pixel resolution). For more information about density-independent pixels, read [Terms and concepts](#), earlier in this document.

## Using new size qualifiers

The different resource configurations that you can specify based on the space available for your layout are summarized in table 2. These new qualifiers offer you more control over the specific screen sizes your application supports, compared to the traditional screen size groups (small, normal, large, and xlarge).

**Note:** The sizes that you specify using these qualifiers are **not the actual screen sizes**. Rather, the sizes are for the width or height in dp units that are **available to your activity's window**. The Android system might use some of the screen for system UI (such as the system bar at the bottom of the screen or the status bar at the top), so some of the screen might not be available for your layout. Thus, the sizes you declare should be specifically about the sizes needed by your activity—the system accounts for any space used by system UI when declaring how much space it provides for your layout. Also beware that the [Action Bar](#) is considered a part of your application's window space, although your layout does not declare it, so it reduces the space available for your layout and you must account for it in your design.

**Table 2.** New configuration qualifiers for screen size (introduced in Android 3.2).

Screen configuration	Qualifier values	Description
smallestWidth Examples:	sw<N>dp sw600dp sw720dp	The fundamental size of a screen, as indicated by the shortest dimension of the available screen area. Specifically, the device's smallestWidth is the shortest of the screen's available height and width (you may also think of it as the "smallest possible width" for the screen). You can use this qualifier to ensure that, regardless of the screen's current orientation, your application's has at least <N> dps of width available for its UI.  For example, if your layout requires that its smallest dimension of screen area be at least 600 dp at all times, then you can use this qualifier to create the layout resources, res/layout-sw600dp/. The system will use these resources only when the smallest dimension of available screen is at least 600dp, regardless of whether the 600dp side is the user-perceived height or width. The smallestWidth is a fixed screen size characteristic of the device; <b>the device's smallestWidth does not change when the screen's orientation changes</b> .
		The smallestWidth of a device takes into account screen decorations and system UI. For example, if the device has some persistent UI elements on the screen that account for space along the axis of the smallestWidth, the system declares the smallestWidth

to be smaller than the actual screen size, because those are screen pixels not available for your UI.

This is an alternative to the generalized screen size qualifiers (small, normal, large, xlarge) that allows you to define a discrete number for the effective size available for your UI. Using `smallestWidth` to determine the general screen size is useful because width is often the driving factor in designing a layout. A UI will often scroll vertically, but have fairly hard constraints on the minimum space it needs horizontally. The available width is also the key factor in determining whether to use a one-pane layout for handsets or multi-pane layout for tablets. Thus, you likely care most about what the smallest possible width will be on each device.

		Specifies a minimum available width in dp units at which the resources should be used—defined by the <code>&lt;N&gt;</code> value. The system's corresponding value for the width changes when the screen's orientation switches between landscape and portrait to reflect the current actual width that's available for your UI.
Available screen width	w<N>dp Examples: w720dp w1024dp	This is often useful to determine whether to use a multi-pane layout, because even on a tablet device, you often won't want the same multi-pane layout for portrait orientation as you do for landscape. Thus, you can use this to specify the minimum width required for the layout, instead of using both the screen size and orientation qualifiers together.
Available screen height	h<N>dp Examples: h720dp h1024dp etc.	Specifies a minimum screen height in dp units at which the resources should be used—defined by the <code>&lt;N&gt;</code> value. The system's corresponding value for the height changes when the screen's orientation switches between landscape and portrait to reflect the current actual height that's available for your UI.

While using these qualifiers might seem more complicated than using screen size groups, it should actually be simpler once you determine the requirements for your UI. When you design your UI, the main thing you probably care about is the actual size at which your application switches between a handset-style UI and a tablet-style UI that uses multiple panes. The exact point of this switch will depend on your particular design—maybe you need a 720dp width for your tablet layout, maybe 600dp is enough, or 480dp, or some number between these. Using these qualifiers in table 2, you are in control of the precise size at which your layout changes.

For more discussion about these size configuration qualifiers, see the [Providing Resources](#) document.

## Configuration examples

To help you target some of your designs for different types of devices, here are some numbers for typical screen widths:

- 320dp: a typical phone screen (240x320 ldpi, 320x480 mdpi, 480x800 hdpi, etc).
- 480dp: a tweener tablet like the Streak (480x800 mdpi).

- 600dp: a 7" tablet (600x1024 mdpi).
- 720dp: a 10" tablet (720x1280 mdpi, 800x1280 mdpi, etc).

Using the size qualifiers from table 2, your application can switch between your different layout resources for handsets and tablets using any number you want for width and/or height. For example, if 600dp is the smallest available width supported by your tablet layout, you can provide these two sets of layouts:

```
res/layout/main_activity.xml          # For handsets
res/layout-sw600dp/main_activity.xml # For tablets
```

In this case, the smallest width of the available screen space must be 600dp in order for the tablet layout to be applied.

For other cases in which you want to further customize your UI to differentiate between sizes such as 7" and 10" tablets, you can define additional smallest width layouts:

```
res/layout/main_activity.xml          # For handsets (smaller than 600dp available)
res/layout-sw600dp/main_activity.xml  # For 7" tablets (600dp wide and bigger)
res/layout-sw720dp/main_activity.xml  # For 10" tablets (720dp wide and bigger)
```

Notice that the previous two sets of example resources use the "smallest width" qualifer, `sw<N>dp`, which specifies the smallest of the screen's two sides, regardless of the device's current orientation. Thus, using `sw<N>dp` is a simple way to specify the overall screen size available for your layout by ignoring the screen's orientation.

However, in some cases, what might be important for your layout is exactly how much width or height is *currently* available. For example, if you have a two-pane layout with two fragments side by side, you might want to use it whenever the screen provides at least 600dp of width, whether the device is in landscape or portrait orientation. In this case, your resources might look like this:

```
res/layout/main_activity.xml          # For handsets (smaller than 600dp available)
res/layout-w600dp/main_activity.xml   # Multi-pane (any screen with 600dp available)
```

Notice that the second set is using the "available width" qualifier, `w<N>dp`. This way, one device may actually use both layouts, depending on the orientation of the screen (if the available width is at least 600dp in one orientation and less than 600dp in the other orientation).

If the available height is a concern for you, then you can do the same using the `h<N>dp` qualifier. Or, even combine the `w<N>dp` and `h<N>dp` qualifiers if you need to be really specific.

## Declaring screen size support

Once you've implemented your layouts for different screen sizes, it's equally important that you declare in your manifest file which screens your application supports.

Along with the new configuration qualifiers for screen size, Android 3.2 introduces new attributes for the [`<supports-screens>`](#) manifest element:

### [`android:requiresSmallestWidthDp`](#)

Specifies the minimum `smallestWidth` required. The `smallestWidth` is the shortest dimension of the screen space (in dp units) that must be available to your application UI—that is, the shortest of the available screen's two dimensions. So, in order for a device to be considered compatible with your application, the device's `smallestWidth` must be equal to or greater than this value. (Usually, the value you supply for this is the "smallest width" that your layout supports, regardless of the screen's current orientation.)

For example, if your application is only for tablet-style devices with a 600dp smallest available width:

```
<manifest ... >
    <supports-screens android:requiresSmallestWidthDp="600" />
    ...
</manifest>
```

However, if your application supports all screen sizes supported by Android (as small as 426dp x 320dp), then you don't need to declare this attribute, because the smallest width your application requires is the smallest possible on any device.

**Caution:** The Android system does not pay attention to this attribute, so it does not affect how your application behaves at runtime. Instead, it is used to enable filtering for your application on services such as Google Play. However, **Google Play currently does not support this attribute for filtering** (on Android 3.2), so you should continue using the other size attributes if your application does not support small screens.

#### [android:compatibleWidthLimitDp](#)

This attribute allows you to enable [screen compatibility mode](#) as a user-optional feature by specifying the maximum "smallest width" that your application supports. If the smallest side of a device's available screen is greater than your value here, users can still install your application, but are offered to run it in screen compatibility mode. By default, screen compatibility mode is disabled and your layout is resized to fit the screen as usual, but a button is available in the system bar that allows users to toggle screen compatibility mode on and off.

**Note:** If your application's layout properly resizes for large screens, you do not need to use this attribute. We recommend that you avoid using this attribute and instead ensure your layout resizes for larger screens by following the recommendations in this document.

#### [android:largestWidthLimitDp](#)

This attribute allows you to force-enable [screen compatibility mode](#) by specifying the maximum "smallest width" that your application supports. If the smallest side of a device's available screen is greater than your value here, the application runs in screen compatibility mode with no way for the user to disable it.

**Note:** If your application's layout properly resizes for large screens, you do not need to use this attribute. We recommend that you avoid using this attribute and instead ensure your layout resizes for larger screens by following the recommendations in this document.

**Caution:** When developing for Android 3.2 and higher, you should not use the older screen size attributes in combination with the attributes listed above. Using both the new attributes and the older size attributes might cause unexpected behavior.

For more information about each of these attributes, follow the respective links above.

## Best Practices

The objective of supporting multiple screens is to create an application that can function properly and look good on any of the generalized screen configurations supported by Android. The previous sections of this document provide information about how Android adapts your application to screen configurations and how you can customize the look of your application on different screen configurations. This section provides some additional tips and an overview of techniques that help ensure that your application scales properly for different screen configurations.

Here is a quick checklist about how you can ensure that your application displays properly on different screens:

1. Use `wrap_content`, `fill_parent`, or `dp` units when specifying dimensions in an XML layout file
2. Do not use hard coded pixel values in your application code
3. Do not use `AbsoluteLayout` (it's deprecated)
4. Supply alternative bitmap drawables for different screen densities

The following sections provide more details.

## 1. Use `wrap_content`, `fill_parent`, or the `dp` unit for layout dimensions

When defining the `android:layout_width` and `android:layout_height` for views in an XML layout file, using "`wrap_content`", "`fill_parent`" or `dp` units guarantees that the view is given an appropriate size on the current device screen.

For instance, a view with a `layout_width="100dp"` measures 100 pixels wide on medium-density screen and the system scales it up to 150 pixels wide on high-density screen, so that the view occupies approximately the same physical space on the screen.

Similarly, you should prefer the `sp` (scale-independent pixel) to define text sizes. The `sp` scale factor depends on a user setting and the system scales the size the same as it does for `dp`.

## 2. Do not use hard-coded pixel values in your application code

For performance reasons and to keep the code simpler, the Android system uses pixels as the standard unit for expressing dimension or coordinate values. That means that the dimensions of a view are always expressed in the code using pixels, but always based on the current screen density. For instance, if `myView.getWidth()` returns 10, the view is 10 pixels wide on the current screen, but on a device with a higher density screen, the value returned might be 15. If you use pixel values in your application code to work with bitmaps that are not pre-scaled for the current screen density, you might need to scale the pixel values that you use in your code to match the un-scaled bitmap source.

If your application manipulates bitmaps or deals with pixel values at runtime, see the section below about [Additional Density Considerations](#).

## 3. Do not use `AbsoluteLayout`

Unlike the other layouts widgets, `AbsoluteLayout` enforces the use of fixed positions to lay out its child views, which can easily lead to user interfaces that do not work well on different displays. Because of this, [AbsoluteLayout](#) was deprecated in Android 1.5 (API Level 3).

You should instead use `RelativeLayout`, which uses relative positioning to lay out its child views. For instance, you can specify that a button widget should appear "to the right of" a text widget.

## 4. Use size and density-specific resources

Although the system scales your layout and drawable resources based on the current screen configuration, you may want to make adjustments to the UI on different screen sizes and provide bitmap drawables that are optimized for different densities. This essentially reiterates the information from earlier in this document.

If you need to control exactly how your application will look on various screen configurations, adjust your layouts and bitmap drawables in configuration-specific resource directories. For example, consider an icon that you want to display on medium and high density screens. Simply create your icon at two different sizes (for in-

stance 100x100 for medium density and 150x150 for high density) and put the two variations in the appropriate directories, using the proper qualifiers:

```
res/drawable-mdpi/icon.png      //for medium-density screens  
res/drawable-hdpi/icon.png     //for high-density screens
```

**Note:** If a density qualifier is not defined in a directory name, the system assumes that the resources in that directory are designed for the baseline medium density and will scale for other densities as appropriate.

For more information about valid configuration qualifiers, see [Using configuration qualifiers](#), earlier in this document.

## Additional Density Considerations

This section describes more about how Android performs scaling for bitmap drawables on different screen densities and how you can further control how bitmaps are drawn on different densities. The information in this section shouldn't be important to most applications, unless you have encountered problems in your application when running on different screen densities or your application manipulates graphics.

To better understand how you can support multiple densities when manipulating graphics at runtime, you should understand that the system helps ensure the proper scale for bitmaps in the following ways:

### 1. *Pre-scaling of resources (such as bitmap drawables)*

Based on the density of the current screen, the system uses any size- or density-specific resources from your application and displays them without scaling. If resources are not available in the correct density, the system loads the default resources and scales them up or down as needed to match the current screen's density. The system assumes that default resources (those from a directory without configuration qualifiers) are designed for the baseline screen density (mdpi), unless they are loaded from a density-specific resource directory. Pre-scaling is, thus, what the system does when resizing a bitmap to the appropriate size for the current screen density.

If you request the dimensions of a pre-scaled resource, the system returns values representing the dimensions *after* scaling. For example, a bitmap designed at 50x50 pixels for an mdpi screen is scaled to 75x75 pixels on an hdpi screen (if there is no alternative resource for hdpi) and the system reports the size as such.

There are some situations in which you might not want Android to pre-scale a resource. The easiest way to avoid pre-scaling is to put the resource in a resource directory with the `nodpi` configuration qualifier. For example:

```
res/drawable-nodpi/icon.png
```

When the system uses the `icon.png` bitmap from this folder, it does not scale it based on the current device density.

### 2. *Auto-scaling of pixel dimensions and coordinates*

An application can disable pre-scaling by setting `android:anyDensity` to "false" in the manifest or programmatically for a [Bitmap](#) by setting `inScaled` to "false". In this case, the system auto-scales any absolute pixel coordinates and pixel dimension values at draw time. It does this to ensure that pixel-defined screen elements are still displayed at approximately the same physical size as they would be at the baseline screen density (mdpi). The system handles this scaling transparently to

the application and reports the scaled pixel dimensions to the application, rather than physical pixel dimensions.

For instance, suppose a device has a WVGA high-density screen, which is 480x800 and about the same size as a traditional HVGA screen, but it's running an application that has disabled pre-scaling. In this case, the system will "lie" to the application when it queries for screen dimensions, and report 320x533 (the approximate mdpi translation for the screen density). Then, when the application does drawing operations, such as invalidating the rectangle from (10,10) to (100, 100), the system transforms the coordinates by scaling them the appropriate amount, and actually invalidate the region (15,15) to (150, 150). This discrepancy may cause unexpected behavior if your application directly manipulates the scaled bitmap, but this is considered a reasonable trade-off to keep the performance of applications as good as possible. If you encounter this situation, read the following section about [Converting dp units to pixel units](#).

Usually, **you should not disable pre-scaling**. The best way to support multiple screens is to follow the basic techniques described above in [How to Support Multiple Screens](#).

If your application manipulates bitmaps or directly interacts with pixels on the screen in some other way, you might need to take additional steps to support different screen densities. For example, if you respond to touch gestures by counting the number of pixels that a finger crosses, you need to use the appropriate density-independent pixel values, instead of actual pixels.

## Scaling Bitmap objects created at runtime



**Figure 5.** Comparison of pre-scaled and auto-scaled bitmaps, from [ApiDemos](#).

If your application creates an in-memory bitmap (a [Bitmap](#) object), the system assumes that the bitmap is designed for the baseline medium-density screen, by default, and auto-scales the bitmap at draw time. The system applies "auto-scaling" to a [Bitmap](#) when the bitmap has unspecified density properties. If you don't properly account for the current device's screen density and specify the bitmap's density properties, the auto-scaling can result in scaling artifacts the same as when you don't provide alternative resources.

To control whether a [Bitmap](#) created at runtime is scaled or not, you can specify the density of the bitmap with [setDensity\(\)](#), passing a density constant from [DisplayMetrics](#), such as [DENSITY\\_HIGH](#) or [DENSITY\\_LOW](#).

If you're creating a [Bitmap](#) using [BitmapFactory](#), such as from a file or a stream, you can use [BitmapFactory.Options](#) to define properties of the bitmap as it already exists, which determine if or how the system will scale it. For example, you can use the [inDensity](#) field to define the density for which the bitmap is designed and the [inScaled](#) field to specify whether the bitmap should scale to match the current device's screen density.

If you set the [inScaled](#) field to `false`, then you disable any pre-scaling that the system may apply to the bitmap and the system will then auto-scale it at draw time. Using auto-scaling instead of pre-scaling can be more CPU expensive, but uses less memory.

Figure 5 demonstrates the results of the pre-scale and auto-scale mechanisms when loading low (120), medium (160) and high (240) density bitmaps on a high-density screen. The differences are subtle, because all of the bitmaps are being scaled to match the current screen density, however the scaled bitmaps have slightly different appearances depending on whether they are pre-scaled or auto-scaled at draw time. You can find the source code for this sample application, which demonstrates using pre-scaled and auto-scaled bitmaps, in [ApiDemos](#).

**Note:** In Android 3.0 and above, there should be no perceivable difference between pre-scaled and auto-scaled bitmaps, due to improvements in the graphics framework.

## Converting dp units to pixel units

In some cases, you will need to express dimensions in dp and then convert them to pixels. Imagine an application in which a scroll or fling gesture is recognized after the user's finger has moved by at least 16 pixels. On a baseline screen, a user's must move by 16 pixels / 160 dpi, which equals 1/10th of an inch (or 2.5 mm) before the gesture is recognized. On a device with a high density display (240dpi), the user's must move by 16 pixels / 240 dpi, which equals 1/15th of an inch (or 1.7 mm). The distance is much shorter and the application thus appears more sensitive to the user.

To fix this issue, the gesture threshold must be expressed in code in dp and then converted to actual pixels. For example:

```
// The gesture threshold expressed in dp
private static final float GESTURE_THRESHOLD_DP = 16.0f;

// Get the screen's density scale
final float scale = getResources().getDisplayMetrics().density;
// Convert the dps to pixels, based on density scale
mGestureThreshold = (int) (GESTURE_THRESHOLD_DP * scale + 0.5f);

// Use mGestureThreshold as a distance in pixels...
```

The [DisplayMetrics.density](#) field specifies the scale factor you must use to convert dp units to pixels, according to the current screen density. On a medium-density screen, [DisplayMetrics.density](#) equals

1.0; on a high-density screen it equals 1.5; on an extra high-density screen, it equals 2.0; and on a low-density screen, it equals 0.75. This figure is the factor by which you should multiply the dp units on order to get the actual pixel count for the current screen. (Then add 0.5f to round the figure up to the nearest whole number, when converting to an integer.) For more information, refer to the [DisplayMetrics](#) class.

However, instead of defining an arbitrary threshold for this kind of event, you should use pre-scaled configuration values that are available from [ViewConfiguration](#).

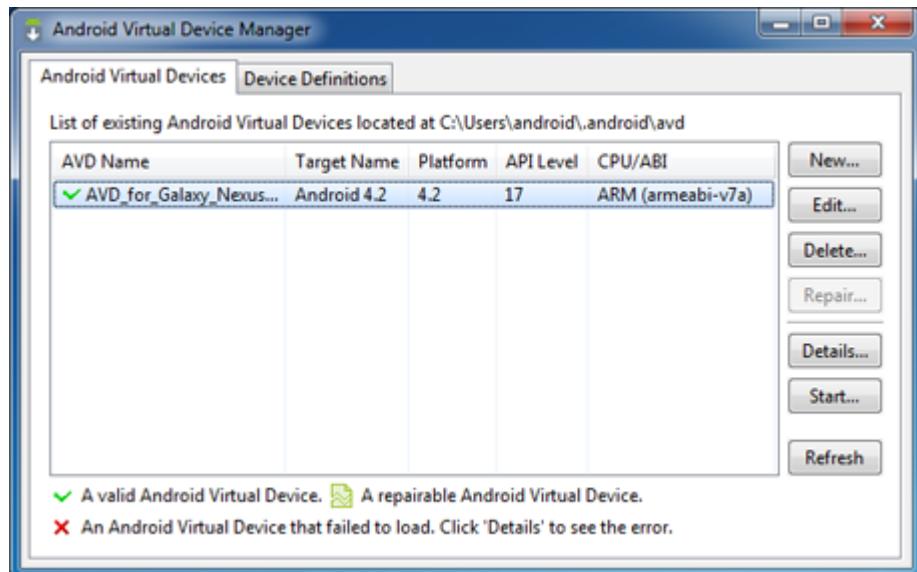
## Using pre-scaled configuration values

You can use the [ViewConfiguration](#) class to access common distances, speeds, and times used by the Android system. For instance, the distance in pixels used by the framework as the scroll threshold can be obtained with [getScaledTouchSlop\(\)](#):

```
private static final int GESTURE_THRESHOLD_DP = ViewConfiguration.get(myContext)
```

Methods in [ViewConfiguration](#) starting with the `getScaled` prefix are guaranteed to return a value in pixels that will display properly regardless of the current screen density.

## How to Test Your Application on Multiple Screens



**Figure 6.** A set of AVDs for testing screens support.

Before publishing your application, you should thoroughly test it in all of the supported screen sizes and densities. The Android SDK includes emulator skins you can use, which replicate the sizes and densities of common screen configurations on which your application is likely to run. You can also modify the default size, density, and resolution of the emulator skins to replicate the characteristics of any specific screen. Using the emulator skins and additional custom configurations allows you to test any possible screen configuration, so you don't have to buy various devices just to test your application's screen support.

To set up an environment for testing your application's screen support, you should create a series of AVDs (Android Virtual Devices), using emulator skins and screen configurations that emulate the screen sizes and densities you want your application to support. To do so, you can use the AVD Manager to create the AVDs and launch them with a graphical interface.

To launch the Android SDK Manager, execute the `SDK Manager.exe` from your Android SDK directory (on Windows only) or execute `android` from the `<sdk>/tools/` directory (on all platforms). Figure 6 shows the AVD Manager with a selection of AVDs, for testing various screen configurations.

Table 3 shows the various emulator skins that are available in the Android SDK, which you can use to emulate some of the most common screen configurations.

For more information about creating and using AVDs to test your application, see [Managing AVDs with AVD Manager](#).

**Table 3.** Various screen configurations available from emulator skins in the Android SDK (indicated in bold) and other representative resolutions.

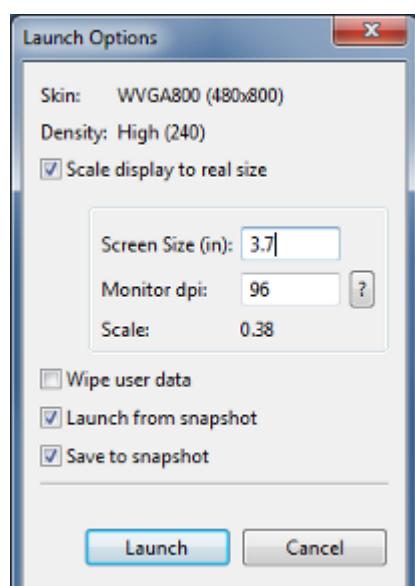
	<b>Low density (120), <i>ldpi</i></b>	<b>Medium density (160), <i>mdpi</i></b>	<b>High density (240), <i>hdpi</i></b>	<b>Extra high density (320), <i>xhdpi</i></b>
<b>Small screen</b>	<b>QVGA (240x320)</b>		480x640	
	<b>WQVGA400 (240x400)</b>		<b>WVGA800 (480x800)</b>	
<b>Normal screen</b>	<b>WQVGA432 (240x432)</b>	<b>HVGA (320x480)</b>	<b>WVGA854 (480x854)</b>	640x960
			600x1024	
	<b>WVGA800** (480x800)</b>	<b>WVGA800* (480x800)</b>		
<b>Large screen</b>	<b>WVGA854** (480x854)</b>	<b>WVGA854* (480x854)</b>		
		600x1024		
<b>Extra Large screen</b>	1024x600	<b>WXGA (1280x800)<sup>†</sup></b>	1536x1152	2048x1536
		1024x768	1920x1152	2560x1536
		1280x768	1920x1200	2560x1600

\* To emulate this configuration, specify a custom density of 160 when creating an AVD that uses a WVGA800 or WVGA854 skin.

\*\* To emulate this configuration, specify a custom density of 120 when creating an AVD that uses a WVGA800 or WVGA854 skin.

† This skin is available with the Android 3.0 platform

To see the relative numbers of active devices that support any given screen configuration, see the [Screen Sizes and Densities](#) dashboard.



**Figure 7.** Size and density options you can set, when starting an AVD from the AVD Manager.

We also recommend that you test your application in an emulator that is set up to run at a physical size that closely matches an actual device. This makes it a lot easier to compare the results at various sizes and densities. To do so you need to know the approximate density, in dpi, of your computer monitor (for instance, a 30" Dell monitor has a density of about 96 dpi). When you launch an AVD from the AVD Manager, you can specify the screen size for the emulator and your monitor dpi in the Launch Options, as shown in figure 7.

If you would like to test your application on a screen that uses a resolution or density not supported by the built-in skins, you can create an AVD that uses a custom resolution or density. When creating the AVD from the AVD Manager, specify the Resolution, instead of selecting a Built-in Skin.

If you are launching your AVD from the command line, you can specify the scale for the emulator with the `-scale` option. For example:

```
emulator -avd <avd_name> -scale 96dpi
```

To refine the size of the emulator, you can instead pass the `-scale` option a number between 0.1 and 3 that represents the desired scaling factor.

For more information about creating AVDs from the command line, see [Managing AVDs from the Command Line](#)

# <uses-configuration>

syntax:

```
<uses-configuration
    android:reqFiveWayNav=["true" | "false"]
    android:reqHardKeyboard=["true" | "false"]
    android:reqKeyboardType=["undefined" | "nokeys" | "qwerty" | "twelvekey"]
    android:reqNavigation=["undefined" | "nonav" | "dpad" | "trackball" | "wheel"]
    android:reqTouchScreen=["undefined" | "notouch" | "stylus" | "finger"] />
```

contained in:

[manifest](#)

description:

Indicates what hardware and software features the application requires. For example, an application might specify that it requires a physical keyboard or a particular navigation device, like a trackball. The specification is used to avoid installing the application on devices where it will not work.

If an application can work with different device configurations, it should include separate <uses-configuration> declarations for each one. Each declaration must be complete. For example, if an application requires a five-way navigation control, a touch screen that can be operated with a finger, and either a standard QWERTY keyboard or a numeric 12-key keypad like those found on most phones, it would specify these requirements with two <uses-configuration> elements as follows:

```
<uses-configuration android:reqFiveWayNav="true" android:reqTouchScreen="finger"
                    android:reqKeyboardType="qwerty" />
<uses-configuration android:reqFiveWayNav="true" android:reqTouchScreen="fingerprint"
                    android:reqKeyboardType="twelvekey" />
```

attributes:

**android:reqFiveWayNav**

Whether or not the application requires a five-way navigation control — "true" if it does, and "false" if not. A five-way control is one that can move the selection up, down, right, or left, and also provides a way of invoking the current selection. It could be a D-pad (directional pad), trackball, or other device.

If an application requires a directional control, but not a control of a particular type, it can set this attribute to "true" and ignore the [reqNavigation](#) attribute. However, if it requires a particular type of directional control, it can ignore this attribute and set [reqNavigation](#) instead.

**android:reqHardKeyboard**

Whether or not the application requires a hardware keyboard — "true" if it does, and "false" if not.

**android:reqKeyboardType**

The type of keyboard the application requires, if any at all. This attribute does not distinguish between hardware and software keyboards. If a hardware keyboard of a certain type is required, specify the type here and also set the [reqHardKeyboard](#) attribute to "true".

The value must be one of the following strings:

**Value**

**Description**

" <code>undefined</code> "	The application does not require a keyboard. (A keyboard requirement is not defined.) This is the default value.
" <code>nokeys</code> "	The application does not require a keyboard.
" <code>qwerty</code> "	The application requires a standard QWERTY keyboard.
" <code>twelvekey</code> "	The application requires a twelve-key keypad, like those on most phones — with keys for the digits from 0 through 9 plus star (*) and pound (#) keys.

#### **android:reqNavigation**

The navigation device required by the application, if any. The value must be one of the following strings:

<b>Value</b>	<b>Description</b>
" <code>undefined</code> "	The application does not require any type of navigation control. (The navigation requirement is not defined.) This is the default value.
" <code>nonav</code> "	The application does not require a navigation control.
" <code>dpad</code> "	The application requires a D-pad (directional pad) for navigation.
" <code>trackball</code> "	The application requires a trackball for navigation.
" <code>wheel</code> "	The application requires a navigation wheel.

If an application requires a navigational control, but the exact type of control doesn't matter, it can set the [reqFiveWayNav](#) attribute to "`true`" rather than set this one.

#### **android:reqTouchScreen**

The type of touch screen the application requires, if any at all. The value must be one of the following strings:

<b>Value</b>	<b>Description</b>
" <code>undefined</code> "	The application doesn't require a touch screen. (The touch screen requirement is undefined.) This is the default value.
" <code>notouch</code> "	The application doesn't require a touch screen.
" <code>stylus</code> "	The application requires a touch screen that's operated with a stylus.
" <code>finger</code> "	The application requires a touch screen that can be operated with a finger.

#### **introduced in:**

API Level 3

#### **see also:**

- [configChanges](#) attribute of the [<activity>](#) element

- [ConfigurationInfo](#)

# <uses-feature>

## In this document

1. [Google Play and Feature-Based Filtering](#)
  1. [Filtering based on explicitly declared features](#)
  2. [Filtering based on implicit features](#)
  3. [Special handling for Bluetooth feature](#)
  4. [Testing the features required by your application](#)
2. [Features Reference](#)
  1. [Hardware features](#)
  2. [Software features](#)
  3. [Permissions that Imply Feature Requirements](#)



### Google Play Filtering

Google Play uses the <uses-feature> elements declared in your app manifest to filter your app from devices that do not meet its hardware and software feature requirements.

By specifying the features that your application requires, you enable Google Play to present your application only to users whose devices meet the application's feature requirements, rather than presenting it to all users.

For important information about how Google Play uses features as the basis for filtering, please read [Google Play and Feature-Based Filtering](#), below.

#### syntax:

```
<uses-feature  
    android:name="string"  
    android:required=["true" | "false"]  
    android:glEsVersion="integer" />
```

#### contained in:

[<manifest>](#)

#### description:

Declares a single hardware or software feature that is used by the application.

The purpose of a <uses-feature> declaration is to inform any external entity of the set of hardware and software features on which your application depends. The element offers a `required` attribute that lets you specify whether your application requires and cannot function without the declared feature, or whether it prefers to have the feature but can function without it. Because feature support can vary across Android devices, the <uses-feature> element serves an important role in letting an application describe the device-variable features that it uses.

The set of available features that your application declares corresponds to the set of feature constants made available by the Android [PackageManager](#), which are listed for convenience in the [Features Reference](#) tables at the bottom of this document.

You must specify each feature in a separate `<uses-feature>` element, so if your application requires multiple features, it would declare multiple `<uses-feature>` elements. For example, an application that requires both Bluetooth and camera features in the device would declare these two elements:

```
<uses-feature android:name="android.hardware.bluetooth" />
<uses-feature android:name="android.hardware.camera" />
```

In general, you should always make sure to declare `<uses-feature>` elements for all of the features that your application requires.

Declared `<uses-feature>` elements are informational only, meaning that the Android system itself does not check for matching feature support on the device before installing an application. However, other services (such as Google Play) or applications may check your application's `<uses-feature>` declarations as part of handling or interacting with your application. For this reason, it's very important that you declare all of the features (from the list below) that your application uses.

For some features, there may exist a specific attribute that allows you to define a version of the feature, such as the version of Open GL used (declared with [glEsVersion](#)). Other features that either do or do not exist for a device, such as a camera, are declared using the [name](#) attribute.

Although the `<uses-feature>` element is only activated for devices running API Level 4 or higher, it is recommended to include these elements for all applications, even if the [minSdkVersion](#) is "3" or lower. Devices running older versions of the platform will simply ignore the element.

**Note:** When declaring a feature, remember that you must also request permissions as appropriate. For example, you must still request the [CAMERA](#) permission before your application can access the camera API. Requesting the permission grants your application access to the appropriate hardware and software, while declaring the features used by your application ensures proper device compatibility.

## attributes:

### **android:name**

Specifies a single hardware or software feature used by the application, as a descriptor string. Valid descriptor values are listed in the [Hardware features](#) and [Software features](#) tables, below.

### **android:required**

Boolean value that indicates whether the application requires the feature specified in `android:name`.

- When you declare `"android:required=true"` for a feature, you are specifying that the application *cannot function, or is not designed to function*, when the specified feature is not present on the device.
- When you declare `"android:required=false"` for a feature, it means that the application *prefers to use the feature* if present on the device, but that it *is designed to function without the specified feature*, if necessary.

The default value for `android:required` if not declared is `"true"`.

### **android:glEsVersion**

The OpenGL ES version required by the application. The higher 16 bits represent the major number and the lower 16 bits represent the minor number. For example, to specify OpenGL ES version 2.0, you would set the value as `"0x00020000"`. To specify OpenGL ES 2.1, if/when such a version were made available, you would set the value as `"0x00020001"`.

An application should specify at most one `android:glEsVersion` attribute in its manifest. If it specifies more than one, the `android:glEsVersion` with the numerically highest value is used and any other values are ignored.

If an application does not specify an `android:glEsVersion` attribute, then it is assumed that the application requires only OpenGL ES 1.0, which is supported by all Android-powered devices.

An application can assume that if a platform supports a given OpenGL ES version, it also supports all numerically lower OpenGL ES versions. Therefore, an application that requires both OpenGL ES 1.0 and OpenGL ES 2.0 must specify that it requires OpenGL ES 2.0.

An application that can work with any of several OpenGL ES versions should only specify the numerically lowest version of OpenGL ES that it requires. (It can check at run-time whether a higher level of OpenGL ES is available.)

**introduced in:**

API Level 4

**see also:**

- [PackageManager](#)
- [FeatureInfo](#)
- [ConfigurationInfo](#)
- [<uses-permission>](#)
- [Filters on Google Play](#)

## Google Play and Feature-Based Filtering

Google Play filters the applications that are visible to users, so that users can see and download only those applications that are compatible with their devices. One of the ways it filters applications is by feature compatibility.

To determine an application's feature compatibility with a given user's device, Google Play compares:

- Features required by the application — an application declares features in `<uses-feature>` elements in its manifest with...
- Features available on the device, in hardware or software — a device reports the features it supports as read-only system properties.

To ensure an accurate comparison of features, the Android Package Manager provides a shared set of feature constants that both applications and devices use to declare feature requirements and support. The available feature constants are listed in the [Features Reference](#) tables at the bottom of this document, and in the class documentation for [PackageManager](#).

When the user launches Google Play, the application queries the Package Manager for the list of features available on the device by calling `getSystemAvailableFeatures()`. The Store application then passes the features list up to Google Play when establishing the session for the user.

Each time you upload an application to the Google Play Developer Console, Google Play scans the application's manifest file. It looks for `<uses-feature>` elements and evaluates them in combination with other elements, in some cases, such as `<uses-sdk>` and `<uses-permission>` elements. After establishing the application's set of required features, it stores that list internally as metadata associated with the application .apk and the application version.

When a user searches or browses for applications using the Google Play application, the service compares the features needed by each application with the features available on the user's device. If all of an application's required features are present on the device, Google Play allows the user to see the application and potentially download it. If any required feature is not supported by the device, Google Play filters the application so that it is not visible to the user and not available for download.

Because the features you declare in `<uses-feature>` elements directly affect how Google Play filters your application, it's important to understand how Google Play evaluates the application's manifest and establishes the set of required features. The sections below provide more information.

## Filtering based on explicitly declared features

An explicitly declared feature is one that your application declares in a `<uses-feature>` element. The feature declaration can include an `android:required=["true" | "false"]` attribute (if you are compiling against API level 5 or higher), which lets you specify whether the application absolutely requires the feature and cannot function properly without it ("`true`"), or whether the application prefers to use the feature if available, but is designed to run without it ("`false`").

Google Play handles explicitly declared features in this way:

- If a feature is explicitly declared as being required, Google Play adds the feature to the list of required features for the application. It then filters the application from users on devices that do not provide that feature. For example:

```
<uses-feature android:name="android.hardware.camera" android:required="true"
```

- If a feature is explicitly declared as *not* being required, Google Play *does not* add the feature to the list of required features. For that reason, an explicitly declared non-required feature is never considered when filtering the application. Even if the device does not provide the declared feature, Google Play will still consider the application compatible with the device and will show it to the user, unless other filtering rules apply. For example:

```
<uses-feature android:name="android.hardware.camera" android:required="false"
```

- If a feature is explicitly declared, but without an `android:required` attribute, Google Play assumes that the feature is required and sets up filtering on it.

In general, if your application is designed to run on Android 1.6 and earlier versions, the `android:required` attribute is not available in the API and Google Play assumes that any and all `<uses-feature>` declarations are required.

**Note:** By declaring a feature explicitly and including an `android:required="false"` attribute, you can effectively disable all filtering on Google Play for the specified feature.

## Filtering based on implicit features

An *implicit* feature is one that an application requires in order to function properly, but which is *not* declared in a `<uses-feature>` element in the manifest file. Strictly speaking, every application should *always* declare all features that it uses or requires, so the absence of a declaration for a feature used by an application should be considered an error. However, as a safeguard for users and developers, Google Play looks for implicit features in each application and sets up filters for those features, just as it would do for an explicitly declared feature.

An application might require a feature but not declare it because:

- The application was compiled against an older version of the Android library (Android 1.5 or earlier) and the `<uses-feature>` element was not available.
- The developer incorrectly assumed that the feature would be present on all devices and a declaration was unnecessary.
- The developer omitted the feature declaration accidentally.
- The developer declared the feature explicitly, but the declaration was not valid. For example, a spelling error in the `<uses-feature>` element name or an unrecognized string value for the `android:name` attribute would invalidate the feature declaration.

To account for the cases above, Google Play attempts to discover an application's implied feature requirements by examining *other elements* declared in the manifest file, specifically, `<uses-permission>` elements.

If an application requests hardware-related permissions, Google Play *assumes that the application uses the underlying hardware features and therefore requires those features*, even though there might be no corresponding to `<uses-feature>` declarations. For such permissions, Google Play adds the underlying hardware features to the metadata that it stores for the application and sets up filters for them.

For example, if an application requests the `CAMERA` permission but does not declare a `<uses-feature>` element for `android.hardware.camera`, Google Play considers that the application requires a camera and should not be shown to users whose devices do not offer a camera.

If you don't want Google Play to filter based on a specific implied feature, you can disable that behavior. To do so, declare the feature explicitly in a `<uses-feature>` element and include an `android:required="false"` attribute. For example, to disable filtering derived from the `CAMERA` permission, you would declare the feature as shown below.

```
<uses-feature android:name="android.hardware.camera" android:required="false" /
```

It's important to understand that the permissions that you request in `<uses-permission>` elements can directly affect how Google Play filters your application. The reference section [Permissions that Imply Feature Requirements](#), below, lists the full set of permissions that imply feature requirements and therefore trigger filtering.

## Special handling for Bluetooth feature

Google Play applies slightly different rules than described above, when determining filtering for Bluetooth.

If an application declares a Bluetooth permission in a `<uses-permission>` element, but does not explicitly declare the Bluetooth feature in a `<uses-feature>` element, Google Play checks the version(s) of the Android platform on which the application is designed to run, as specified in the `<uses-sdk>` element.

As shown in the table below, Google Play enables filtering for the Bluetooth feature only if the application declares its lowest or targeted platform as Android 2.0 (API level 5) or higher. However, note that Google Play applies the normal rules for filtering when the application explicitly declares the Bluetooth feature in a `<uses-feature>` element.

**Table 1.** How Google Play determines the Bluetooth feature requirement for an application that requests a Bluetooth permission but does not declare the Bluetooth feature in a `<uses-feature>` element.

If <code>minSdkVersion</code> or <code>targetSdkVersion</code> is ...	<code>Result</code>
... 5 or higher	Google Play filters for the Bluetooth feature.

<code>&lt;=4 (or uses-sdk is not declared)</code>	<code>&lt;=4</code>	Google Play <i>will not</i> filter the application from any devices based on their reported support for the <code>android.hardware.bluetooth</code> feature.
<code>&lt;=4</code>	<code>&gt;=5</code>	Google Play filters the application from any devices that do not support the <code>android.hardware.bluetooth</code> feature (including older releases).
<code>&gt;=5</code>	<code>&gt;=5</code>	

The examples below illustrate the different filtering effects, based on how Google Play handles the Bluetooth feature.

**In first example, an application that is designed to run on older API levels declares a Bluetooth permission, but does not declare the Bluetooth feature in a `<uses-feature>` element.**

*Result:* Google Play does not filter the application from any device.

```
<manifest ...>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-sdk android:minSdkVersion="3" />
    ...
</manifest>
```

**In the second example, below, the same application also declares a target API level of "5".**

*Result:* Google Play now assumes that the feature is required and will filter the application from all devices that do not report Bluetooth support, including devices running older versions of the platform.

```
<manifest ...>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="5" />
    ...
</manifest>
```

**Here the same application now specifically declares the Bluetooth feature.**

*Result:* Identical to the previous example (filtering is applied).

```
<manifest ...>
    <uses-feature android:name="android.hardware.bluetooth" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="5" />
    ...
</manifest>
```

**Finally, in the case below, the same application adds an `android:required="false"` attribute.**

*Result:* Google Play disables filtering based on Bluetooth feature support, for all devices.

```
<manifest ...>
    <uses-feature android:name="android.hardware.bluetooth" android:required="false" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="5" />
    ...
</manifest>
```

## Testing the features required by your application

You can use the `aapt` tool, included in the Android SDK, to determine how Google Play will filter your application, based on its declared features and permissions. To do so, run `aapt` with the `dump badging` com-

mand. This causes `aapt` to parse your application's manifest and apply the same rules as used by Google Play to determine the features that your application requires.

To use the tool, follow these steps:

1. First, build and export your application as an unsigned `.apk`. If you are developing in Eclipse with ADT, right-click the project and select **Android Tools > Export Unsigned Application Package**. Select a destination filename and path and click **OK**.
2. Next, locate the `aapt` tool, if it is not already in your PATH. If you are using SDK Tools r8 or higher, you can find `aapt` in the `<SDK>/platform-tools/` directory.

**Note:** You must use the version of `aapt` that is provided for the latest Platform-Tools component available. If you do not have the latest Platform-Tools component, download it using the [Android SDK Manager](#).

3. Run `aapt` using this syntax:

```
$ aapt dump badging <path_to_exported_.apk>
```

Here's an example of the command output for the second Bluetooth example, above:

```
$ ./aapt dump badging BTExample.apk
package: name='com.example.android.btexample' versionCode='1' versionName='1'
uses-permission:'android.permission.BLUETOOTH_ADMIN'
uses-feature:'android.hardware.bluetooth'
sdkVersion:'3'
targetSdkVersion:'5'
application: label='BT Example' icon='res/drawable/app_bt_ex.png'
launchable activity name='com.example.android.btexample.MyActivity' label='MyActivity'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'small' 'normal' 'large'
locales: '--_--'
densities: '160'
```

## Features Reference

The tables below provide reference information about hardware and software features and the permissions that can imply them on Google Play.

### Hardware features

The table below describes the hardware feature descriptors supported by the most current platform release. To signal that your application uses or requires a hardware feature, declare each value in a `android:name` attribute in a separate `<uses-feature>` element.

Feature Type	Feature Descriptor	Description
Audio	<code>android.hardware.audio.low_latency</code>	The application uses a low-latency audio pipeline on the device and is sensitive to delays or lag in sound input or output.

Bluetooth	android.hardware.bluetooth	The application uses Bluetooth radio features in the device.
	android.hardware.camera	The application uses the device's camera. If the device supports multiple cameras, the application uses the camera that facing away from the screen.
	android.hardware.camera.autofocus	Subfeature. The application uses the device camera's autofocus capability.
Camera	android.hardware.camera.flash	Subfeature. The application uses the device camera's flash.
	android.hardware.camera.front	Subfeature. The application uses front-facing camera on the device.
	android.hardware.camera.any	The application uses at least one camera facing in any direction. Use this in preference to android.hardware.camera if a back-facing camera is not required.
Location	android.hardware.location	The application uses one or more features on the device for determining location, such as GPS location, network location, or cell location.
	android.hardware.location.network	Subfeature. The application uses coarse location coordinates obtained from a network-based geolocation system supported on the device.
	android.hardware.location.gps	Subfeature. The application uses precise location coordinates obtained from a Global Positioning System receiver on the device.
Microphone	android.hardware.microphone	The application uses a microphone on the device.
NFC	android.hardware.nfc	The application uses Near Field Communications radio features in the device.
Sensors	android.hardware.sensor.accelerometer	The application uses motion readings from an accelerometer on the device.
	android.hardware.sensor.barometer	The application uses the device's barometer.
	android.hardware.sensor.compass	The application uses directional readings from a magnetometer (compass) on the device.
	android.hardware.sensor.gyroscope	The application uses the device's gyroscope sensor.
	android.hardware.sensor.light	The application uses the device's light sensor.

	android.hardware.sensor.proximity	The application uses the device's proximity sensor.
	android.hardware.screen.landscape	The application requires landscape orientation.
Screen	android.hardware.screen.portrait	The application requires portrait orientation.
Telephony	android.hardware.telephony	The application uses telephony features on the device, such as telephony radio with data communication services. Subfeature. The application uses CDMA telephony radio features on the device. Subfeature. The application uses GSM telephony radio features on the device.
Television	android.hardware.type.television	The application is designed for a television user experience.
Touchscreen	android.hardware.faketouch	The application uses basic touch interaction events, such as "click down", "click up", and drag.

`android.hardware.faketouch.multitouch.distinct`

The application performs distinct tracking of two or more "fingers" on a fake touch interface. This is a superset of the faketouch feature.

`android.hardware.faketouch.multitouch.jazzhand`

The application performs distinct tracking of five or more "fingers" on a fake touch interface. This is a superset of the faketouch feature.

`android.hardware.touchscreen`

The application uses touchscreen capabilities for gestures that are more interactive than basic touch events, such as a fling. This is a superset of the basic faketouch feature.

	android.hardware.touchscreen.multitouch	The application uses basic two-point multitouch capabilities on the device screen, such as for pinch gestures, but does not need to track touches independently. This is a superset of touchscreen feature.
	android.hardware.touchscreen.multitouch.distinct	Subfeature. The application uses advanced multipoint multitouch capabilities on the device screen, such as for tracking two or more points fully independently. This is a superset of multitouch feature.
	android.hardware.touchscreen.multitouch.jazzhand	The application uses advanced multipoint multitouch capabilities on the device screen, for tracking up to five points fully independently. This is a superset of distinct multitouch feature.
USB	android.hardware.usb.host	The application uses USB host mode features (behaves as the host and connects to USB devices).
	android.hardware.usb.accessory	The application uses USB accessory features (behaves as the US device and connects to USB hosts).
Wifi	android.hardware.wifi	The application uses 802.11 networking (wifi) features on the device.

## Software features

The table below describes the software feature descriptors supported by the most current platform release. To signal that your application uses or requires a software feature, declare each value in a `android:name` attribute in a separate `<uses-feature>` element.

Feature	Attribute Value	Description	Comments
Live Wallpaper	android.software.live_wallpaper	The application uses or provides Live Wallpapers.	

android.software.sip	The application uses SIP service on the device.
SIP/VOIP android.software.sip.voip	Subfeature. The application uses SIP-based VOIP service on the device. This subfeature implicitly declares the android.software.sip parent feature, unless declared with android:required="false".

## Permissions that Imply Feature Requirements

Some feature constants listed in the tables above were made available to applications *after* the corresponding API; for example, the android.hardware.bluetooth feature was added in Android 2.2 (API level 8), but the bluetooth API that it refers to was added in Android 2.0 (API level 5). Because of this, some apps were able to use the API before they had the ability to declare that they require the API via the <uses-feature> system.

To prevent those apps from being made available unintentionally, Google Play assumes that certain hardware-related permissions indicate that the underlying hardware features are required by default. For instance, applications that use Bluetooth must request the BLUETOOTH permission in a <uses-permission> element — for legacy apps, Google Play assumes that the permission declaration means that the underlying android.hardware.bluetooth feature is required by the application and sets up filtering based on that feature.

The table below lists permissions that imply feature requirements equivalent to those declared in <uses-feature> elements. Note that <uses-feature> declarations, including any declared android:required attribute, always take precedence over features implied by the permissions below.

For any of the permissions below, you can disable filtering based on the implied feature by explicitly declaring the implied feature explicitly, in a <uses-feature> element, with an android:required="false" attribute. For example, to disable any filtering based on the CAMERA permission, you would add this <uses-feature> declaration to the manifest file:

```
<uses-feature android:name="android.hardware.camera" android:required="false" />
```

Category	This Permission...	Implies This Feature Requirement
Bluetooth	BLUETOOTH	android.hardware.bluetooth  (See <a href="#">Special handling for Bluetooth feature</a> for details.)
Camera	BLUETOOTH_ADMIN CAMERA ACCESS_MOCK_LOCATION ACCESS_LOCATION_EXTRA_COMMANDS INSTALL_LOCATION_PROVIDER	android.hardware.bluetooth android.hardware.camera <i>and</i> android.hardware.camera.autofocus android.hardware.location android.hardware.location android.hardware.location.network <i>and</i> android.hardware.location android.hardware.location.gps <i>and</i> android.hardware.location
Location	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION	

Microphone	RECORD_AUDIO	android.hardware.microphone
	CALL_PHONE	android.hardware.telephony
	CALL_PRIVILEGED	android.hardware.telephony
	MODIFY_PHONE_STATE	android.hardware.telephony
	PROCESS_OUTGOING_CALLS	android.hardware.telephony
	READ_SMS	android.hardware.telephony
Telephony	RECEIVE_SMS	android.hardware.telephony
	RECEIVE_MMS	android.hardware.telephony
	RECEIVE_WAP_PUSH	android.hardware.telephony
	SEND_SMS	android.hardware.telephony
	WRITE_APN_SETTINGS	android.hardware.telephony
	WRITE_SMS	android.hardware.telephony
	ACCESS_WIFI_STATE	android.hardware.wifi
Wifi	CHANGE_WIFI_STATE	android.hardware.wifi
	CHANGE_WIFI_MULTICAST_STATE	android.hardware.wifi

# In this document

1. [What is API Level?](#)
2. [Uses of API Level in Android](#)
3. [Development Considerations](#)
  1. [Application forward compatibility](#)
  2. [Application backward compatibility](#)
  3. [Selecting a platform version and API Level](#)
  4. [Declaring a minimum API Level](#)
  5. [Testing against higher API Levels](#)
4. [Using a Provisional API Level](#)
5. [Filtering the Reference Documentation by API Level](#)



## Google Play Filtering

Google Play uses the `<uses-sdk>` attributes declared in your app manifest to filter your app from devices that do not meet its platform version requirements. Before setting these attributes, make sure that you understand [Google Play filters](#).

### syntax:

```
<uses-sdk android:minSdkVersion="integer"  
        android:targetSdkVersion="integer"  
        android:maxSdkVersion="integer" />
```

### contained in:

[`<manifest>`](#)

### description:

Lets you express an application's compatibility with one or more versions of the Android platform, by means of an API Level integer. The API Level expressed by an application will be compared to the API Level of a given Android system, which may vary among different Android devices.

Despite its name, this element is used to specify the API Level, *not* the version number of the SDK (software development kit) or Android platform. The API Level is always a single integer. You cannot derive the API Level from its associated Android version number (for example, it is not the same as the major version or the sum of the major and minor versions).

Also read the document about [Versioning Your Applications](#).

### attributes:

`android:minSdkVersion`

An integer designating the minimum API Level required for the application to run. The Android system will prevent the user from installing the application if the system's API Level is lower than the value specified in this attribute. You should always declare this attribute.

**Caution:** If you do not declare this attribute, the system assumes a default value of "1", which indicates that your application is compatible with all versions of Android. If your application is *not* compatible with all versions (for instance, it uses APIs introduced in API Level 3) and you have not declared the proper `minSdkVersion`, then when installed on a system with an API Level less than 3,

the application will crash during runtime when attempting to access the unavailable APIs. For this reason, be certain to declare the appropriate API Level in the `minSdkVersion` attribute.

#### **android:targetSdkVersion**

An integer designating the API Level that the application targets. If not set, the default value equals that given to `minSdkVersion`.

This attribute informs the system that you have tested against the target version and the system should not enable any compatibility behaviors to maintain your app's forward-compatibility with the target version. The application is still able to run on older versions (down to `minSdkVersion`).

As Android evolves with each new version, some behaviors and even appearances might change. However, if the API level of the platform is higher than the version declared by your app's `targetSdkVersion`, the system may enable compatibility behaviors to ensure that your app continues to work the way you expect. You can disable such compatibility behaviors by specifying `targetSdkVersion` to match the API level of the platform on which it's running. For example, setting this value to "11" or higher allows the system to apply a new default theme (Holo) to your app when running on Android 3.0 or higher and also disables [screen compatibility mode](#) when running on larger screens (because support for API level 11 implicitly supports larger screens).

There are many compatibility behaviors that the system may enable based on the value you set for this attribute. Several of these behaviors are described by the corresponding platform versions in the [Build.VERSION\\_CODES](#) reference.

To maintain your application along with each Android release, you should increase the value of this attribute to match the latest API level, then thoroughly test your application on the corresponding platform version.

Introduced in: API Level 4

#### **android:maxSdkVersion**

An integer designating the maximum API Level on which the application is designed to run.

In Android 1.5, 1.6, 2.0, and 2.0.1, the system checks the value of this attribute when installing an application and when re-validating the application after a system update. In either case, if the application's `maxSdkVersion` attribute is lower than the API Level used by the system itself, then the system will not allow the application to be installed. In the case of re-validation after system update, this effectively removes your application from the device.

To illustrate how this attribute can affect your application after system updates, consider the following example:

An application declaring `maxSdkVersion="5"` in its manifest is published on Google Play. A user whose device is running Android 1.6 (API Level 4) downloads and installs the app. After a few weeks, the user receives an over-the-air system update to Android 2.0 (API Level 5). After the update is installed, the system checks the application's `maxSdkVersion` and successfully re-validates it. The application functions as normal. However, some time later, the device receives another system update, this time to Android 2.0.1 (API Level 6). After the update, the system can no longer re-validate the application because the system's own API Level (6) is now higher than the maximum supported by the application (5). The system prevents the application from being visible to the user, in effect removing it from the device.

**Warning:** Declaring this attribute is not recommended. First, there is no need to set the attribute as means of blocking deployment of your application onto new versions of the Android platform as they

are released. By design, new versions of the platform are fully backward-compatible. Your application should work properly on new versions, provided it uses only standard APIs and follows development best practices. Second, note that in some cases, declaring the attribute can **result in your application being removed from users' devices after a system update** to a higher API Level. Most devices on which your application is likely to be installed will receive periodic system updates over the air, so you should consider their effect on your application before setting this attribute.

Introduced in: API Level 4

Future versions of Android (beyond Android 2.0.1) will no longer check or enforce the `maxSdkVersion` attribute during installation or re-validation. Google Play will continue to use the attribute as a filter, however, when presenting users with applications available for download.

**introduced in:**

API Level 1

## What is API Level?

API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.

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

- A core set of packages and classes
- A set of XML elements and attributes for declaring a manifest file
- A set of XML elements and attributes for declaring and accessing resources
- A set of Intents
- A set of permissions that applications can request, as well as permission enforcements included in the system

Each successive version of the Android platform can include updates to the Android application framework API that it delivers.

Updates to the framework API are designed so that the new API remains compatible with earlier versions of the API. That is, most changes in the API are additive and introduce new or replacement functionality. As parts of the API are upgraded, the older replaced parts are deprecated but are not removed, so that existing applications can still use them. In a very small number of cases, parts of the API may be modified or removed, although typically such changes are only needed to ensure API robustness and application or system security. All other API parts from earlier revisions are carried forward without modification.

The framework API that an Android platform delivers is specified using an integer identifier called "API Level". Each Android platform version supports exactly one API Level, although support is implicit for all earlier API Levels (down to API Level 1). The initial release of the Android platform provided API Level 1 and subsequent releases have incremented the API Level.

The table below specifies the API Level supported by each version of the Android platform. For information about the relative numbers of devices that are running each version, see the [Platform Versions dashboards page](#).

Platform Version	API Level	VERSION_CODE	Notes
<a href="#">Android 4.3</a>	<a href="#">18</a>	<a href="#">JELLY_BEAN_MR2</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.2, 4.2.2</a>	<a href="#">17</a>	<a href="#">JELLY_BEAN_MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.1, 4.1.1</a>	<a href="#">16</a>	<a href="#">JELLY_BEAN</a>	<a href="#">Platform Highlights</a>

<a href="#">Android 4.0.3, 4.0.4</a>	<a href="#">15</a>	<a href="#">ICE CREAM SANDWICH MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.0, 4.0.1, 4.0.2</a>	<a href="#">14</a>	<a href="#">ICE CREAM SANDWICH</a>	
<a href="#">Android 3.2</a>	<a href="#">13</a>	<a href="#">HONEYCOMB MR2</a>	
<a href="#">Android 3.1.x</a>	<a href="#">12</a>	<a href="#">HONEYCOMB MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 3.0.x</a>	<a href="#">11</a>	<a href="#">HONEYCOMB</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.4</a>	<a href="#">10</a>	<a href="#">GINGERBREAD MR1</a>	
<a href="#">Android 2.3.3</a>			
<a href="#">Android 2.3.2</a>			<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.1</a>	<a href="#">9</a>	<a href="#">GINGERBREAD</a>	
<a href="#">Android 2.3</a>			
<a href="#">Android 2.2.x</a>	<a href="#">8</a>	<a href="#">FROYO</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.1.x</a>	<a href="#">7</a>	<a href="#">ECLAIR_MR1</a>	
<a href="#">Android 2.0.1</a>	<a href="#">6</a>	<a href="#">ECLAIR_0_1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.0</a>	<a href="#">5</a>	<a href="#">ECLAIR</a>	
<a href="#">Android 1.6</a>	<a href="#">4</a>	<a href="#">DONUT</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.5</a>	<a href="#">3</a>	<a href="#">CUPCAKE</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.1</a>	<a href="#">2</a>	<a href="#">BASE_1_1</a>	
Android 1.0	<a href="#">1</a>	<a href="#">BASE</a>	

## Uses of API Level in Android

The API Level identifier serves a key role in ensuring the best possible experience for users and application developers:

- It lets the Android platform describe the maximum framework API revision that it supports
- It lets applications describe the framework API revision that they require
- It lets the system negotiate the installation of applications on the user's device, such that version-incompatible applications are not installed.

Each Android platform version stores its API Level identifier internally, in the Android system itself.

Applications can use a manifest element provided by the framework API — `<uses-sdk>` — to describe the minimum and maximum API Levels under which they are able to run, as well as the preferred API Level that they are designed to support. The element offers three key attributes:

- `android:minSdkVersion` — Specifies the minimum API Level on which the application is able to run. The default value is "1".
- `android:targetSdkVersion` — Specifies the API Level on which the application is designed to run. In some cases, this allows the application to use manifest elements or behaviors defined in the target API Level, rather than being restricted to using only those defined for the minimum API Level.
- `android:maxSdkVersion` — Specifies the maximum API Level on which the application is able to run. **Important:** Please read the [`<uses-sdk>`](#) documentation before using this attribute.

For example, to specify the minimum system API Level that an application requires in order to run, the application would include in its manifest a `<uses-sdk>` element with a `android:minSdkVersion` attribute. The value of `android:minSdkVersion` would be the integer corresponding to the API Level of the earliest version of the Android platform under which the application can run.

When the user attempts to install an application, or when revalidating an application after a system update, the Android system first checks the `<uses-sdk>` attributes in the application's manifest and compares the values against its own internal API Level. The system allows the installation to begin only if these conditions are met:

- If a `android:minSdkVersion` attribute is declared, its value must be less than or equal to the system's API Level integer. If not declared, the system assumes that the application requires API Level 1.
- If a `android:maxSdkVersion` attribute is declared, its value must be equal to or greater than the system's API Level integer. If not declared, the system assumes that the application has no maximum API Level. Please read the [`<uses-sdk>`](#) documentation for more information about how the system handles this attribute.

When declared in an application's manifest, a `<uses-sdk>` element might look like this:

```
<manifest>
  <uses-sdk android:minSdkVersion="5" />
  ...
</manifest>
```

The principal reason that an application would declare an API Level in `android:minSdkVersion` is to tell the Android system that it is using APIs that were *introduced* in the API Level specified. If the application were to be somehow installed on a platform with a lower API Level, then it would crash at run-time when it tried to access APIs that don't exist. The system prevents such an outcome by not allowing the application to be installed if the lowest API Level it requires is higher than that of the platform version on the target device.

For example, the [`android.appwidget`](#) package was introduced with API Level 3. If an application uses that API, it must declare a `android:minSdkVersion` attribute with a value of "3". The application will then be installable on platforms such as Android 1.5 (API Level 3) and Android 1.6 (API Level 4), but not on the Android 1.1 (API Level 2) and Android 1.0 platforms (API Level 1).

For more information about how to specify an application's API Level requirements, see the [`<uses-sdk>`](#) section of the manifest file documentation.

## Development Considerations

The sections below provide information related to API level that you should consider when developing your application.

### Application forward compatibility

Android applications are generally forward-compatible with new versions of the Android platform.

Because almost all changes to the framework API are additive, an Android application developed using any given version of the API (as specified by its API Level) is forward-compatible with later versions of the Android platform and higher API levels. The application should be able to run on all later versions of the Android platform, except in isolated cases where the application uses a part of the API that is later removed for some reason.

Forward compatibility is important because many Android-powered devices receive over-the-air (OTA) system updates. The user may install your application and use it successfully, then later receive an OTA update to a new version of the Android platform. Once the update is installed, your application will run in a new run-time version of the environment, but one that has the API and system capabilities that your application depends on.

In some cases, changes *below* the API, such those in the underlying system itself, may affect your application when it is run in the new environment. For that reason it's important for you, as the application developer, to understand how the application will look and behave in each system environment. To help you test your application on various versions of the Android platform, the Android SDK includes multiple platforms that you can

download. Each platform includes a compatible system image that you can run in an AVD, to test your application.

## Application backward compatibility

Android applications are not necessarily backward compatible with versions of the Android platform older than the version against which they were compiled.

Each new version of the Android platform can include new framework APIs, such as those that give applications access to new platform capabilities or replace existing API parts. The new APIs are accessible to applications when running on the new platform and, as mentioned above, also when running on later versions of the platform, as specified by API Level. Conversely, because earlier versions of the platform do not include the new APIs, applications that use the new APIs are unable to run on those platforms.

Although it's unlikely that an Android-powered device would be downgraded to a previous version of the platform, it's important to realize that there are likely to be many devices in the field that run earlier versions of the platform. Even among devices that receive OTA updates, some might lag and might not receive an update for a significant amount of time.

## Selecting a platform version and API Level

When you are developing your application, you will need to choose the platform version against which you will compile the application. In general, you should compile your application against the lowest possible version of the platform that your application can support.

You can determine the lowest possible platform version by compiling the application against successively lower build targets. After you determine the lowest version, you should create an AVD using the corresponding platform version (and API Level) and fully test your application. Make sure to declare a `android:minSdkVersion` attribute in the application's manifest and set its value to the API Level of the platform version.

## Declaring a minimum API Level

If you build an application that uses APIs or system features introduced in the latest platform version, you should set the `android:minSdkVersion` attribute to the API Level of the latest platform version. This ensures that users will only be able to install your application if their devices are running a compatible version of the Android platform. In turn, this ensures that your application can function properly on their devices.

If your application uses APIs introduced in the latest platform version but does *not* declare a `android:minSdkVersion` attribute, then it will run properly on devices running the latest version of the platform, but *not* on devices running earlier versions of the platform. In the latter case, the application will crash at runtime when it tries to use APIs that don't exist on the earlier versions.

## Testing against higher API Levels

After compiling your application, you should make sure to test it on the platform specified in the application's `android:minSdkVersion` attribute. To do so, create an AVD that uses the platform version required by your application. Additionally, to ensure forward-compatibility, you should run and test the application on all platforms that use a higher API Level than that used by your application.

The Android SDK includes multiple platform versions that you can use, including the latest version, and provides an updater tool that you can use to download other platform versions as necessary.

To access the updater, use the `android` command-line tool, located in the `<sdk>/tools` directory. You can launch the SDK updater by executing `android sdk`. You can also simply double-click the `android.bat` (Windows) or `android` (OS X/Linux) file. In ADT, you can also access the updater by selecting **Window > Android SDK Manager**.

To run your application against different platform versions in the emulator, create an AVD for each platform version that you want to test. For more information about AVDs, see [Creating and Managing Virtual Devices](#). If you are using a physical device for testing, ensure that you know the API Level of the Android platform it runs. See the table at the top of this document for a list of platform versions and their API Levels.

## Using a Provisional API Level

In some cases, an "Early Look" Android SDK platform may be available. To let you begin developing on the platform although the APIs may not be final, the platform's API Level integer will not be specified. You must instead use the platform's *provisional API Level* in your application manifest, in order to build applications against the platform. A provisional API Level is not an integer, but a string matching the codename of the unreleased platform version. The provisional API Level will be specified in the release notes for the Early Look SDK release notes and is case-sensitive.

The use of a provisional API Level is designed to protect developers and device users from inadvertently publishing or installing applications based on the Early Look framework API, which may not run properly on actual devices running the final system image.

The provisional API Level will only be valid while using the Early Look SDK and can only be used to run applications in the emulator. An application using the provisional API Level can never be installed on an Android device. At the final release of the platform, you must replace any instances of the provisional API Level in your application manifest with the final platform's actual API Level integer.

## Filtering the Reference Documentation by API Level

Reference documentation pages on the Android Developers site offer a "Filter by API Level" control in the top-right area of each page. You can use the control to show documentation only for parts of the API that are actually accessible to your application, based on the API Level that it specifies in the `android:minSdkVersion` attribute of its manifest file.

To use filtering, select the checkbox to enable filtering, just below the page search box. Then set the "Filter by API Level" control to the same API Level as specified by your application. Notice that APIs introduced in a later API Level are then grayed out and their content is masked, since they would not be accessible to your application.

Filtering by API Level in the documentation does not provide a view of what is new or introduced in each API Level — it simply provides a way to view the entire API associated with a given API Level, while excluding API elements introduced in later API Levels.

If you decide that you don't want to filter the API documentation, just disable the feature using the checkbox. By default, API Level filtering is disabled, so that you can view the full framework API, regardless of API Level.

Also note that the reference documentation for individual API elements specifies the API Level at which each element was introduced. The API Level for packages and classes is specified as "Since <api level>" at the top-right corner of the content area on each documentation page. The API Level for class members is specified in their detailed description headers, at the right margin.

# In this document

1. [What is API Level?](#)
2. [Uses of API Level in Android](#)
3. [Development Considerations](#)
  1. [Application forward compatibility](#)
  2. [Application backward compatibility](#)
  3. [Selecting a platform version and API Level](#)
  4. [Declaring a minimum API Level](#)
  5. [Testing against higher API Levels](#)
4. [Using a Provisional API Level](#)
5. [Filtering the Reference Documentation by API Level](#)



## Google Play Filtering

Google Play uses the `<uses-sdk>` attributes declared in your app manifest to filter your app from devices that do not meet its platform version requirements. Before setting these attributes, make sure that you understand [Google Play filters](#).

### syntax:

```
<uses-sdk android:minSdkVersion="integer"  
        android:targetSdkVersion="integer"  
        android:maxSdkVersion="integer" />
```

### contained in:

[`<manifest>`](#)

### description:

Lets you express an application's compatibility with one or more versions of the Android platform, by means of an API Level integer. The API Level expressed by an application will be compared to the API Level of a given Android system, which may vary among different Android devices.

Despite its name, this element is used to specify the API Level, *not* the version number of the SDK (software development kit) or Android platform. The API Level is always a single integer. You cannot derive the API Level from its associated Android version number (for example, it is not the same as the major version or the sum of the major and minor versions).

Also read the document about [Versioning Your Applications](#).

### attributes:

`android:minSdkVersion`

An integer designating the minimum API Level required for the application to run. The Android system will prevent the user from installing the application if the system's API Level is lower than the value specified in this attribute. You should always declare this attribute.

**Caution:** If you do not declare this attribute, the system assumes a default value of "1", which indicates that your application is compatible with all versions of Android. If your application is *not* compatible with all versions (for instance, it uses APIs introduced in API Level 3) and you have not declared the proper `minSdkVersion`, then when installed on a system with an API Level less than 3,

the application will crash during runtime when attempting to access the unavailable APIs. For this reason, be certain to declare the appropriate API Level in the `minSdkVersion` attribute.

#### **android:targetSdkVersion**

An integer designating the API Level that the application targets. If not set, the default value equals that given to `minSdkVersion`.

This attribute informs the system that you have tested against the target version and the system should not enable any compatibility behaviors to maintain your app's forward-compatibility with the target version. The application is still able to run on older versions (down to `minSdkVersion`).

As Android evolves with each new version, some behaviors and even appearances might change. However, if the API level of the platform is higher than the version declared by your app's `targetSdkVersion`, the system may enable compatibility behaviors to ensure that your app continues to work the way you expect. You can disable such compatibility behaviors by specifying `targetSdkVersion` to match the API level of the platform on which it's running. For example, setting this value to "11" or higher allows the system to apply a new default theme (Holo) to your app when running on Android 3.0 or higher and also disables [screen compatibility mode](#) when running on larger screens (because support for API level 11 implicitly supports larger screens).

There are many compatibility behaviors that the system may enable based on the value you set for this attribute. Several of these behaviors are described by the corresponding platform versions in the [Build.VERSION\\_CODES](#) reference.

To maintain your application along with each Android release, you should increase the value of this attribute to match the latest API level, then thoroughly test your application on the corresponding platform version.

Introduced in: API Level 4

#### **android:maxSdkVersion**

An integer designating the maximum API Level on which the application is designed to run.

In Android 1.5, 1.6, 2.0, and 2.0.1, the system checks the value of this attribute when installing an application and when re-validating the application after a system update. In either case, if the application's `maxSdkVersion` attribute is lower than the API Level used by the system itself, then the system will not allow the application to be installed. In the case of re-validation after system update, this effectively removes your application from the device.

To illustrate how this attribute can affect your application after system updates, consider the following example:

An application declaring `maxSdkVersion="5"` in its manifest is published on Google Play. A user whose device is running Android 1.6 (API Level 4) downloads and installs the app. After a few weeks, the user receives an over-the-air system update to Android 2.0 (API Level 5). After the update is installed, the system checks the application's `maxSdkVersion` and successfully re-validates it. The application functions as normal. However, some time later, the device receives another system update, this time to Android 2.0.1 (API Level 6). After the update, the system can no longer re-validate the application because the system's own API Level (6) is now higher than the maximum supported by the application (5). The system prevents the application from being visible to the user, in effect removing it from the device.

**Warning:** Declaring this attribute is not recommended. First, there is no need to set the attribute as means of blocking deployment of your application onto new versions of the Android platform as they

are released. By design, new versions of the platform are fully backward-compatible. Your application should work properly on new versions, provided it uses only standard APIs and follows development best practices. Second, note that in some cases, declaring the attribute can **result in your application being removed from users' devices after a system update** to a higher API Level. Most devices on which your application is likely to be installed will receive periodic system updates over the air, so you should consider their effect on your application before setting this attribute.

Introduced in: API Level 4

Future versions of Android (beyond Android 2.0.1) will no longer check or enforce the `maxSdkVersion` attribute during installation or re-validation. Google Play will continue to use the attribute as a filter, however, when presenting users with applications available for download.

**introduced in:**

API Level 1

## What is API Level?

API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.

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

- A core set of packages and classes
- A set of XML elements and attributes for declaring a manifest file
- A set of XML elements and attributes for declaring and accessing resources
- A set of Intents
- A set of permissions that applications can request, as well as permission enforcements included in the system

Each successive version of the Android platform can include updates to the Android application framework API that it delivers.

Updates to the framework API are designed so that the new API remains compatible with earlier versions of the API. That is, most changes in the API are additive and introduce new or replacement functionality. As parts of the API are upgraded, the older replaced parts are deprecated but are not removed, so that existing applications can still use them. In a very small number of cases, parts of the API may be modified or removed, although typically such changes are only needed to ensure API robustness and application or system security. All other API parts from earlier revisions are carried forward without modification.

The framework API that an Android platform delivers is specified using an integer identifier called "API Level". Each Android platform version supports exactly one API Level, although support is implicit for all earlier API Levels (down to API Level 1). The initial release of the Android platform provided API Level 1 and subsequent releases have incremented the API Level.

The table below specifies the API Level supported by each version of the Android platform. For information about the relative numbers of devices that are running each version, see the [Platform Versions dashboards page](#).

Platform Version	API Level	VERSION_CODE	Notes
<a href="#">Android 4.3</a>	<a href="#">18</a>	<a href="#">JELLY_BEAN_MR2</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.2, 4.2.2</a>	<a href="#">17</a>	<a href="#">JELLY_BEAN_MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.1, 4.1.1</a>	<a href="#">16</a>	<a href="#">JELLY_BEAN</a>	<a href="#">Platform Highlights</a>

<a href="#">Android 4.0.3, 4.0.4</a>	<a href="#">15</a>	<a href="#">ICE CREAM SANDWICH MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 4.0, 4.0.1, 4.0.2</a>	<a href="#">14</a>	<a href="#">ICE CREAM SANDWICH</a>	
<a href="#">Android 3.2</a>	<a href="#">13</a>	<a href="#">HONEYCOMB MR2</a>	
<a href="#">Android 3.1.x</a>	<a href="#">12</a>	<a href="#">HONEYCOMB MR1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 3.0.x</a>	<a href="#">11</a>	<a href="#">HONEYCOMB</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.4</a>	<a href="#">10</a>	<a href="#">GINGERBREAD MR1</a>	
<a href="#">Android 2.3.3</a>			
<a href="#">Android 2.3.2</a>			<a href="#">Platform Highlights</a>
<a href="#">Android 2.3.1</a>	<a href="#">9</a>	<a href="#">GINGERBREAD</a>	
<a href="#">Android 2.3</a>			
<a href="#">Android 2.2.x</a>	<a href="#">8</a>	<a href="#">FROYO</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.1.x</a>	<a href="#">7</a>	<a href="#">ECLAIR_MR1</a>	
<a href="#">Android 2.0.1</a>	<a href="#">6</a>	<a href="#">ECLAIR_0_1</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 2.0</a>	<a href="#">5</a>	<a href="#">ECLAIR</a>	
<a href="#">Android 1.6</a>	<a href="#">4</a>	<a href="#">DONUT</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.5</a>	<a href="#">3</a>	<a href="#">CUPCAKE</a>	<a href="#">Platform Highlights</a>
<a href="#">Android 1.1</a>	<a href="#">2</a>	<a href="#">BASE_1_1</a>	
Android 1.0	<a href="#">1</a>	<a href="#">BASE</a>	

## Uses of API Level in Android

The API Level identifier serves a key role in ensuring the best possible experience for users and application developers:

- It lets the Android platform describe the maximum framework API revision that it supports
- It lets applications describe the framework API revision that they require
- It lets the system negotiate the installation of applications on the user's device, such that version-incompatible applications are not installed.

Each Android platform version stores its API Level identifier internally, in the Android system itself.

Applications can use a manifest element provided by the framework API — `<uses-sdk>` — to describe the minimum and maximum API Levels under which they are able to run, as well as the preferred API Level that they are designed to support. The element offers three key attributes:

- `android:minSdkVersion` — Specifies the minimum API Level on which the application is able to run. The default value is "1".
- `android:targetSdkVersion` — Specifies the API Level on which the application is designed to run. In some cases, this allows the application to use manifest elements or behaviors defined in the target API Level, rather than being restricted to using only those defined for the minimum API Level.
- `android:maxSdkVersion` — Specifies the maximum API Level on which the application is able to run. **Important:** Please read the [`<uses-sdk>`](#) documentation before using this attribute.

For example, to specify the minimum system API Level that an application requires in order to run, the application would include in its manifest a `<uses-sdk>` element with a `android:minSdkVersion` attribute. The value of `android:minSdkVersion` would be the integer corresponding to the API Level of the earliest version of the Android platform under which the application can run.

When the user attempts to install an application, or when revalidating an application after a system update, the Android system first checks the `<uses-sdk>` attributes in the application's manifest and compares the values against its own internal API Level. The system allows the installation to begin only if these conditions are met:

- If a `android:minSdkVersion` attribute is declared, its value must be less than or equal to the system's API Level integer. If not declared, the system assumes that the application requires API Level 1.
- If a `android:maxSdkVersion` attribute is declared, its value must be equal to or greater than the system's API Level integer. If not declared, the system assumes that the application has no maximum API Level. Please read the [`<uses-sdk>`](#) documentation for more information about how the system handles this attribute.

When declared in an application's manifest, a `<uses-sdk>` element might look like this:

```
<manifest>
  <uses-sdk android:minSdkVersion="5" />
  ...
</manifest>
```

The principal reason that an application would declare an API Level in `android:minSdkVersion` is to tell the Android system that it is using APIs that were *introduced* in the API Level specified. If the application were to be somehow installed on a platform with a lower API Level, then it would crash at run-time when it tried to access APIs that don't exist. The system prevents such an outcome by not allowing the application to be installed if the lowest API Level it requires is higher than that of the platform version on the target device.

For example, the [`android.appwidget`](#) package was introduced with API Level 3. If an application uses that API, it must declare a `android:minSdkVersion` attribute with a value of "3". The application will then be installable on platforms such as Android 1.5 (API Level 3) and Android 1.6 (API Level 4), but not on the Android 1.1 (API Level 2) and Android 1.0 platforms (API Level 1).

For more information about how to specify an application's API Level requirements, see the [`<uses-sdk>`](#) section of the manifest file documentation.

## Development Considerations

The sections below provide information related to API level that you should consider when developing your application.

### Application forward compatibility

Android applications are generally forward-compatible with new versions of the Android platform.

Because almost all changes to the framework API are additive, an Android application developed using any given version of the API (as specified by its API Level) is forward-compatible with later versions of the Android platform and higher API levels. The application should be able to run on all later versions of the Android platform, except in isolated cases where the application uses a part of the API that is later removed for some reason.

Forward compatibility is important because many Android-powered devices receive over-the-air (OTA) system updates. The user may install your application and use it successfully, then later receive an OTA update to a new version of the Android platform. Once the update is installed, your application will run in a new run-time version of the environment, but one that has the API and system capabilities that your application depends on.

In some cases, changes *below* the API, such those in the underlying system itself, may affect your application when it is run in the new environment. For that reason it's important for you, as the application developer, to understand how the application will look and behave in each system environment. To help you test your application on various versions of the Android platform, the Android SDK includes multiple platforms that you can

download. Each platform includes a compatible system image that you can run in an AVD, to test your application.

## Application backward compatibility

Android applications are not necessarily backward compatible with versions of the Android platform older than the version against which they were compiled.

Each new version of the Android platform can include new framework APIs, such as those that give applications access to new platform capabilities or replace existing API parts. The new APIs are accessible to applications when running on the new platform and, as mentioned above, also when running on later versions of the platform, as specified by API Level. Conversely, because earlier versions of the platform do not include the new APIs, applications that use the new APIs are unable to run on those platforms.

Although it's unlikely that an Android-powered device would be downgraded to a previous version of the platform, it's important to realize that there are likely to be many devices in the field that run earlier versions of the platform. Even among devices that receive OTA updates, some might lag and might not receive an update for a significant amount of time.

## Selecting a platform version and API Level

When you are developing your application, you will need to choose the platform version against which you will compile the application. In general, you should compile your application against the lowest possible version of the platform that your application can support.

You can determine the lowest possible platform version by compiling the application against successively lower build targets. After you determine the lowest version, you should create an AVD using the corresponding platform version (and API Level) and fully test your application. Make sure to declare a `android:minSdkVersion` attribute in the application's manifest and set its value to the API Level of the platform version.

## Declaring a minimum API Level

If you build an application that uses APIs or system features introduced in the latest platform version, you should set the `android:minSdkVersion` attribute to the API Level of the latest platform version. This ensures that users will only be able to install your application if their devices are running a compatible version of the Android platform. In turn, this ensures that your application can function properly on their devices.

If your application uses APIs introduced in the latest platform version but does *not* declare a `android:minSdkVersion` attribute, then it will run properly on devices running the latest version of the platform, but *not* on devices running earlier versions of the platform. In the latter case, the application will crash at runtime when it tries to use APIs that don't exist on the earlier versions.

## Testing against higher API Levels

After compiling your application, you should make sure to test it on the platform specified in the application's `android:minSdkVersion` attribute. To do so, create an AVD that uses the platform version required by your application. Additionally, to ensure forward-compatibility, you should run and test the application on all platforms that use a higher API Level than that used by your application.

The Android SDK includes multiple platform versions that you can use, including the latest version, and provides an updater tool that you can use to download other platform versions as necessary.

To access the updater, use the `android` command-line tool, located in the `<sdk>/tools` directory. You can launch the SDK updater by executing `android sdk`. You can also simply double-click the `android.bat` (Windows) or `android` (OS X/Linux) file. In ADT, you can also access the updater by selecting **Window > Android SDK Manager**.

To run your application against different platform versions in the emulator, create an AVD for each platform version that you want to test. For more information about AVDs, see [Creating and Managing Virtual Devices](#). If you are using a physical device for testing, ensure that you know the API Level of the Android platform it runs. See the table at the top of this document for a list of platform versions and their API Levels.

## Using a Provisional API Level

In some cases, an "Early Look" Android SDK platform may be available. To let you begin developing on the platform although the APIs may not be final, the platform's API Level integer will not be specified. You must instead use the platform's *provisional API Level* in your application manifest, in order to build applications against the platform. A provisional API Level is not an integer, but a string matching the codename of the unreleased platform version. The provisional API Level will be specified in the release notes for the Early Look SDK release notes and is case-sensitive.

The use of a provisional API Level is designed to protect developers and device users from inadvertently publishing or installing applications based on the Early Look framework API, which may not run properly on actual devices running the final system image.

The provisional API Level will only be valid while using the Early Look SDK and can only be used to run applications in the emulator. An application using the provisional API Level can never be installed on an Android device. At the final release of the platform, you must replace any instances of the provisional API Level in your application manifest with the final platform's actual API Level integer.

## Filtering the Reference Documentation by API Level

Reference documentation pages on the Android Developers site offer a "Filter by API Level" control in the top-right area of each page. You can use the control to show documentation only for parts of the API that are actually accessible to your application, based on the API Level that it specifies in the `android:minSdkVersion` attribute of its manifest file.

To use filtering, select the checkbox to enable filtering, just below the page search box. Then set the "Filter by API Level" control to the same API Level as specified by your application. Notice that APIs introduced in a later API Level are then grayed out and their content is masked, since they would not be accessible to your application.

Filtering by API Level in the documentation does not provide a view of what is new or introduced in each API Level — it simply provides a way to view the entire API associated with a given API Level, while excluding API elements introduced in later API Levels.

If you decide that you don't want to filter the API documentation, just disable the feature using the checkbox. By default, API Level filtering is disabled, so that you can view the full framework API, regardless of API Level.

Also note that the reference documentation for individual API elements specifies the API Level at which each element was introduced. The API Level for packages and classes is specified as "Since <api level>" at the top-right corner of the content area on each documentation page. The API Level for class members is specified in their detailed description headers, at the right margin.



# App Resources

It takes more than just code to build a great app. Resources are the additional files and static content that your code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and more.

## Blog Articles

### New Tools For Managing Screen Sizes

Android 3.2 includes new tools for supporting devices with a wide range of screen sizes. One important result is better support for a new size of screen; what is typically called a ?7-inch? tablet. This release also offers several new APIs to simplify developers? work in adjusting to different screen sizes.

### Holo Everywhere

Before Android 4.0 the variance in system themes from device to device could make it difficult to design an app with a single predictable look and feel. We set out to improve this situation for the developer community in Ice Cream Sandwich and beyond.

### New Mode for Apps on Large Screens

Android tablets are becoming more popular, and we're pleased to note that the vast majority of apps resize to the larger screens just fine. To keep the few apps that don't resize well from frustrating users with awkward-looking apps on their tablets, Android 3.2 introduces a screen compatibility mode that makes these apps more usable on tablets.

## Training

### Supporting Different Devices

This class teaches you how to use basic platform features that leverage alternative resources and other features so your app can provide an optimized user experience on a variety of Android-compatible devices, using a single application package (APK).

## **Designing for Multiple Screens**

This class shows you how to implement a user interface that's optimized for several screen configurations.

# Activities

## Quickview

- An activity provides a user interface for a single screen in your application
- Activities can move into the background and then be resumed with their state restored

## In this document

1. [Creating an Activity](#)
  1. [Implementing a user interface](#)
  2. [Declaring the activity in the manifest](#)
2. [Starting an Activity](#)
  1. [Starting an activity for a result](#)
3. [Shutting Down an Activity](#)
4. [Managing the Activity Lifecycle](#)
  1. [Implementing the lifecycle callbacks](#)
  2. [Saving activity state](#)
  3. [Handling configuration changes](#)
  4. [Coordinating activities](#)

## Key classes

1. [Activity](#)

## See also

1. [Tasks and Back Stack](#)

An [Activity](#) is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. The back stack abides to the basic "last in, first out" stack mechanism, so, when the user is done with the current activity and presses the *Back* button, it is popped from the stack (and destroyed) and the previous activity resumes. (The back stack is discussed more in the [Tasks and Back Stack](#) document.)

When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides you the opportunity to perform specific work that's appropriate to that state change. For instance, when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted. These state transitions are all part of the activity lifecycle.

The rest of this document discusses the basics of how to build and use an activity, including a complete discussion of how the activity lifecycle works, so you can properly manage the transition between various activity states.

## Creating an Activity

To create an activity, you must create a subclass of [Activity](#) (or an existing subclass of it). In your subclass, you need to implement callback methods that the system calls when the activity transitions between various states of its lifecycle, such as when the activity is being created, stopped, resumed, or destroyed. The two most important callback methods are:

### [onCreate\(\)](#)

You must implement this method. The system calls this when creating your activity. Within your implementation, you should initialize the essential components of your activity. Most importantly, this is where you must call [setContentView\(\)](#) to define the layout for the activity's user interface.

### [onPause\(\)](#)

The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

There are several other lifecycle callback methods that you should use in order to provide a fluid user experience between activities and handle unexpected interruptions that cause your activity to be stopped and even destroyed. All of the lifecycle callback methods are discussed later, in the section about [Managing the Activity Lifecycle](#).

## Implementing a user interface

The user interface for an activity is provided by a hierarchy of views—objects derived from the [View](#) class. Each view controls a particular rectangular space within the activity's window and can respond to user interaction. For example, a view might be a button that initiates an action when the user touches it.

Android provides a number of ready-made views that you can use to design and organize your layout. "Widgets" are views that provide a visual (and interactive) elements for the screen, such as a button, text field, checkbox, or just an image. "Layouts" are views derived from [ViewGroup](#) that provide a unique layout model for its child views, such as a linear layout, a grid layout, or relative layout. You can also subclass the [View](#) and [ViewGroup](#) classes (or existing subclasses) to create your own widgets and layouts and apply them to your activity layout.

The most common way to define a layout using views is with an XML layout file saved in your application resources. This way, you can maintain the design of your user interface separately from the source code that defines the activity's behavior. You can set the layout as the UI for your activity with [setContentView\(\)](#), passing the resource ID for the layout. However, you can also create new [Views](#) in your activity code and build a view hierarchy by inserting new [Views](#) into a [ViewGroup](#), then use that layout by passing the root [ViewGroup](#) to [setContentView\(\)](#).

For information about creating a user interface, see the [User Interface](#) documentation.

## Declaring the activity in the manifest

You must declare your activity in the manifest file in order for it to be accessible to the system. To declare your activity, open your manifest file and add an [<activity>](#) element as a child of the [<application>](#) element. For example:

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
...
</manifest >
```

There are several other attributes that you can include in this element, to define properties such as the label for the activity, an icon for the activity, or a theme to style the activity's UI. The [android:name](#) attribute is the only required attribute—it specifies the class name of the activity. Once you publish your application, you should not change this name, because if you do, you might break some functionality, such as application shortcuts (read the blog post, [Things That Cannot Change](#)).

See the [<activity>](#) element reference for more information about declaring your activity in the manifest.

## Using intent filters

An [<activity>](#) element can also specify various intent filters—using the [<intent-filter>](#) element—in order to declare how other application components may activate it.

When you create a new application using the Android SDK tools, the stub activity that's created for you automatically includes an intent filter that declares the activity responds to the "main" action and should be placed in the "launcher" category. The intent filter looks like this:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

The [<action>](#) element specifies that this is the "main" entry point to the application. The [<category>](#) element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).

If you intend for your application to be self-contained and not allow other applications to activate its activities, then you don't need any other intent filters. Only one activity should have the "main" action and "launcher" category, as in the previous example. Activities that you don't want to make available to other applications should have no intent filters and you can start them yourself using explicit intents (as discussed in the following section).

However, if you want your activity to respond to implicit intents that are delivered from other applications (and your own), then you must define additional intent filters for your activity. For each type of intent to which you want to respond, you must include an [<intent-filter>](#) that includes an [<action>](#) element and, optionally, a [<category>](#) element and/or a [<data>](#) element. These elements specify the type of intent to which your activity can respond.

For more information about how your activities can respond to intents, see the [Intents and Intent Filters](#) document.

# Starting an Activity

You can start another activity by calling `startActivity()`, passing it an `Intent` that describes the activity you want to start. The intent specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application). An intent can also carry small amounts of data to be used by the activity that is started.

When working within your own application, you'll often need to simply launch a known activity. You can do so by creating an intent that explicitly defines the activity you want to start, using the class name. For example, here's how one activity starts another activity named `SignInActivity`:

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

However, your application might also want to perform some action, such as send an email, text message, or status update, using data from your activity. In this case, your application might not have its own activities to perform such actions, so you can instead leverage the activities provided by other applications on the device, which can perform the actions for you. This is where intents are really valuable—you can create an intent that describes an action you want to perform and the system launches the appropriate activity from another application. If there are multiple activities that can handle the intent, then the user can select which one to use. For example, if you want to allow the user to send an email message, you can create the following intent:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

The `EXTRA_EMAIL` extra added to the intent is a string array of email addresses to which the email should be sent. When an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form. In this situation, the email application's activity starts and when the user is done, your activity resumes.

## Starting an activity for a result

Sometimes, you might want to receive a result from the activity that you start. In that case, start the activity by calling `startActivityForResult()` (instead of `startActivity()`). To then receive the result from the subsequent activity, implement the `onActivityResult()` callback method. When the subsequent activity is done, it returns a result in an `Intent` to your `onActivityResult()` method.

For example, perhaps you want the user to pick one of their contacts, so your activity can do something with the information in that contact. Here's how you can create such an intent and handle the result:

```
private void pickContact() {
    // Create an intent to "pick" a contact, as defined by the content provider
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode == PICK_CONTACT_REQUEST)
        // Perform a query to the contact's content provider for the contact's
        Cursor cursor = getContentResolver().query(data.getData(),
```

```

        new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not empty
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Do something with the selected contact's name...
        }
    }
}

```

This example shows the basic logic you should use in your [onActivityResult\(\)](#) method in order to handle an activity result. The first condition checks whether the request was successful—if it was, then the `resultCode` will be [RESULT\\_OK](#)—and whether the request to which this result is responding is known—in this case, the `requestCode` matches the second parameter sent with [startActivityForResult\(\)](#). From there, the code handles the activity result by querying the data returned in an [Intent](#) (the `data` parameter).

What happens is, a [ContentResolver](#) performs a query against a content provider, which returns a [Cursor](#) that allows the queried data to be read. For more information, see the [Content Providers](#) document.

For more information about using intents, see the [Intents and Intent Filters](#) document.

## Shutting Down an Activity

You can shut down an activity by calling its [finish\(\)](#) method. You can also shut down a separate activity that you previously started by calling [finishActivity\(\)](#).

**Note:** In most cases, you should not explicitly finish an activity using these methods. As discussed in the following section about the activity lifecycle, the Android system manages the life of an activity for you, so you do not need to finish your own activities. Calling these methods could adversely affect the expected user experience and should only be used when you absolutely do not want the user to return to this instance of the activity.

## Managing the Activity Lifecycle

Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application. The lifecycle of an activity is directly affected by its association with other activities, its task and back stack.

An activity can exist in essentially three states:

### **Resumed**

The activity is in the foreground of the screen and has user focus. (This state is also sometimes referred to as "running".)

### **Paused**

Another activity is in the foreground and has focus, but this one is still visible. That is, another activity is visible on top of this one and that activity is partially transparent or doesn't cover the entire screen. A paused activity is completely alive (the [Activity](#) object is retained in memory, it maintains all state and member information, and remains attached to the window manager), but can be killed by the system in extremely low memory situations.

### **Stopped**

The activity is completely obscured by another activity (the activity is now in the "background"). A stopped activity is also still alive (the [Activity](#) object is retained in memory, it maintains all state and

member information, but is *not* attached to the window manager). However, it is no longer visible to the user and it can be killed by the system when memory is needed elsewhere.

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its [finish\(\)](#) method), or simply killing its process. When the activity is opened again (after being finished or killed), it must be created all over.

## Implementing the lifecycle callbacks

When an activity transitions into and out of the different states described above, it is notified through various callback methods. All of the callback methods are hooks that you can override to do appropriate work when the state of your activity changes. The following skeleton activity includes each of the fundamental lifecycle methods:

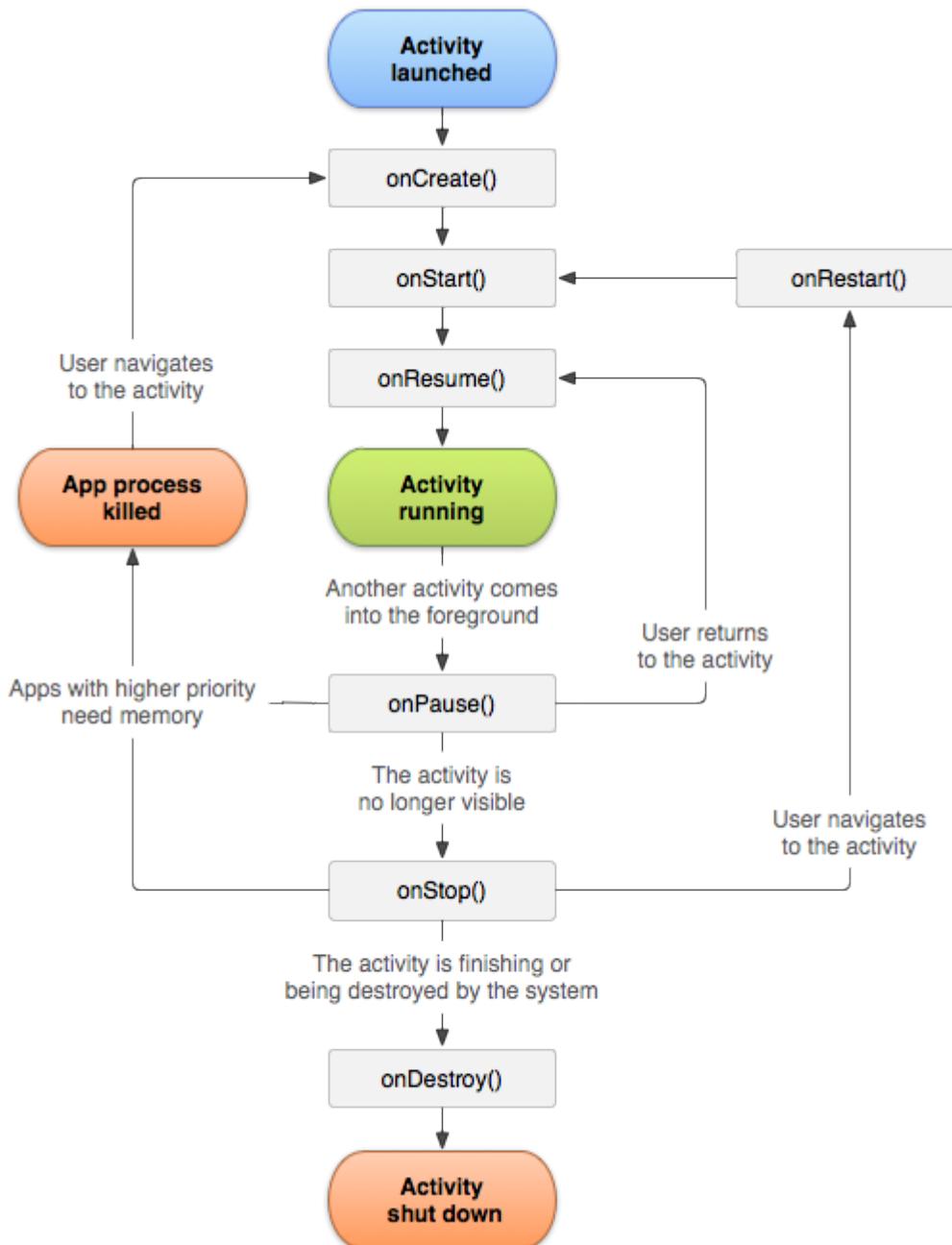
```
public class ExampleActivity extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        // The activity is being created.  
    }  
    @Override  
    protected void onStart\(\) {  
        super.onStart();  
        // The activity is about to become visible.  
    }  
    @Override  
    protected void onResume\(\) {  
        super.onResume();  
        // The activity has become visible (it is now "resumed").  
    }  
    @Override  
    protected void onPause\(\) {  
        super.onPause();  
        // Another activity is taking focus (this activity is about to be "paused").  
    }  
    @Override  
    protected void onStop\(\) {  
        super.onStop();  
        // The activity is no longer visible (it is now "stopped").  
    }  
    @Override  
    protected void onDestroy\(\) {  
        super.onDestroy();  
        // The activity is about to be destroyed.  
    }  
}
```

**Note:** Your implementation of these lifecycle methods must always call the superclass implementation before doing any work, as shown in the examples above.

Taken together, these methods define the entire lifecycle of an activity. By implementing these methods, you can monitor three nested loops in the activity lifecycle:

- The **entire lifetime** of an activity happens between the call to `onCreate()` and the call to `onDestroy()`. Your activity should perform setup of "global" state (such as defining layout) in `onCreate()`, and release all remaining resources in `onDestroy()`. For example, if your activity has a thread running in the background to download data from the network, it might create that thread in `onCreate()` and then stop the thread in `onDestroy()`.
- The **visible lifetime** of an activity happens between the call to `onStart()` and the call to `onStop()`. During this time, the user can see the activity on-screen and interact with it. For example, `onStop()` is called when a new activity starts and this one is no longer visible. Between these two methods, you can maintain resources that are needed to show the activity to the user. For example, you can register a `BroadcastReceiver` in `onStart()` to monitor changes that impact your UI, and unregister it in `onStop()` when the user can no longer see what you are displaying. The system might call `onStart()` and `onStop()` multiple times during the entire lifetime of the activity, as the activity alternates between being visible and hidden to the user.
- The **foreground lifetime** of an activity happens between the call to `onResume()` and the call to `onPause()`. During this time, the activity is in front of all other activities on screen and has user input focus. An activity can frequently transition in and out of the foreground—for example, `onPause()` is called when the device goes to sleep or when a dialog appears. Because this state can transition often, the code in these two methods should be fairly lightweight in order to avoid slow transitions that make the user wait.

Figure 1 illustrates these loops and the paths an activity might take between states. The rectangles represent the callback methods you can implement to perform operations when the activity transitions between states.



**Figure 1.** The activity lifecycle.

The same lifecycle callback methods are listed in table 1, which describes each of the callback methods in more detail and locates each one within the activity's overall lifecycle, including whether the system can kill the activity after the callback method completes.

**Table 1.** A summary of the activity lifecycle's callback methods.

Method	Description	Killable after?	Next
<a href="#">onCreate()</a>	Called when the activity is first created. This is where you should do all of your normal static set up — create views, bind data to lists, and so on. This method is passed a Bundle object containing the activity's previous state, if that state was captured (see <a href="#">Saving Activity State</a> , later).	No	<a href="#">onStart()</a>

Method	Description	Killable after?	Next
	Always followed by <code>onStart()</code> .		
<code>onRestart()</code>	Called after the activity has been stopped, just prior to it being started again.  Always followed by <code>onStart()</code>	No	<code>onStart()</code>
<code>onStart()</code>	Called just before the activity becomes visible to the user.  Followed by <code>onResume()</code> if the activity comes to the foreground, or <code>onStop()</code> if it becomes hidden.	No	<code>onResume()</code> or <code>onStop()</code>
<code>onResume()</code>	Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it.  Always followed by <code>onPause()</code> .	No	<code>onPause()</code>
<code>onPause()</code>	Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.  Followed either by <code>onResume()</code> if the activity returns back to the front, or by <code>onStop()</code> if it becomes invisible to the user.	Yes	<code>onResume()</code> or <code>onStop()</code>
<code>onStop()</code>	Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it.  Followed either by <code>onRestart()</code> if the activity is coming back to interact with the user, or by <code>onDestroy()</code> if this activity is going away.	Yes	<code>onRestart()</code> or <code>onDestroy()</code>
<code>onDestroy()</code>	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing (someone called <code>finish()</code> on	Yes	<i>nothing</i>

Method	Description	Killable after?	Next
	it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the <a href="#">isFinishing()</a> method.		

The column labeled "Killable after?" indicates whether or not the system can kill the process hosting the activity at any time *after the method returns*, without executing another line of the activity's code. Three methods are marked "yes": ([onPause\(\)](#), [onStop\(\)](#), and [onDestroy\(\)](#)). Because [onPause\(\)](#) is the first of the three, once the activity is created, [onPause\(\)](#) is the last method that's guaranteed to be called before the process *can* be killed—if the system must recover memory in an emergency, then [onStop\(\)](#) and [onDestroy\(\)](#) might not be called. Therefore, you should use [onPause\(\)](#) to write crucial persistent data (such as user edits) to storage. However, you should be selective about what information must be retained during [onPause\(\)](#), because any blocking procedures in this method block the transition to the next activity and slow the user experience.

Methods that are marked "No" in the **Killable** column protect the process hosting the activity from being killed from the moment they are called. Thus, an activity is killable from the time [onPause\(\)](#) returns to the time [onResume\(\)](#) is called. It will not again be killable until [onPause\(\)](#) is again called and returns.

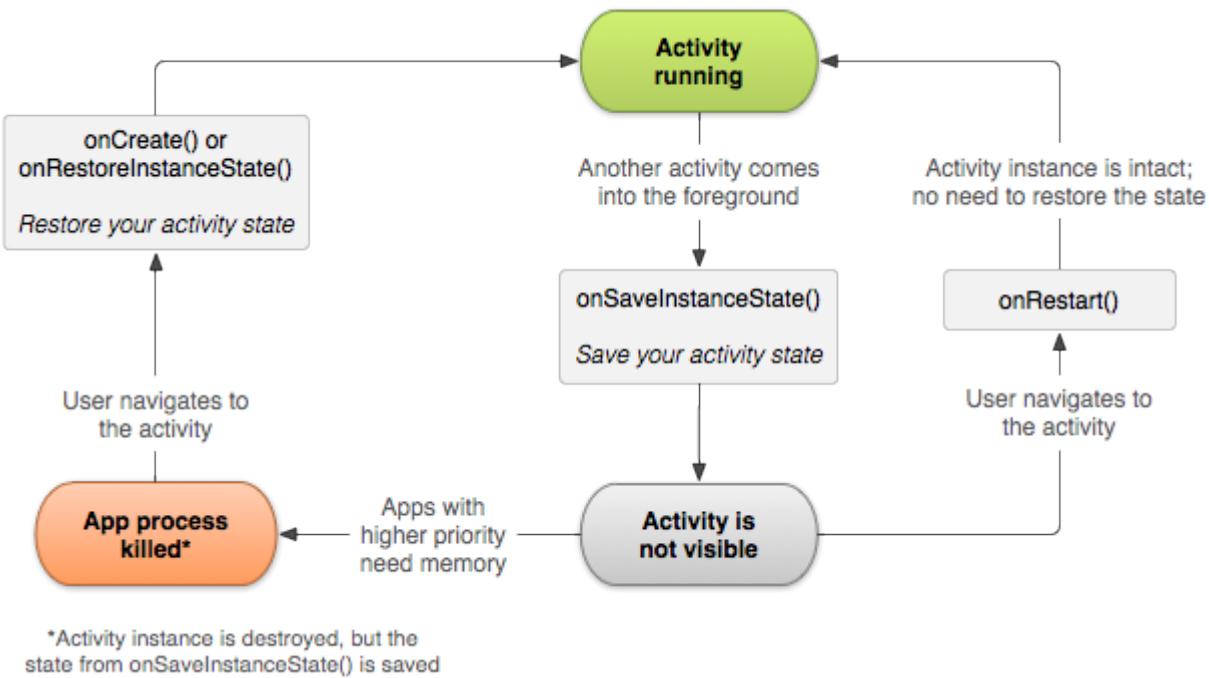
**Note:** An activity that's not technically "killable" by this definition in table 1 might still be killed by the system—but that would happen only in extreme circumstances when there is no other recourse. When an activity might be killed is discussed more in the [Processes and Threading](#) document.

## Saving activity state

The introduction to [Managing the Activity Lifecycle](#) briefly mentions that when an activity is paused or stopped, the state of the activity is retained. This is true because the [Activity](#) object is still held in memory when it is paused or stopped—all information about its members and current state is still alive. Thus, any changes the user made within the activity are retained so that when the activity returns to the foreground (when it "resumes"), those changes are still there.

However, when the system destroys an activity in order to recover memory, the [Activity](#) object is destroyed, so the system cannot simply resume it with its state intact. Instead, the system must recreate the [Activity](#) object if the user navigates back to it. Yet, the user is unaware that the system destroyed the activity and recreated it and, thus, probably expects the activity to be exactly as it was. In this situation, you can ensure that important information about the activity state is preserved by implementing an additional callback method that allows you to save information about the state of your activity: [onSaveInstanceState\(\)](#).

The system calls [onSaveInstanceState\(\)](#) before making the activity vulnerable to destruction. The system passes this method a [Bundle](#) in which you can save state information about the activity as name-value pairs, using methods such as [putString\(\)](#) and [putInt\(\)](#). Then, if the system kills your application process and the user navigates back to your activity, the system recreates the activity and passes the [Bundle](#) to both [onCreate\(\)](#) and [onRestoreInstanceState\(\)](#). Using either of these methods, you can extract your saved state from the [Bundle](#) and restore the activity state. If there is no state information to restore, then the [Bundle](#) passed to you is null (which is the case when the activity is created for the first time).



**Figure 2.** The two ways in which an activity returns to user focus with its state intact: either the activity is destroyed, then recreated and the activity must restore the previously saved state, or the activity is stopped, then resumed and the activity state remains intact.

**Note:** There's no guarantee that `onSaveInstanceState()` will be called before your activity is destroyed, because there are cases in which it won't be necessary to save the state (such as when the user leaves your activity using the *Back* button, because the user is explicitly closing the activity). If the system calls `onSaveInstanceState()`, it does so before `onStop()` and possibly before `onPause()`.

However, even if you do nothing and do not implement `onSaveInstanceState()`, some of the activity state is restored by the `Activity` class's default implementation of `onSaveInstanceState()`. Specifically, the default implementation calls the corresponding `onSaveInstanceState()` method for every `View` in the layout, which allows each view to provide information about itself that should be saved. Almost every widget in the Android framework implements this method as appropriate, such that any visible changes to the UI are automatically saved and restored when your activity is recreated. For example, the `EditText` widget saves any text entered by the user and the `CheckBox` widget saves whether it's checked or not. The only work required by you is to provide a unique ID (with the `android:id` attribute) for each widget you want to save its state. If a widget does not have an ID, then the system cannot save its state.

You can also explicitly stop a view in your layout from saving its state by setting the `android:saveEnabled` attribute to "false" or by calling the `setSaveEnabled()` method. Usually, you should not disable this, but you might if you want to restore the state of the activity UI differently.

Although the default implementation of `onSaveInstanceState()` saves useful information about your activity's UI, you still might need to override it to save additional information. For example, you might need to save member values that changed during the activity's life (which might correlate to values restored in the UI, but the members that hold those UI values are not restored, by default).

Because the default implementation of `onSaveInstanceState()` helps save the state of the UI, if you override the method in order to save additional state information, you should always call the superclass implementation of `onSaveInstanceState()` before doing any work. Likewise, you should also call the superclass implementation of `onRestoreInstanceState()` if you override it, so the default implementation can restore view states.

**Note:** Because `onSaveInstanceState()` is not guaranteed to be called, you should use it only to record the transient state of the activity (the state of the UI)—you should never use it to store persistent data. Instead, you should use `onPause()` to store persistent data (such as data that should be saved to a database) when the user leaves the activity.

A good way to test your application's ability to restore its state is to simply rotate the device so that the screen orientation changes. When the screen orientation changes, the system destroys and recreates the activity in order to apply alternative resources that might be available for the new screen configuration. For this reason alone, it's very important that your activity completely restores its state when it is recreated, because users regularly rotate the screen while using applications.

## Handling configuration changes

Some device configurations can change during runtime (such as screen orientation, keyboard availability, and language). When such a change occurs, Android recreates the running activity (the system calls `onDestroy()`, then immediately calls `onCreate()`). This behavior is designed to help your application adapt to new configurations by automatically reloading your application with alternative resources that you've provided (such as different layouts for different screen orientations and sizes).

If you properly design your activity to handle a restart due to a screen orientation change and restore the activity state as described above, your application will be more resilient to other unexpected events in the activity lifecycle.

The best way to handle such a restart is to save and restore the state of your activity using `onSaveInstanceState()` and `onRestoreInstanceState()` (or `onCreate()`), as discussed in the previous section.

For more information about configuration changes that happen at runtime and how you can handle them, read the guide to [Handling Runtime Changes](#).

## Coordinating activities

When one activity starts another, they both experience lifecycle transitions. The first activity pauses and stops (though, it won't stop if it's still visible in the background), while the other activity is created. In case these activities share data saved to disc or elsewhere, it's important to understand that the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.

The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process and one is starting the other. Here's the order of operations that occur when Activity A starts Activity B:

1. Activity A's `onPause()` method executes.
2. Activity B's `onCreate()`, `onStart()`, and `onResume()` methods execute in sequence. (Activity B now has user focus.)
3. Then, if Activity A is no longer visible on screen, its `onStop()` method executes.

This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another. For example, if you must write to a database when the first activity stops so that the following activity can read it, then you should write to the database during `onPause()` instead of during `onStop()`.