

# Building Your First App

Welcome to Android application development!

This class teaches you how to build your first Android app. You'll learn how to create an Android project and run a debuggable version of the app. You'll also learn some fundamentals of Android app design, including how to build a simple user interface and handle user input.

Before you start this class, be sure you have your development environment set up. You need to:

1. Download the Android SDK.
2. Install the ADT plugin for Eclipse (if you'll use the Eclipse IDE).
3. Download the latest SDK tools and platforms using the SDK Manager.

If you haven't already done these tasks, start by downloading the [Android SDK \(/sdk/index.html\)](#) and following the install steps. Once you've finished the setup, you're ready to begin this class.

This class uses a tutorial format that incrementally builds a small Android app that teaches you some fundamental concepts about Android development, so it's important that you follow each step.

[Start the first lesson > \(creating-project.html\)](#)

## DEPENDENCIES AND PREREQUISITES

- [Android SDK](#)
- [ADT Plugin](#) 20.0.0 or higher (if you're using Eclipse)

# Creating an Android Project

An Android project contains all the files that comprise the source code for your Android app. The Android SDK tools make it easy to start a new Android project with a set of default project directories and files.

This lesson shows how to create a new project either using Eclipse (with the ADT plugin) or using the SDK tools from a command line.

Note: You should already have the Android SDK installed, and if you're using Eclipse, you should also have the [ADT plugin \(/tools/sdk/eclipse-adt.html\)](#) installed (version 20.0.0 or higher). If you don't have these, follow the guide to [Installing the Android SDK \(/sdk/installing/index.html\)](#) before you start this lesson.

## THIS LESSON TEACHES YOU TO

1. [Create a Project with Eclipse](#)
2. [Create a Project with Command Line Tools](#)

## YOU SHOULD ALSO READ

- [Installing the SDK](#)
- [Managing Projects](#)

## Create a Project with Eclipse

1. In Eclipse, click New Android App Project  in the toolbar. (If you don't see this button, then you have not installed the ADT plugin—see [Installing the Eclipse Plugin](#).)

2. Fill in the form that appears:

- *Application Name* is the app name that appears to users. For this project, use "My First App."
- *Project Name* is the name of your project directory and the name visible in Eclipse.
- *Package Name* is the package namespace for your app (following the same rules as packages in the Java programming language). Your package name must be unique across all packages installed on the Android system. For this reason, it's generally best if you use a name that begins with the reverse domain name of your organization or publisher entity. For this project, you can use something like "com.example.myfirstapp."

However, you cannot publish your app on Google Play using the "com.example" namespace.

- *Build SDK* is the platform version against which you will compile your app. By default, this is set to the latest version of Android available in your SDK. (It should be Android 4.1 or greater; if you don't have such a version available, you must install one using the [SDK Manager](#)). You

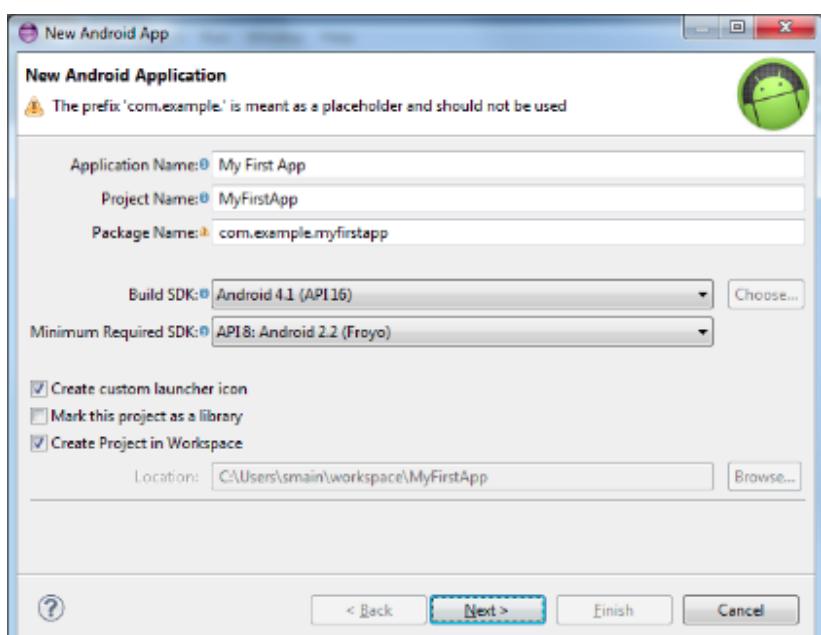


Figure 1. The New Android App Project wizard in Eclipse.

can still build your app to support older versions, but setting the build target to the latest version allows you to enable new features and optimize your app for a great user experience on the latest devices.

- *Minimum Required SDK* is the lowest version of Android that your app supports. To support as many devices as possible, you should set this to the lowest version available that allows your app to provide its core feature set. If any feature of your app is possible only on newer versions of Android and it's not critical to the app's core feature set, you can enable the feature only when running on the versions that support it.

Leave this set to the default value for this project.

Click Next.

3. The following screen provides tools to help you create a launcher icon for your app.

You can customize an icon in several ways and the tool generates an icon for all screen densities. Before you publish your app, you should be sure your icon meets the specifications defined in the [Iconography \(/design/style/iconography.html\)](#) design guide.

Click Next.

4. Now you can select an activity template from which to begin building your app.

For this project, select BlankActivity and click Next.

5. Leave all the details for the activity in their default state and click Finish.

Your Android project is now set up with some default files and you're ready to begin building the app. Continue to the [next lesson \(running-app.html\)](#).

## Create a Project with Command Line Tools

---

If you're not using the Eclipse IDE with the ADT plugin, you can instead create your project using the SDK tools from a command line:

1. Change directories into the Android SDK's tools/ path.
2. Execute:

```
android list targets
```

This prints a list of the available Android platforms that you've downloaded for your SDK. Find the platform against which you want to compile your app. Make a note of the target id. We recommend that you select the highest version possible. You can still build your app to support older versions, but setting the build target to the latest version allows you to optimize your app for the latest devices.

If you don't see any targets listed, you need to install some using the Android SDK Manager tool. See [Adding Platforms and Packages \(/sdk/installing/adding-packages.html\)](#).

3. Execute:

```
android create project --target <target-id> --name MyFirstApp \
```

```
--path <path-to-workspace>/MyFirstApp --activity MainActivity \
--package com.example.myfirstapp
```

Replace `<target-id>` with an id from the list of targets (from the previous step) and replace `<path-to-workspace>` with the location in which you want to save your Android projects.

Your Android project is now set up with several default configurations and you're ready to begin building the app. Continue to the [next lesson \(running-app.html\)](#).

Tip: Add the `platform-tools/` as well as the `tools/` directory to your PATH environment variable.

# Running Your App

If you followed the [previous lesson \(creating-project.html\)](#) to create an Android project, it includes a default set of "Hello World" source files that allow you to immediately run the app.

How you run your app depends on two things: whether you have a real Android-powered device and whether you're using Eclipse. This lesson shows you how to install and run your app on a real device and on the Android emulator, and in both cases with either Eclipse or the command line tools.

Before you run your app, you should be aware of a few directories and files in the Android project:

## AndroidManifest.xml

The [manifest file](#) describes the fundamental characteristics of the app and defines each of its components. You'll learn about various declarations in this file as you read more training classes.

### src/

Directory for your app's main source files. By default, it includes an [Activity](#) class that runs when your app is launched using the app icon.

### res/

Contains several sub-directories for [app resources](#). Here are just a few:

#### drawable-hdpi/

Directory for drawable objects (such as bitmaps) that are designed for high-density (hdpi) screens. Other drawable directories contain assets designed for other screen densities.

#### layout/

Directory for files that define your app's user interface.

#### values/

Directory for other various XML files that contain a collection of resources, such as string and color definitions.

When you build and run the default Android app, the default [Activity](#) ([/reference/android/app/Activity.html](#)) class starts and loads a layout file that says "Hello World." The result is nothing exciting, but it's important that you understand how to run your app before you start developing.

## Run on a Real Device

If you have a real Android-powered device, here's how you can install and run your app:

1. Plug in your device to your development machine with a USB cable. If you're developing on Windows, you might need to install the appropriate USB driver for your device. For help installing drivers, see the [OEM USB Drivers](#) document.

### THIS LESSON TEACHES YOU TO

1. [Run on a Real Device](#)
2. [Run on the Emulator](#)

### YOU SHOULD ALSO READ

- [Using Hardware Devices](#)
- [Managing Virtual Devices](#)
- [Managing Projects](#)

2. Ensure that USB debugging is enabled in the device Settings (open Settings and navigate to Applications > Development on most devices, or click Developer options on Android 4.0 and higher).

To run the app from Eclipse, open one of your project's files and click Run  from the toolbar. Eclipse installs the app on your connected device and starts it.

Or to run your app from a command line:

1. Change directories to the root of your Android project and execute:

```
ant debug
```

2. Make sure the Android SDK platform-tools/ directory is included in your PATH environment variable, then execute:

```
adb install bin/MyFirstApp-debug.apk
```

3. On your device, locate *MyFirstActivity* and open it.

That's how you build and run your Android app on a device! To start developing, continue to the [next lesson \(building-ui.html\)](#).

## Run on the Emulator

Whether you're using Eclipse or the command line, to run your app on the emulator you need to first create an [Android Virtual Device \(/tools/devices/index.html\)](#) (AVD). An AVD is a device configuration for the Android emulator that allows you to model different devices.

To create an AVD:

1. Launch the Android Virtual Device Manager:

- a. In Eclipse, click  from the toolbar.
- b. From the command line, change directories to `<sdk>/tools/` and execute:

```
android avd
```

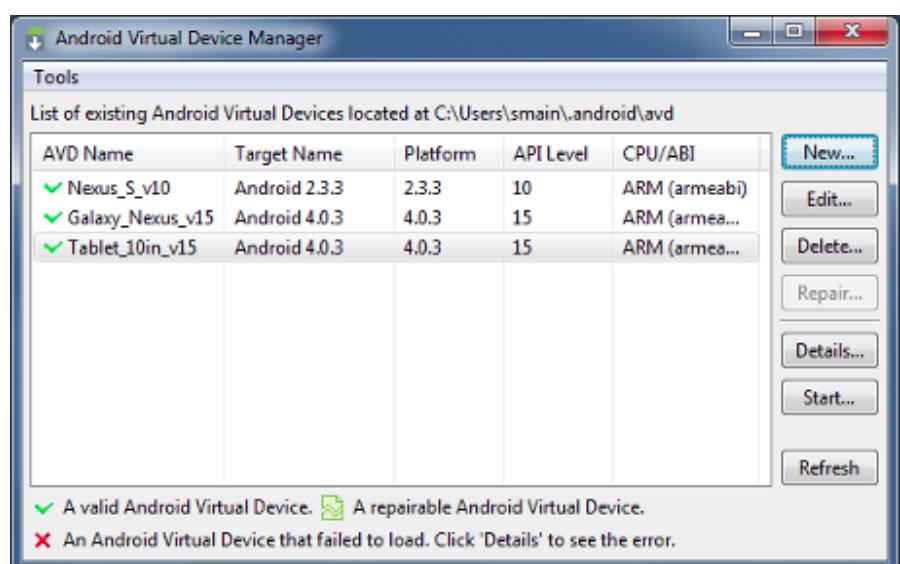


Figure 1. The AVD Manager showing a few virtual devices.

2. In the *Android Virtual Device Manager* panel, click **New**.
3. Fill in the details for the AVD. Give it a name, a platform target, an SD card size, and a skin

- (HVGA is default).
4. Click Create AVD.
  5. Select the new AVD from the *Android Virtual Device Manager* and click Start.
  6. After the emulator boots up, unlock the emulator screen.

To run the app from Eclipse, open one of your project's files and click Run  from the toolbar. Eclipse installs the app on your AVD and starts it.

Or to run your app from the command line:

1. Change directories to the root of your Android project and execute:

```
ant debug
```

2. Make sure the Android SDK platform-tools/ directory is included in your PATH environment variable, then execute:

```
adb install bin/MyFirstApp-debug.apk
```

3. On the emulator, locate *MyFirstActivity* and open it.

That's how you build and run your Android app on the emulator! To start developing, continue to the [next lesson \(building-ui.html\)](#).

# Building a Simple User Interface

The graphical user interface for an Android app is built using a hierarchy of [View \(/reference/android/view/View.html\)](#) and [ViewGroup \(/reference/android/view/ViewGroup.html\)](#) objects. [View \(/reference/android/view/View.html\)](#) objects are usually UI widgets such as [buttons \(/guide/topics/ui/controls/button.html\)](#) or [text fields \(/guide/topics/ui/controls/text.html\)](#) and [ViewGroup \(/reference/android/view/ViewGroup.html\)](#) objects are invisible view containers that define how the child views are laid out, such as in a grid or a vertical list.

Android provides an XML vocabulary that corresponds to the subclasses of [View \(/reference/android/view/View.html\)](#) and [ViewGroup \(/reference/android/view/ViewGroup.html\)](#) so you can define your UI in XML using a hierarchy of UI elements.

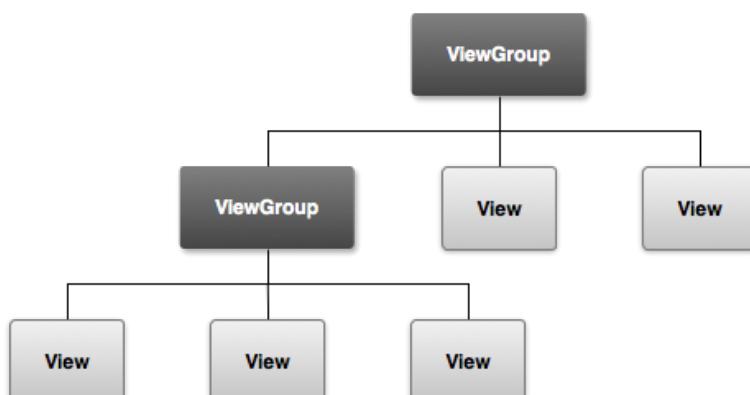


Figure 1. Illustration of how [ViewGroup \(/reference/android/view/ViewGroup.html\)](#) objects form branches in the layout and contain other [View \(/reference/android/view/View.html\)](#) objects.

In this lesson, you'll create a layout in XML that includes a text field and a button. In the following lesson, you'll respond when the button is pressed by sending the content of the text field to another activity.

## Create a Linear Layout

Open the `activity_main.xml` file from the `res/layout/` directory.

Note: In Eclipse, when you open a layout file, you're first shown the Graphical Layout editor. This is an editor that helps you build layouts using WYSIWYG tools. For this lesson, you're going to work directly with the XML, so click the `activity_main.xml` tab at the bottom of the screen to open

### THIS LESSON TEACHES YOU TO

1. [Create a Linear Layout](#)
2. [Add a Text Field](#)
3. [Add String Resources](#)
4. [Add a Button](#)
5. [Make the Input Box Fill in the Screen Width](#)

### YOU SHOULD ALSO READ

- [Layouts](#)

### Alternative Layouts

Declaring your UI layout in XML rather than runtime code is useful for several reasons, but it's especially important so you can create different layouts for different screen sizes. For example, you can create two versions of a layout and tell the system to use one on "small" screens and the other on "large" screens. For more information, see the class about [Supporting Different Devices \(/training/basics/supporting-devices/index.html\)](#).

the XML editor.

The BlankActivity template you used to start this project creates the `activity_main.xml` file with a [RelativeLayout](#) (`/reference/android/widget/RelativeLayout.html`) root view and a [TextView](#) (`/reference/android/widget/TextView.html`) child view.

First, delete the [`<TextView>`](#) (`/reference/android/widget/TextView.html`) element and change the [`<RelativeLayout>`](#) (`/reference/android/widget/RelativeLayout.html`) element to [`<LinearLayout>`](#) (`/reference/android/widget/LinearLayout.html`). Then add the [`android:orientation`](#) (`/reference/android/widget/LinearLayout.html#attr android:orientation`) attribute and set it to "horizontal". The result looks like this:

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

[LinearLayout](#) (`/reference/android/widget/LinearLayout.html`) is a view group (a subclass of [ViewGroup](#) (`/reference/android/view/ViewGroup.html`)) that lays out child views in either a vertical or horizontal orientation, as specified by the [`android:orientation`](#) (`/reference/android/widget/LinearLayout.html#attr android:orientation`) attribute. Each child of a [LinearLayout](#) (`/reference/android/widget/LinearLayout.html`) appears on the screen in the order in which it appears in the XML.

The other two attributes, [`android:layout\_width`](#) (`/reference/android/view/View.html#attr android:layout_width`) and [`android:layout\_height`](#) (`/reference/android/view/View.html#attr android:layout_height`), are required for all views in order to specify their size.

Because the [LinearLayout](#) (`/reference/android/widget/LinearLayout.html`) is the root view in the layout, it should fill the entire screen area that's available to the app by setting the width and height to "match\_parent". This value declares that the view should expand its width or height to *match* the width or height of the parent view.

For more information about layout properties, see the [Layout](#) (`/guide/topics/ui/declaring-layout.html`) guide.

## Add a Text Field

To create a user-editable text field, add an [`<EditText>`](#) (`/reference/android/widget/EditText.html`) element inside the [`<LinearLayout>`](#) (`/reference/android/widget/LinearLayout.html`).

Like every [View](#) (`/reference/android/view/View.html`) object, you must define certain XML attributes to specify the [`<EditText>`](#) (`/reference/android/widget/EditText.html`) object's properties. Here's how you should declare it inside the [`<LinearLayout>`](#) (`/reference/android/widget/LinearLayout.html`) element:

```
<EditText android:id="@+id/edit_message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/edit_message" />
```

About these attributes:

#### android:id

This provides a unique identifier for the view, which you can use to reference the object from your app code, such as to read and manipulate the object (you'll see this in the next lesson).

The at sign (@) is required when you're referring to any resource object from XML. It is followed by the resource type (id in this case), a slash, then the resource name (edit\_message).

The plus sign (+) before the resource type is needed only when you're defining a resource ID for the first time. When you compile the app, the SDK tools use the ID name to create a new resource ID in your project's gen/R.java file that refers to the [EditText](#) element.

Once the resource ID is declared once this way, other references to the ID do not need the plus sign. Using the plus sign is necessary only when specifying a new resource ID and not needed for concrete resources such as strings or layouts. See the sidebar for more information about resource objects.

#### android:layout\_width and android:layout\_height

Instead of using specific sizes for the width and height, the "wrap\_content" value specifies that the view should be only as big as needed to fit the contents of the view. If you were to instead use "match\_parent", then the [EditText](#) element would fill the screen, because it would match the size of the parent [LinearLayout](#). For more information, see the [Layouts](#) guide.

#### android:hint

This is a default string to display when the text field is empty. Instead of using a hard-coded string as the value, the "@string/edit\_message" value refers to a string resource defined in a separate file. Because this refers to a concrete resource (not just an identifier), it does not need the plus sign. However, because you haven't defined the string resource yet, you'll see a compiler error at first. You'll fix this in the next section by defining the string.

Note: This string resource has the same name as the element ID: edit\_message. However, references to resources are always scoped by the resource type (such as id or string), so using the same name does not cause collisions.

## About resource objects

A resource object is simply a unique integer name that's associated with an app resource, such as a bitmap, layout file, or string.

Every resource has a corresponding resource object defined in your project's gen/R.java file. You can use the object names in the R class to refer to your resources, such as when you need to specify a string value for the [android:hint](#) attribute. You can also create arbitrary resource IDs that you associate with a view using the [android:id](#) attribute, which allows you to reference that view from other code.

The SDK tools generate the R.java each time you compile your app. You should never modify this file by hand.

For more information, read the guide to [Providing Resources](#) ([/guide/topics/resources/providing-resources.html](#)).

## Add String Resources

When you need to add text in the user interface, you should always specify each string as a resource. String resources allow you to manage all UI text in a single location, which makes it easier to find and update text. Externalizing the strings also allows you to localize your app to different languages by providing alternative definitions for each string resource.

By default, your Android project includes a string resource file at `res/values/strings.xml`. Open this file and delete the `<string>` element named "hello\_world". Then add a new one named "edit\_message" and set the value to "Enter a message."

While you're in this file, also add a "Send" string for the button you'll soon add, called "button\_send".

The result for `strings.xml` looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">My First App</string>
    <string name="edit_message">Enter a message</string>
    <string name="button_send">Send</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">MainActivity</string>
</resources>
```

For more information about using string resources to localize your app for other languages, see the [Supporting Different Devices \(/training/basics/supporting-devices/index.html\)](#) class.

## Add a Button

Now add a `<Button>` ([/reference/android/widget/Button.html](#)) to the layout, immediately following the `<EditText>` ([/reference/android/widget/EditText.html](#)) element:

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

The height and width are set to "wrap\_content" so the button is only as big as necessary to fit the button's text. This button doesn't need the `android:id` ([/reference/android/view/View.html#attr\\_android:id](#)) attribute, because it won't be referenced from the activity code.

## Make the Input Box Fill in the Screen Width

The layout is currently designed so that both the `<EditText>` ([/reference/android/widget/EditText.html](#))

and [Button](#) widgets are only as big as necessary to fit their content, as shown in figure 2.

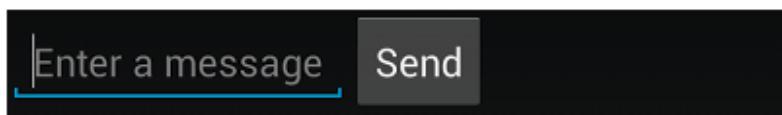


Figure 2. The [EditText](#) and [Button](#) widgets have their widths set to "wrap\_content".

This works fine for the button, but not as well for the text field, because the user might type something longer. So, it would be nice to fill the unused screen width with the text field. You can do this inside a [LinearLayout](#) with the *weight* property, which you can specify using the [android:layout\\_weight](#) attribute.

The weight value is a number that specifies the amount of remaining space each view should consume, relative to the amount consumed by sibling views. This works kind of like the amount of ingredients in a drink recipe: "2 parts vodka, 1 part coffee liqueur" means two-thirds of the drink is vodka. For example, if you give one view a weight of 2 and another one a weight of 1, the sum is 3, so the first view fills 2/3 of the remaining space and the second view fills the rest. If you add a third view and give it a weight of 1, then the first view (with weight of 2) now gets 1/2 the remaining space, while the remaining two each get 1/4.

The default weight for all views is 0, so if you specify any weight value greater than 0 to only one view, then that view fills whatever space remains after all views are given the space they require. So, to fill the remaining space in your layout with the [EditText](#) element, give it a weight of 1 and leave the button with no weight.

```
<EditText  
    android:layout_weight="1"  
    ... />
```

In order to improve the layout efficiency when you specify the weight, you should change the width of the [EditText](#) to be zero (0dp). Setting the width to zero improves layout performance because using "wrap\_content" as the width requires the system to calculate a width that is ultimately irrelevant because the weight value requires another width calculation to fill the remaining space.

```
<EditText  
    android:layout_weight="1"  
    android:layout_width="0dp"  
    ... />
```

Figure 3 shows the result when you assign all weight to the [EditText](#) element.



Figure 3. The `EditText` (</reference/android/widget/EditText.html>) widget is given all the layout weight, so fills the remaining space in the `LinearLayout` (</reference/android/widget/LinearLayout.html>).

Here's how your complete layout file should now look:

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

This layout is applied by the default `Activity` (</reference/android/app/Activity.html>) class that the SDK tools generated when you created the project, so you can now run the app to see the results:

- In Eclipse, click Run from the toolbar.
- Or from a command line, change directories to the root of your Android project and execute:

```
ant debug
adb install bin/MyFirstApp-debug.apk
```

Continue to the next lesson to learn how you can respond to button presses, read content from the text field, start another activity, and more.

# Starting Another Activity

After completing the [previous lesson \(building-ui.html\)](#), you have an app that shows an activity (a single screen) with a text field and a button. In this lesson, you'll add some code to MainActivity that starts a new activity when the user clicks the Send button.

## Respond to the Send Button

To respond to the button's on-click event, open the main.xml layout file and add the [android:onClick](#) ([/reference/android/view/View.html#attr\\_android:onClick](#)) attribute to the [<Button>](#) ([/reference/android/widget/Button.html](#)) element:

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/button_send"  
    android:onClick="sendMessage" />
```

The [android:onClick](#) ([/reference/android/view/View.html#attr\\_android:onClick](#)) attribute's value, "sendMessage", is the name of a method in your activity that the system calls when the user clicks the button.

Open the MainActivity class and add the corresponding method:

```
/** Called when the user clicks the Send button */  
public void sendMessage(View view) {  
    // Do something in response to button  
}
```

**Tip:** In Eclipse, press Ctrl + Shift + O to import missing classes (Cmd + Shift + O on Mac).

In order for the system to match this method to the method name given to [android:onClick](#) ([/reference/android/view/View.html#attr\\_android:onClick](#)), the signature must be exactly as shown. Specifically, the method must:

- Be public
- Have a void return value
- Have a [View](#) as the only parameter (this will be the [View](#) that was clicked)

Next, you'll fill in this method to read the contents of the text field and deliver that text to another activity.

### THIS LESSON TEACHES YOU TO

1. [Respond to the Send Button](#)
2. [Build an Intent](#)
3. [Start the Second Activity](#)
4. [Create the Second Activity](#)
  1. [Add it to the manifest](#)
5. [Receive the Intent](#)
6. [Display the Message](#)

### YOU SHOULD ALSO READ

- [Installing the SDK](#)

## Build an Intent

An [Intent](#) ([/reference/android/content/Intent.html](#)) is an object that provides runtime binding between separate components (such as two activities). The [Intent](#) ([/reference/android/content/Intent.html](#)) represents an app's "intent to do something." You can use intents for a wide variety of tasks, but most often they're used to start another activity.

Inside the `sendMessage()` method, create an [Intent](#) ([/reference/android/content/Intent.html](#)) to start an activity called `DisplayMessageActivity`:

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

The constructor used here takes two parameters:

- A [Context](#) as its first parameter (this is used because the [Activity](#) class is a subclass of [Context](#))
- The [Class](#) of the app component to which the system should deliver the [Intent](#) (in this case, the activity that should be started)

Note: The reference to `DisplayMessageActivity` will raise an error if you're using an IDE such as Eclipse because the class doesn't exist yet. Ignore the error for now; you'll create the class soon.

An intent not only allows you to start another activity, but it can carry a bundle of data to the activity as well. So, use `findViewById()` ([/reference/android/app/Activity.html#findViewById\(int\)](#)) to get the `EditText` ([/reference/android/widget/EditText.html](#)) element and add its text value to the intent:

```
Intent intent = new Intent(this, DisplayMessageActivity.class);
EditText editText = (EditText) findViewById(R.id.editText);
String message = editText.getText().toString();
intent.putExtra(EXTRA_MESSAGE, message);
```

An [Intent](#) ([/reference/android/content/Intent.html](#)) can carry a collection of various data types as key-value pairs called *extras*. The `putExtra()` ([/reference/android/content/Intent.html#putExtra\(java.lang.String, android.os.Bundle\)](#)) method takes the key name in the first parameter and the value in the second parameter.

In order for the next activity to query the extra data, you should define your key using a public constant. So add the `EXTRA_MESSAGE` definition to the top of the `MainActivity` class:

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

### Sending an intent to other apps

The intent created in this lesson is what's considered an *explicit intent*, because the [Intent](#) ([/reference/android/content/Intent.html](#)) specifies the exact app component to which the intent should be given. However, intents can also be *implicit*, in which case the [Intent](#) ([/reference/android/content/Intent.html](#)) does not specify the desired component, but allows any app installed on the device to respond to the intent as long as it satisfies the meta-data specifications for the action that's specified in various [Intent](#) ([/reference/android/content/Intent.html](#)) parameters. For more information, see the class about [Interacting with Other Apps](#) ([/training/basics/intents/index.html](#)).

```
public final static String EXTRA_MESSAGE = "com.example.myfirstapp.MESSAGE";  
...  
}
```

It's generally a good practice to define keys for intent extras using your app's package name as a prefix. This ensures they are unique, in case your app interacts with other apps.

## Start the Second Activity

To start an activity, you simply need to call [startActivity\(\)](#) ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#)) and pass it your [Intent](#) ([/reference/android/content/Intent.html](#)). The system receives this call and starts an instance of the [Activity](#) ([/reference/android/app/Activity.html](#)) specified by the [Intent](#) ([/reference/android/content/Intent.html](#)).

With this new code, the complete sendMessage() method that's invoked by the Send button now looks like this:

```
/** Called when the user clicks the Send button */  
public void sendMessage(View view) {  
    Intent intent = new Intent(this, DisplayMessageActivity.class);  
    EditText editText = (EditText) findViewById(R.id.edit_message);  
    String message = editText.getText().toString();  
    intent.putExtra(EXTRA_MESSAGE, message);  
    startActivity(intent);  
}
```

Now you need to create the `DisplayMessageActivity` class in order for this to work.

## Create the Second Activity

To create a new activity using Eclipse:

1. Click New  in the toolbar.
2. In the window that appears, open the Android folder and select Android Activity. Click Next.
3. Select BlankActivity and click Next.
4. Fill in the activity details:
  - o Project: MyFirstApp
  - o Activity Name: DisplayMessageActivity
  - o Layout Name: activity\_display\_message
  - o Navigation Type: None
  - o Hierarchical Parent: com.example.myfirstapp.MainActivity
  - o Title: My Message

Click Finish.

If you're using a different IDE or the command line tools, create a new file named

DisplayMessageActivity.java in the project's src/ directory, next to the original MainActivity.java file.

Open the DisplayMessageActivity.java file. If you used Eclipse to create it, the class already includes an implementation of the required [onCreate\(\)](#) ([/reference/android](#)

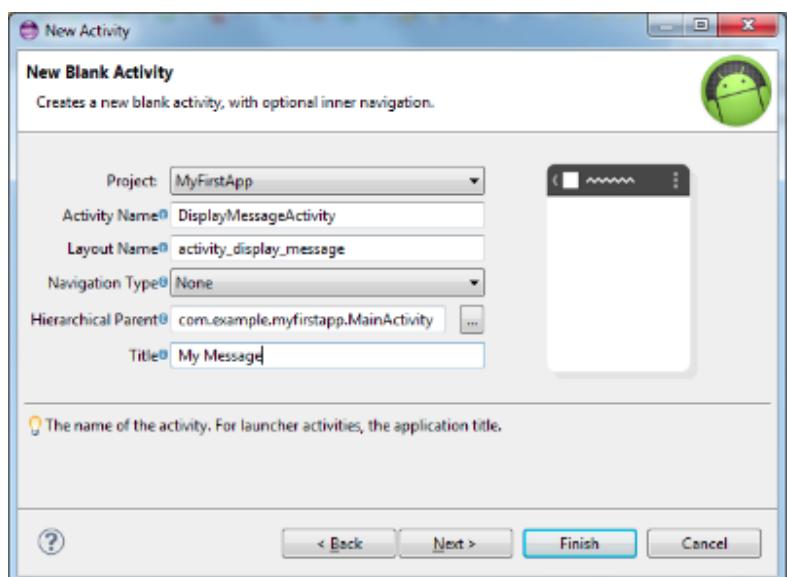


Figure 1. The new activity wizard in Eclipse.

[/app/Activity.html#onCreate\(android.os.Bundle\)](#) method. There's also an implementation of the [onCreateOptionsMenu\(\)](#) ([/reference/android](#)  
[/app/Activity.html#onCreateOptionsMenu\(android.view.Menu\)](#)) method, but you won't need it for this app so you can remove it. The class should look like this:

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

All subclasses of [Activity](#) ([/reference/android/app/Activity.html](#)) must implement the [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method. The system calls this when creating a new instance of the activity. It is where you must define the activity layout and where you should perform initial setup for the activity components.

## Add it to the manifest

You must declare all activities in your manifest file, `AndroidManifest.xml`, using an [`<activity>`](#) ([/guide/topics/manifest/activity-element.html](#)) element.

When you use the Eclipse tools to create the activity, it creates a default entry. It should look like this:

```
<application ... >  
    ...  
    <activity  
        android:name=".DisplayMessageActivity"
```

```
    android:label="@string/title_activity_display_message" >
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value="com.example.myfirstapp.MainActivity" />
</activity>
</application>
```

The `<meta-data>` (</guide/topics/manifest/meta-data-element.html>) element declares the name of this activity's parent activity within the app's logical hierarchy. The Android [Support Library](/tools/extras/support-library.html) (</tools/extras/support-library.html>) uses this information to implement default navigation behaviors, such as [Up navigation](/design/patterns/navigation.html) (</design/patterns/navigation.html>).

Note: During [installation](/sdk/installing/adding-packages.html) (</sdk/installing/adding-packages.html>), you should have downloaded the latest Support Library. Eclipse automatically includes this library in your app project (you can see the library's JAR file listed under *Android Dependencies*). If you're not using Eclipse, you may need to manually add the library to your project—follow this guide for [setting up the Support Library](/tools/extras/support-library.html#SettingUp) (</tools/extras/support-library.html#SettingUp>).

The app is now runnable because the [Intent](/reference/android/content/Intent.html) (</reference/android/content/Intent.html>) in the first activity now resolves to the `DisplayMessageActivity` class. If you run the app now, clicking the Send button starts the second activity, but it's still using the default "Hello world" layout.

## Receive the Intent

Every [Activity](/reference/android/app/Activity.html) (</reference/android/app/Activity.html>) is invoked by an [Intent](/reference/android/content/Intent.html) (</reference/android/content/Intent.html>), regardless of how the user navigated there. You can get the [Intent](/reference/android/content/Intent.html) (</reference/android/content/Intent.html>) that started your activity by calling `getIntent()` ([/reference/android/app/Activity.html#getIntent\(\)](/reference/android/app/Activity.html#getIntent())) and retrieve the data contained within it.

In the `DisplayMessageActivity` class's [`onCreate\(\)`](/reference/android/app/Activity.html#onCreate(android.os.Bundle)) ([>/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](/reference/android/app/Activity.html#onCreate(android.os.Bundle))) method, get the intent and extract the message delivered by `MainActivity`:

```
Intent intent = getIntent();
String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
```

## Display the Message

To show the message on the screen, create a [TextView](/reference/android/widget/TextView.html) (</reference/android/widget/TextView.html>) widget and set the text using `setText()` ([>/reference/android/widget/TextView.html#setText\(char\[\], int, int\)](/reference/android/widget/TextView.html#setText(char[], int, int))). Then add the [TextView](/reference/android/widget/TextView.html) (</reference/android/widget/TextView.html>) as the root view of the activity's layout by passing it to `setContentView()` ([>/reference/android/app/Activity.html#setContentView\(android.view.View\)](/reference/android/app/Activity.html#setContentView(android.view.View))).

The complete [`onCreate\(\)`](/reference/android/app/Activity.html#onCreate(android.os.Bundle)) ([>/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](/reference/android/app/Activity.html#onCreate(android.os.Bundle))) method for `DisplayMessageActivity` now looks like this:

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    // Get the message from the intent  
    Intent intent = getIntent();  
    String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);  
  
    // Create the text view  
    TextView textView = new TextView(this);  
    textView.setTextSize(40);  
    textView.setText(message);  
  
    // Set the text view as the activity layout  
    setContentView(textView);  
}
```

You can now run the app. When it opens, type a message in the text field, click Send, and the message appears on the second activity.

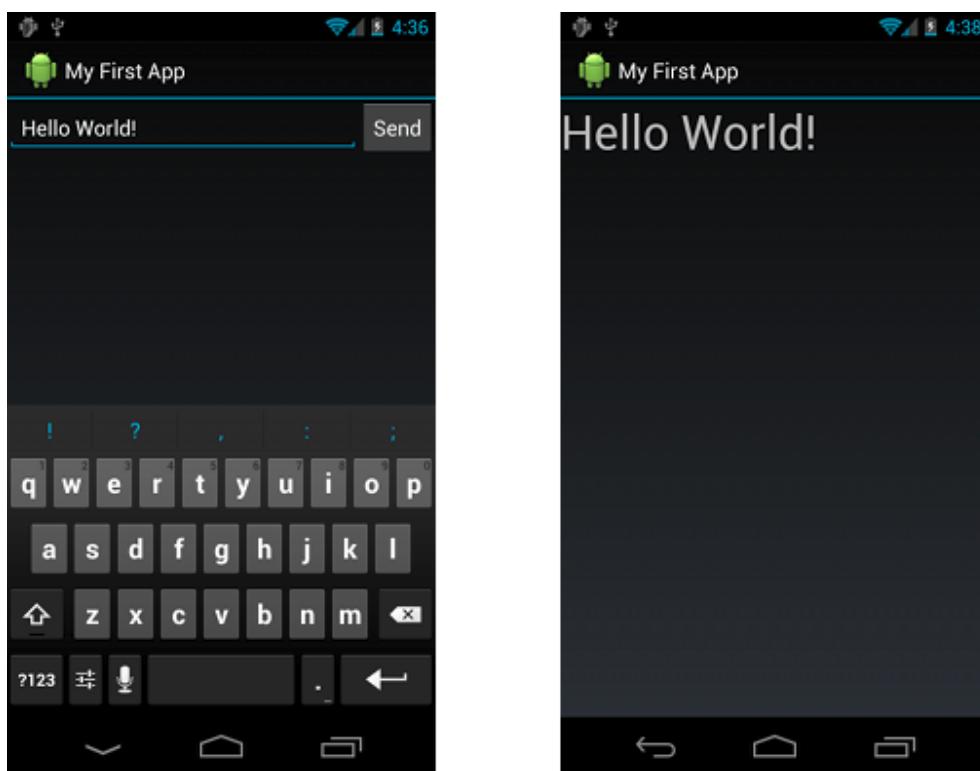


Figure 2. Both activities in the final app, running on Android 4.0.

That's it, you've built your first Android app!

To learn more about building Android apps, continue to follow the basic training classes. The next class is [Managing the Activity Lifecycle \(/training/basics/activity-lifecycle/index.html\)](#).

# Managing the Activity Lifecycle

As a user navigates through, out of, and back to your app, the [Activity](#) (`/reference/android/app/Activity.html`) instances in your app transition between different states in their lifecycle. For instance, when your activity starts for the first time, it comes to the foreground of the system and receives user focus. During this process, the Android system calls a series of lifecycle methods on the activity in which you set up the user interface and other components. If the user performs an action that starts another activity or switches to another app, the system calls another set of lifecycle methods on your activity as it moves into the background (where the activity is no longer visible, but the instance and its state remains intact).

Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity. For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot.

This class explains important lifecycle callback methods that each [Activity](#) (`/reference/android/app/Activity.html`) instance receives and how you can use them so your activity does what the user expects and does not consume system resources when your activity doesn't need them.

## Lessons

### **Starting an Activity**

Learn the basics about the activity lifecycle, how the user can launch your app, and how to perform basic activity creation.

### **Pausing and Resuming an Activity**

Learn what happens when your activity is paused (partially obscured) and resumed and what you should do during these state changes.

### **Stopping and Restarting an Activity**

Learn what happens when the user completely leaves your activity and returns to it.

### **Recreating an Activity**

Learn what happens when your activity is destroyed and how you can rebuild the activity state when necessary.

### **DEPENDENCIES AND PREREQUISITES**

- How to create an Android project (see [Creating an Android Project](#))

### **YOU SHOULD ALSO READ**

- [Activities](#)

### **TRY IT OUT**

[Download the demo](#)

ActivityLifecycle.zip

# Starting an Activity

Unlike other programming paradigms in which apps are launched with a `main()` method, the Android system initiates code in an [Activity](#) ([/reference/android/app/Activity.html](#)) instance by invoking specific callback methods that correspond to specific stages of its lifecycle. There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity.

This lesson provides an overview of the most important lifecycle methods and shows you how to handle the first lifecycle callback that creates a new instance of your activity.

## Understand the Lifecycle Callbacks

During the life of an activity, the system calls a core set of lifecycle methods in a sequence similar to a step pyramid. That is, each stage of the activity lifecycle is a separate step on the pyramid. As the system creates a new activity instance, each callback method moves the activity state one step toward the top. The top of the pyramid is the point at which the activity is running in the foreground and the user can interact with it.

As the user begins to leave the activity, the system calls other methods that move the activity state back down the pyramid in order to dismantle the activity. In some cases, the activity will move only part way down the pyramid and wait (such as when the user switches to another app), from which point the activity can move back to the top (if the user returns to the activity) and resume where the user left off.

### THIS LESSON TEACHES YOU TO

1. [Understand the Lifecycle Callbacks](#)
2. [Specify Your App's Launcher Activity](#)
3. [Create a New Instance](#)
4. [Destroy the Activity](#)

### YOU SHOULD ALSO READ

- [Activities](#)

### TRY IT OUT

[Download the demo](#)

ActivityLifecycle.zip

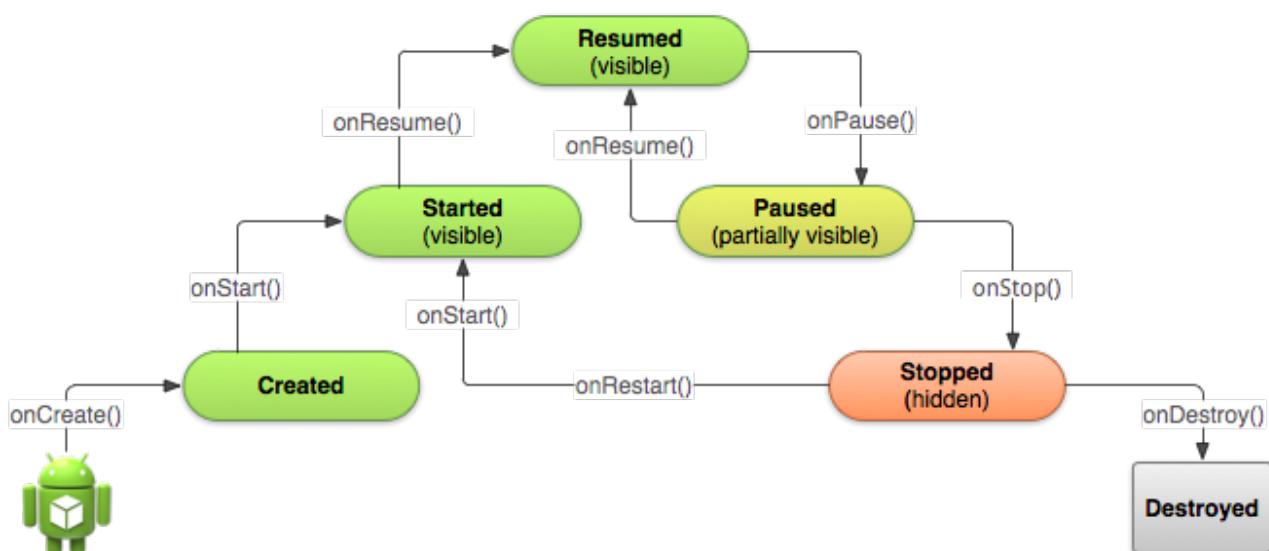


Figure 1. A simplified illustration of the Activity lifecycle, expressed as a step pyramid. This shows how, for every callback used to take the activity a step toward the Resumed state at the top, there's a callback method that takes the activity a step down. The activity can also return to the resumed state from the Paused and Stopped state.

Depending on the complexity of your activity, you probably don't need to implement all the lifecycle methods. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect. Implementing your activity lifecycle methods properly ensures your app behaves well in several ways, including that it:

- Does not crash if the user receives a phone call or switches to another app while using your app.
- Does not consume valuable system resources when the user is not actively using it.
- Does not lose the user's progress if they leave your app and return to it at a later time.
- Does not crash or lose the user's progress when the screen rotates between landscape and portrait orientation.

As you'll learn in the following lessons, there are several situations in which an activity transitions between different states that are illustrated in figure 1. However, only three of these states can be static. That is, the activity can exist in one of only three states for an extended period of time:

### Resumed

In this state, the activity is in the foreground and the user can interact with it. (Also sometimes referred to as the "running" state.)

### Paused

In this state, the activity is partially obscured by another activity—the other activity that's in the foreground is semi-transparent or doesn't cover the entire screen. The paused activity does not receive user input and cannot execute any code.

### Stopped

In this state, the activity is completely hidden and not visible to the user; it is considered to be in the background. While stopped, the activity instance and all its state information such as member variables is retained, but it cannot execute any code.

The other states (Created and Started) are transient and the system quickly moves from them to the next state by calling the next lifecycle callback method. That is, after the system calls [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)), it quickly calls

`onStart()` ([/reference/android/app/Activity.html#onStart\(\)](#)), which is quickly followed by `onResume()` ([/reference/android/app/Activity.html#onResume\(\)](#)).

That's it for the basic activity lifecycle. Now you'll start learning about some of the specific lifecycle behaviors.

## Specify Your App's Launcher Activity

When the user selects your app icon from the Home screen, the system calls the `onCreate()` ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method for the `Activity` ([/reference/android/app/Activity.html](#)) in your app that you've declared to be the "launcher" (or "main") activity. This is the activity that serves as the main entry point to your app's user interface.

You can define which activity to use as the main activity in the Android manifest file, `AndroidManifest.xml` ([/guide/topics/manifest/manifest-intro.html](#)), which is at the root of your project directory.

The main activity for your app must be declared in the manifest with an `<intent-filter>` ([/guide/topics/manifest/intent-filter-element.html](#)) that includes the `MAIN` ([/reference/android/content/Intent.html#ACTION\\_MAIN](#)) action and `LAUNCHER` ([/reference/android/content/Intent.html#CATEGORY\\_LAUNCHER](#)) category. For example:

```
<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>
```

Note: When you create a new Android project with the Android SDK tools, the default project files include an `Activity` ([/reference/android/app/Activity.html](#)) class that's declared in the manifest with this filter.

If either the `MAIN` ([/reference/android/content/Intent.html#ACTION\\_MAIN](#)) action or `LAUNCHER` ([/reference/android/content/Intent.html#CATEGORY\\_LAUNCHER](#)) category are not declared for one of your activities, then your app icon will not appear in the Home screen's list of apps.

## Create a New Instance

Most apps include several different activities that allow the user to perform different actions. Whether an activity is the main activity that's created when the user clicks your app icon or a different activity that your app starts in response to a user action, the system creates every new instance of `Activity` ([/reference/android/app/Activity.html](#)) by calling its `onCreate()` ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method.

You must implement the `onCreate()` ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method to perform basic application startup logic

that should happen only once for the entire life of the activity. For example, your implementation of [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) should define the user interface and possibly instantiate some class-scope variables.

For example, the following example of the [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method shows some code that performs some fundamental setup for the activity, such as declaring the user interface (defined in an XML layout file), defining member variables, and configuring some of the UI.

```
TextView mTextView; // Member variable for text view in the layout

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

    // Set the user interface layout for this Activity
    // The layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);

    // Initialize member TextView so we can manipulate it later
    mTextView = (TextView) findViewById(R.id.text_message);

    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        // For the main activity, make sure the app icon in the action bar
        // does not behave as a button
        ActionBar actionBar = getActionBar();
        actionBar.setHomeButtonEnabled(false);
    }
}
```

Caution: Using the [SDK\\_INT](#) ([/reference/android/os/Build.VERSION.html#SDK\\_INT](#)) to prevent older system's from executing new APIs works in this way on Android 2.0 (API level 5) and higher only. Older versions will encounter a runtime exception.

Once the [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) finishes execution, the system calls the [onStart\(\)](#) ([/reference/android/app/Activity.html#onStart\(\)](#)) and [onResume\(\)](#) ([/reference/android/app/Activity.html#onResume\(\)](#)) methods in quick succession. Your activity never resides in the Created or Started states. Technically, the activity becomes visible to the user when [onStart\(\)](#) ([/reference/android/app/Activity.html#onStart\(\)](#)) is called, but [onResume\(\)](#) ([/reference/android/app/Activity.html#onResume\(\)](#)) quickly follows and the activity remains in the Resumed state until something occurs to change that, such as when a phone call is received, the user navigates to another activity, or the device screen turns off.

In the other lessons that follow, you'll see how the other start up methods, [onStart\(\)](#) ([/reference/android/app/Activity.html#onStart\(\)](#)) and [onResume\(\)](#) ([/reference/android/app/Activity.html#onResume\(\)](#)), are useful during your activity's lifecycle when used to resume the activity from the Paused or Stopped states.

Note: The [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method

includes a parameter called `savedInstanceState` that's discussed in the latter lesson about [Recreating an Activity \(recreating.html\)](#).

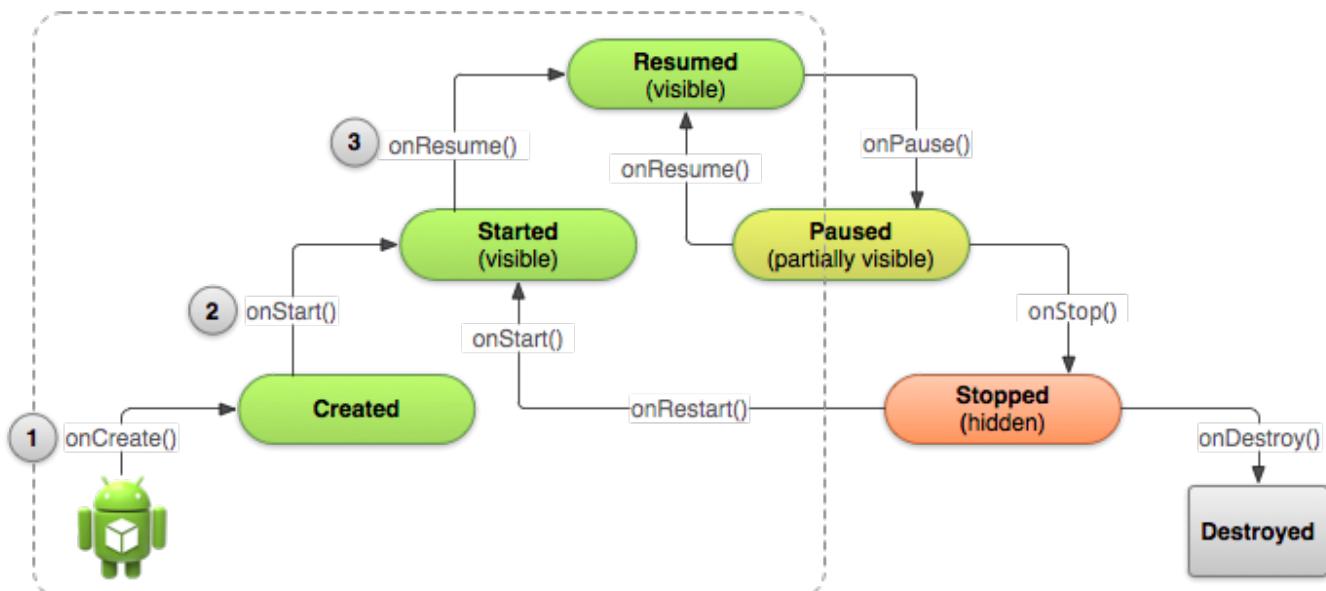


Figure 2. Another illustration of the activity lifecycle structure with an emphasis on the three main callbacks that the system calls in sequence when creating a new instance of the activity: [onCreate\(\)](#), [onStart\(\)](#), and [onResume\(\)](#). Once this sequence of callbacks complete, the activity reaches the Resumed state where users can interact with the activity until they switch to a different activity.

## Destroy the Activity

While the activity's first lifecycle callback is [onCreate\(\)](#), its very last callback is [onDestroy\(\)](#). The system calls this method on your activity as the final signal that your activity instance is being completely removed from the system memory.

Most apps don't need to implement this method because local class references are destroyed with the activity and your activity should perform most cleanup during [onPause\(\)](#) and [onStop\(\)](#). However, if your activity includes background threads that you created during [onCreate\(\)](#) or other long-running resources that could potentially leak memory if not properly closed, you should kill them during [onDestroy\(\)](#).

```

@Override
public void onDestroy() {
    super.onDestroy(); // Always call the superclass

    // Stop method tracing that the activity started during onCreate()
    android.os.Debug.stopMethodTracing();
}

```

}

Note: The system calls [onDestroy\(\)](#) after it has already called [onPause\(\)](#) and [onStop\(\)](#) in all situations except one: when you call [finish\(\)](#) from within the [onCreate\(\)](#) method. In some cases, such as when your activity operates as a temporary decision maker to launch another activity, you might call [finish\(\)](#) from within [onCreate\(\)](#) to destroy the activity. In this case, the system immediately calls [onDestroy\(\)](#) without calling any of the other lifecycle methods.

# Pausing and Resuming an Activity

During normal app use, the foreground activity is sometimes obstructed by other visual components that cause the activity to *pause*. For example, when a semi-transparent activity opens (such as one in the style of a dialog), the previous activity pauses. As long as the activity is still partially visible but currently not the activity in focus, it remains paused.

However, once the activity is fully-obstructed and not visible, it *stops* (which is discussed in the next lesson).

As your activity enters the paused state, the system calls the `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())) method on your `Activity` (</reference/android/app/Activity.html>), which allows you to stop ongoing actions that should not continue while paused (such as a video) or persist any information that should be permanently saved in case the user continues to leave your app. If the user returns to your activity from the paused state, the system resumes it and calls the `onResume()` ([/reference/android/app/Activity.html#onResume\(\)](/reference/android/app/Activity.html#onResume())) method.

Note: When your activity receives a call to `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())), it may be an indication that the activity will be paused for a moment and the user may return focus to your activity. However, it's usually the first indication that the user is leaving your activity.

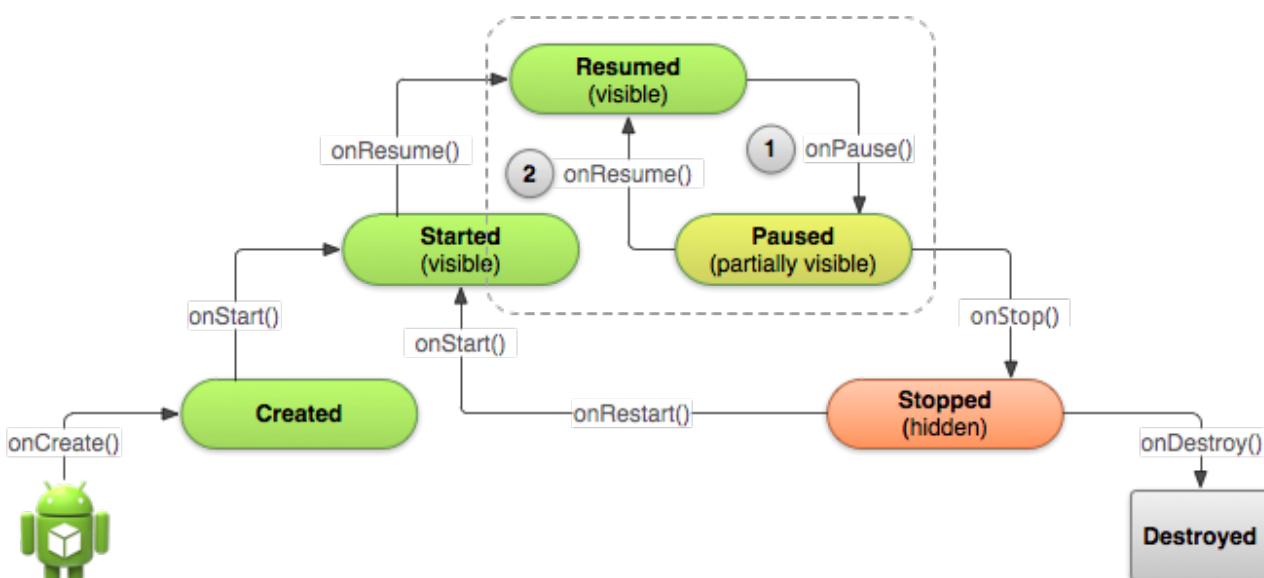


Figure 1. When a semi-transparent activity obscures your activity, the system calls `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())) and the activity waits in the Paused state (1). If the user returns to the activity while it's still paused, the system calls `onResume()` ([/reference/android/app/Activity.html#onResume\(\)](/reference/android/app/Activity.html#onResume())) (2).

## THIS LESSON TEACHES YOU TO

1. [Pause Your Activity](#)
2. [Resume Your Activity](#)

## YOU SHOULD ALSO READ

- [Activities](#)

## TRY IT OUT

[Download the demo](#)

ActivityLifecycle.zip

## Pause Your Activity

When the system calls `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())) for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity and it will soon enter the Stopped state. You should usually use the `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())) callback to:

- Stop animations or other ongoing actions that could consume CPU.
- Commit unsaved changes, but only if users expect such changes to be permanently saved when they leave (such as a draft email).
- Release system resources, such as broadcast receivers, handles to sensors (like GPS), or any resources that may affect battery life while your activity is paused and the user does not need them.

For example, if your application uses the `Camera` (</reference/android/hardware/Camera.html>), the `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())) method is a good place to release it.

```
@Override  
public void onPause() {  
    super.onPause(); // Always call the superclass method first  
  
    // Release the Camera because we don't need it when paused  
    // and other activities might need to use it.  
    if (mCamera != null) {  
        mCamera.release()  
        mCamera = null;  
    }  
}
```

Generally, you should not use `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())) to store user changes (such as personal information entered into a form) to permanent storage. The only time you should persist user changes to permanent storage within `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())) is when you're certain users expect the changes to be auto-saved (such as when drafting an email). However, you should avoid performing CPU-intensive work during `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())), such as writing to a database, because it can slow the visible transition to the next activity (you should instead perform heavy-load shutdown operations during `onStop()` ([/reference/android/app/Activity.html#onStop\(\)](/reference/android/app/Activity.html#onStop()))).

You should keep the amount of operations done in the `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](/reference/android/app/Activity.html#onPause())) method relatively simple in order to allow for a speedy transition to the user's next destination if your activity is actually being stopped.

Note: When your activity is paused, the `Activity` (</reference/android/app/Activity.html>) instance is kept resident in memory and is recalled when the activity resumes. You don't need to re-initialize components that were created during any of the callback methods leading up to the Resumed state.

## Resume Your Activity

When the user resumes your activity from the Paused state, the system calls the [onResume\(\)](#) ([/reference/android/app/Activity.html#onResume\(\)](#)) method.

Be aware that the system calls this method every time your activity comes into the foreground, including when it's created for the first time. As such, you should implement [onResume\(\)](#) ([/reference/android/app/Activity.html#onResume\(\)](#)) to initialize components that you release during [onPause\(\)](#) ([/reference/android/app/Activity.html#onPause\(\)](#)) and perform any other initializations that must occur each time the activity enters the Resumed state (such as begin animations and initialize components only used while the activity has user focus).

The following example of [onResume\(\)](#) ([/reference/android/app/Activity.html#onResume\(\)](#)) is the counterpart to the [onPause\(\)](#) ([/reference/android/app/Activity.html#onPause\(\)](#)) example above, so it initializes the camera that's released when the activity pauses.

```
@Override  
public void onResume() {  
    super.onResume(); // Always call the superclass method first  
  
    // Get the Camera instance as the activity achieves full user focus  
    if (mCamera == null) {  
        initializeCamera(); // Local method to handle camera init  
    }  
}
```

# Stopping and Restarting an Activity

Properly stopping and restarting your activity is an important process in the activity lifecycle that ensures your users perceive that your app is always alive and doesn't lose their progress. There are a few key scenarios in which your activity is stopped and restarted:

- The user opens the Recent Apps window and switches from your app to another app. The activity in your app that's currently in the foreground is stopped. If the user returns to your app from the Home screen launcher icon or the Recent Apps window, the activity restarts.
- The user performs an action in your app that starts a new activity. The current activity is stopped when the second activity is created. If the user then presses the *Back* button, the first activity is restarted.
- The user receives a phone call while using your app on his or her phone.

The [Activity \(/reference/android/app/Activity.html\)](#) class provides two lifecycle methods, [onStop\(\) \(/reference/android/app/Activity.html#onStop\(\)\)](#) and [onRestart\(\) \(/reference/android/app/Activity.html#onRestart\(\)\)](#), which allow you to specifically handle how your activity handles being stopped and restarted. Unlike the paused state, which identifies a partial UI obstruction, the stopped state guarantees that the UI is no longer visible and the user's focus is in a separate activity (or an entirely separate app).

Note: Because the system retains your [Activity \(/reference/android/app/Activity.html\)](#) instance in system memory when it is stopped, it's possible that you don't need to implement the [onStop\(\) \(/reference/android/app/Activity.html#onStop\(\)\)](#) and [onRestart\(\) \(/reference/android/app/Activity.html#onRestart\(\)\)](#) (or even [onStart\(\) \(/reference/android/app/Activity.html#onStart\(\)\)](#)) methods at all. For most activities that are relatively simple, the activity will stop and restart just fine and you might only need to use [onPause\(\) \(/reference/android/app/Activity.html#onPause\(\)\)](#) to pause ongoing actions and disconnect from system resources.

## THIS LESSON TEACHES YOU TO

1. [Stop Your Activity](#)
2. [Start/Rotate Your Activity](#)

## YOU SHOULD ALSO READ

- [Activities](#)

## TRY IT OUT

[Download the demo](#)

ActivityLifecycle.zip

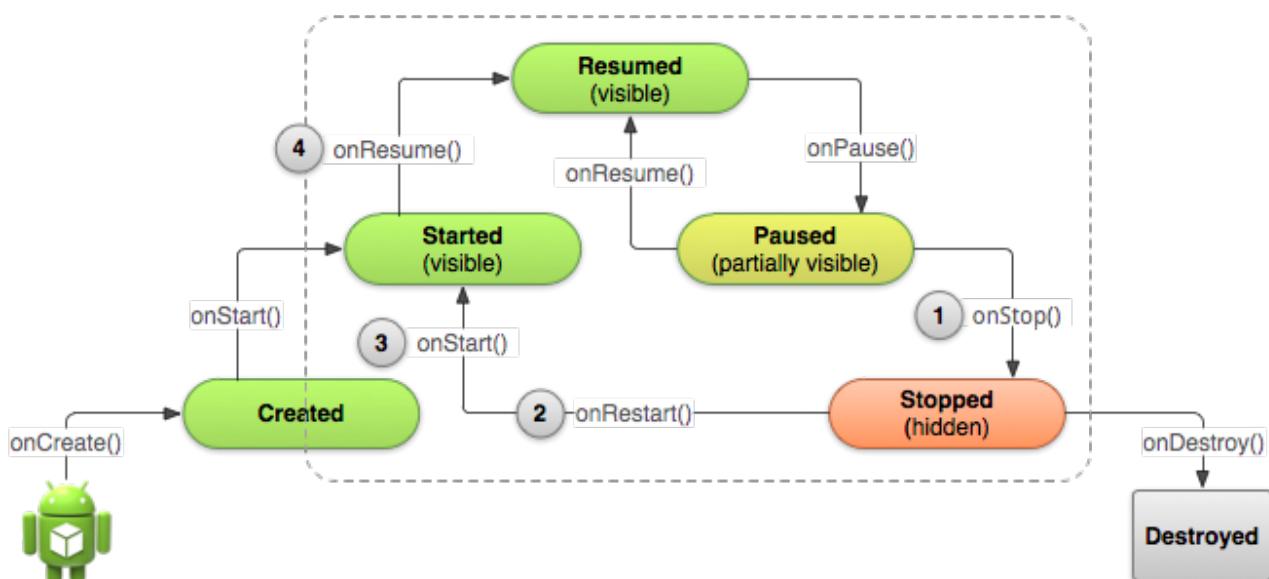


Figure 1. When the user leaves your activity, the system calls `onStop()` ([/reference/android/app/Activity.html#onStop\(\)](#)) to stop the activity (1). If the user returns while the activity is stopped, the system calls `onRestart()` ([/reference/android/app/Activity.html#onRestart\(\)](#)) (2), quickly followed by `onStart()` ([/reference/android/app/Activity.html#onStart\(\)](#)) (3) and `onResume()` ([/reference/android/app/Activity.html#onResume\(\)](#)) (4). Notice that no matter what scenario causes the activity to stop, the system always calls `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](#)) before calling `onStop()` ([/reference/android/app/Activity.html#onStop\(\)](#)).

## Stop Your Activity

When your activity receives a call to the `onStop()` ([/reference/android/app/Activity.html#onStop\(\)](#)) method, it's no longer visible and should release almost all resources that aren't needed while the user is not using it. Once your activity is stopped, the system might destroy the instance if it needs to recover system memory. In extreme cases, the system might simply kill your app process without calling the activity's final `onDestroy()` ([/reference/android/app/Activity.html#onDestroy\(\)](#)) callback, so it's important you use `onStop()` ([/reference/android/app/Activity.html#onStop\(\)](#)) to release resources that might leak memory.

Although the `onPause()` ([/reference/android/app/Activity.html#onPause\(\)](#)) method is called before `onStop()` ([/reference/android/app/Activity.html#onStop\(\)](#)), you should use `onStop()` ([/reference/android/app/Activity.html#onStop\(\)](#)) to perform larger, more CPU intensive shut-down operations, such as writing information to a database.

For example, here's an implementation of `onStop()` ([/reference/android/app/Activity.html#onStop\(\)](#)) that saves the contents of a draft note to persistent storage:

```

@Override
protected void onStop() {
    super.onStop(); // Always call the superclass method first

    // Save the note's current draft, because the activity is stopping
}

```

```
// and we want to be sure the current note progress isn't lost.  
ContentValues values = new ContentValues();  
values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());  
values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());  
  
getContentResolver().update(  
    mUri, // The URI for the note to update.  
    values, // The map of column names and new values to apply to them.  
    null, // No SELECT criteria are used.  
    null // No WHERE columns are used.  
);  
}
```

When your activity is stopped, the [Activity](#) ([/reference/android/app/Activity.html](#)) object is kept resident in memory and is recalled when the activity resumes. You don't need to re-initialize components that were created during any of the callback methods leading up to the Resumed state. The system also keeps track of the current state for each [View](#) ([/reference/android/view/View.html](#)) in the layout, so if the user entered text into an [EditText](#) ([/reference/android/widget/EditText.html](#)) widget, that content is retained so you don't need to save and restore it.

Note: Even if the system destroys your activity while it's stopped, it still retains the state of the [View](#) ([/reference/android/view/View.html](#)) objects (such as text in an [EditText](#) ([/reference/android/widget/EditText.html](#))) in a [Bundle](#) ([/reference/android/os/Bundle.html](#)) (a blob of key-value pairs) and restores them if the user navigates back to the same instance of the activity (the [next lesson](#) ([recreating.html](#)) talks more about using a [Bundle](#) ([/reference/android/os/Bundle.html](#)) to save other state data in case your activity is destroyed and recreated).

## Start/Restart Your Activity

---

When your activity comes back to the foreground from the stopped state, it receives a call to [onRestart\(\)](#) ([/reference/android/app/Activity.html#onRestart\(\)](#)). The system also calls the [onStart\(\)](#) ([/reference/android/app/Activity.html#onStart\(\)](#)) method, which happens every time your activity becomes visible (whether being restarted or created for the first time). The [onRestart\(\)](#) ([/reference/android/app/Activity.html#onRestart\(\)](#)) method, however, is called only when the activity resumes from the stopped state, so you can use it to perform special restoration work that might be necessary only if the activity was previously stopped, but not destroyed.

It's uncommon that an app needs to use [onRestart\(\)](#) ([/reference/android/app/Activity.html#onRestart\(\)](#)) to restore the activity's state, so there aren't any guidelines for this method that apply to the general population of apps. However, because your [onStop\(\)](#) ([/reference/android/app/Activity.html#onStop\(\)](#)) method should essentially clean up all your activity's resources, you'll need to re-instantiate them when the activity restarts. Yet, you also need to instantiate them when your activity is created for the first time (when there's no existing instance of the activity). For this reason, you should usually use the [onStart\(\)](#) ([/reference/android/app/Activity.html#onStart\(\)](#)) callback method as the counterpart to the [onStop\(\)](#) ([/reference/android/app/Activity.html#onStop\(\)](#)) method, because the system calls [onStart\(\)](#) ([/reference/android/app/Activity.html#onStart\(\)](#)) both when it creates your activity and when it restarts the activity from the stopped state.

For example, because the user might have been away from your app for a long time before coming back it, the [onStart\(\) \(/reference/android/app/Activity.html#onStart\(\)\)](#) method is a good place to verify that required system features are enabled:

```
@Override  
protected void onStart() {  
    super.onStart(); // Always call the superclass method first  
  
    // The activity is either being restarted or started for the first time  
    // so this is where we should make sure that GPS is enabled  
    LocationManager locationManager =  
        (LocationManager) getSystemService(Context.LOCATION_SERVICE);  
    boolean gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PR  
  
    if (!gpsEnabled) {  
        // Create a dialog here that requests the user to enable GPS, and use an  
        // with the android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS act  
        // to take the user to the Settings screen to enable GPS when they click  
    }  
}  
  
@Override  
protected void onRestart() {  
    super.onRestart(); // Always call the superclass method first  
  
    // Activity being restarted from stopped state  
}
```

When the system destroys your activity, it calls the [onDestroy\(\) \(/reference/android/app/Activity.html#onDestroy\(\)\)](#) method for your [Activity \(/reference/android/app/Activity.html\)](#). Because you should generally have released most of your resources with [onStop\(\) \(/reference/android/app/Activity.html#onStop\(\)\)](#), by the time you receive a call to [onDestroy\(\) \(/reference/android/app/Activity.html#onDestroy\(\)\)](#), there's not much that most apps need to do. This method is your last chance to clean out resources that could lead to a memory leak, so you should be sure that additional threads are destroyed and other long-running actions like method tracing are also stopped.

# Recreating an Activity

There are a few scenarios in which your activity is destroyed due to normal app behavior, such as when the user presses the *Back* button or your activity signals its own destruction by calling [finish\(\) \(/reference/android/app/Activity.html#finish\(\)\)](#). The system may also destroy your activity if it's currently stopped and hasn't been used in a long time or the foreground activity requires more resources so the system must shut down background processes to recover memory.

When your activity is destroyed because the user presses *Back* or the activity finishes itself, the system's concept of that [Activity \(/reference/android/app/Activity.html\)](#) instance is gone forever because the behavior indicates the activity is no longer needed. However, if the system destroys the activity due to system constraints (rather than normal app behavior), then although the actual [Activity \(/reference/android/app/Activity.html\)](#) instance is gone, the system remembers that it existed such that if the user navigates back to it, the system creates a new instance of the activity using a set of saved data that describes the state of the activity when it was destroyed. The saved data that the system uses to restore the previous state is called the "instance state" and is a collection of key-value pairs stored in a [Bundle \(/reference/android/os/Bundle.html\)](#) object.

**Caution:** Your activity will be destroyed and recreated each time the user rotates the screen. When the screen changes orientation, the system destroys and recreates the foreground activity because the screen configuration has changed and your activity might need to load alternative resources (such as the layout).

By default, the system uses the [Bundle \(/reference/android/os/Bundle.html\)](#) instance state to save information about each [View \(/reference/android/view/View.html\)](#) object in your activity layout (such as the text value entered into an [EditText \(/reference/android/widget/EditText.html\)](#) object). So, if your activity instance is destroyed and recreated, the state of the layout is restored to its previous state with no code required by you. However, your activity might have more state information that you'd like to restore, such as member variables that track the user's progress in the activity.

**Note:** In order for the Android system to restore the state of the views in your activity, each view must have a unique ID, supplied by the [android:id \(/reference/android/view/View.html#attr\\_android:id\)](#) attribute.

To save additional data about the activity state, you must override the [onSaveInstanceState\(\) \(/reference/android/app/Activity.html#onSaveInstanceState\(android.os.Bundle\)\)](#) callback method. The system calls this method when the user is leaving your activity and passes it the [Bundle \(/reference/android/os/Bundle.html\)](#) object that will be saved in the event that your activity is destroyed unexpectedly. If the system must recreate the activity instance later, it passes the same [Bundle \(/reference/android/os/Bundle.html\)](#) object to both the [onRestoreInstanceState\(\) \(/reference/android/app/Activity.html#onRestoreInstanceState\(android.os.Bundle\)\)](#) and [onCreate\(\) \(/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)\)](#) methods.

## THIS LESSON TEACHES YOU TO

1. [Save Your Activity State](#)
2. [Restore Your Activity State](#)

## YOU SHOULD ALSO READ

- [Supporting Different Screens](#)
- [Handling Runtime Changes](#)
- [Activities](#)



Figure 2. As the system begins to stop your activity, it calls [onSaveInstanceState\(\)](#) ([/reference/android/app/Activity.html#onSaveInstanceState\(android.os.Bundle\)](#)) (1) so you can specify additional state data you'd like to save in case the [Activity](#) ([/reference/android/app/Activity.html](#)) instance must be recreated. If the activity is destroyed and the same instance must be recreated, the system passes the state data defined at (1) to both the [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method (2) and the [onRestoreInstanceState\(\)](#) ([/reference/android/app/Activity.html#onRestoreInstanceState\(android.os.Bundle\)](#)) method (3).

## Save Your Activity State

As your activity begins to stop, the system calls [onSaveInstanceState\(\)](#) ([/reference/android/app/Activity.html#onSaveInstanceState\(android.os.Bundle\)](#)) so your activity can save state information with a collection of key-value pairs. The default implementation of this method saves information about the state of the activity's view hierarchy, such as the text in an [EditText](#) ([/reference/android/widget/EditText.html](#)) widget or the scroll position of a [ListView](#) ([/reference/android/widget/ListView.html](#)).

To save additional state information for your activity, you must implement [onSaveInstanceState\(\)](#) ([/reference/android/app/Activity.html#onSaveInstanceState\(android.os.Bundle\)](#)) and add key-value pairs to the [Bundle](#) ([/reference/android/os/Bundle.html](#)) object. For example:

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);

    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

Caution: Always call the superclass implementation of [onSaveInstanceState\(\)](#) ([/reference/android/app/Activity.html#onSaveInstanceState\(android.os.Bundle\)](#)) so the default implementation can save the state of the view hierarchy.

## Restore Your Activity State

When your activity is recreated after it was previously destroyed, you can recover your saved state from the [Bundle](#) ([/reference/android/os/Bundle.html](#)) that the system passes your activity. Both the [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) and [onRestoreInstanceState\(\)](#) ([/reference/android/app/Activity.html#onRestoreInstanceState\(android.os.Bundle\)](#)) callback methods receive the same [Bundle](#) ([/reference/android/os/Bundle.html](#)) that contains the instance state information.

Because the [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method is called whether the system is creating a new instance of your activity or recreating a previous one, you must check whether the state [Bundle](#) ([/reference/android/os/Bundle.html](#)) is null before you attempt to read it. If it is null, then the system is creating a new instance of the activity, instead of restoring a previous one that was destroyed.

For example, here's how you can restore some state data in [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)):

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState); // Always call the superclass first  
  
    // Check whether we're recreating a previously destroyed instance  
    if (savedInstanceState != null) {  
        // Restore value of members from saved state  
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);  
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);  
    } else {  
        // Probably initialize members with default values for a new instance  
    }  
    ...  
}
```

Instead of restoring the state during [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) you may choose to implement [onRestoreInstanceState\(\)](#) ([/reference/android/app/Activity.html#onRestoreInstanceState\(android.os.Bundle\)](#)), which the system calls after the [onStart\(\)](#) ([/reference/android/app/Activity.html#onStart\(\)](#)) method. The system calls [onRestoreInstanceState\(\)](#) ([/reference/android/app/Activity.html#onRestoreInstanceState\(android.os.Bundle\)](#)) only if there is a saved state to restore, so you do not need to check whether the [Bundle](#) ([/reference/android/os/Bundle.html](#)) is null:

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
```

```
// Always call the superclass so it can restore the view hierarchy  
super.onRestoreInstanceState(savedInstanceState);  
  
// Restore state members from saved instance  
mCurrentScore = savedInstanceState.getInt(STATE_SCORE);  
mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);  
}
```

Caution: Always call the superclass implementation of [onRestoreInstanceState\(\)](#) ([/reference/android/app/Activity.html#onRestoreInstanceState\(android.os.Bundle\)](#)) so the default implementation can restore the state of the view hierarchy.

To learn more about recreating your activity due to a restart event at runtime (such as when the screen rotates), read [Handling Runtime Changes](#) ([/guide/topics/resources/runtime-changes.html](#)).

# Supporting Different Devices

Android devices come in many shapes and sizes all around the world. With a wide range of device types, you have an opportunity to reach a huge audience with your app. In order to be as successful as possible on Android, your app needs to adapt to various device configurations. Some of the important variations that you should consider include different languages, screen sizes, and versions of the Android platform.

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).

## Lessons

---

### [Supporting Different Languages](#)

Learn how to support multiple languages with alternative string resources.

### [Supporting Different Screens](#)

Learn how to optimize the user experience for different screen sizes and densities.

### [Supporting Different Platform Versions](#)

Learn how to use APIs available in new versions of Android while continuing to support older versions of Android.

---

#### **DEPENDENCIES AND PREREQUISITES**

---

- Android 1.6 or higher

---

#### **YOU SHOULD ALSO READ**

---

- [Application Resources](#)
- [Designing for Multiple Screens](#)

# Supporting Different Languages

It's always a good practice to extract UI strings from your app code and keep them in an external file. Android makes this easy with a resources directory in each Android project.

If you created your project using the Android SDK Tools (read [Creating an Android Project \(/training/basics/firstapp/creating-project.html\)](#)), the tools create a res/ directory in the top level of the project. Within this res/ directory are subdirectories for various resource types. There are also a few default files such as res/values/strings.xml, which holds your string values.

## THIS CLASS TEACHES YOU TO

1. [Create Locale Directories and String Files](#)
2. [Use the String Resources](#)

## YOU SHOULD ALSO READ

- [Localization](#)

## Create Locale Directories and String Files

To add support for more languages, create additional values directories inside res/ that include a hyphen and the ISO country code at the end of the directory name. For example, values-es/ is the directory containing simple resources for the Locales with the language code "es". Android loads the appropriate resources according to the locale settings of the device at run time.

Once you've decided on the languages you will support, create the resource subdirectories and string resource files. For example:

```
MyProject/
  res/
    values/
      strings.xml
    values-es/
      strings.xml
    values-fr/
      strings.xml
```

Add the string values for each locale into the appropriate file.

At runtime, the Android system uses the appropriate set of string resources based on the locale currently set for the user's device.

For example, the following are some different string resource files for different languages.

English (default locale), /values/strings.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="title">My Application</string>
  <string name="hello_world">Hello World!</string>
```

```
</resources>
```

Spanish, /values-es/strings.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="title">Mi Aplicación</string>
    <string name="hello_world">Hola Mundo!</string>
</resources>
```

French, /values-fr/strings.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="title">Mon Application</string>
    <string name="hello_world">Bonjour le monde !</string>
</resources>
```

Note: You can use the locale qualifier (or any configuration qualifier) on any resource type, such as if you want to provide localized versions of your bitmap drawable. For more information, see [Localization \(/guide/topics/resources/localization.html\)](#).

## Use the String Resources

You can reference your string resources in your source code and other XML files using the resource name defined by the `<string>` element's name attribute.

In your source code, you can refer to a string resource with the syntax `R.string.<string_name>`. There are a variety of methods that accept a string resource this way.

For example:

```
// Get a string resource from your app's Resources
String hello = getResources().getString(R.string.hello_world);

// Or supply a string resource to a method that requires a string
TextView textView = new TextView(this);
textView.setText(R.string.hello_world);
```

In other XML files, you can refer to a string resource with the syntax `@string/<string_name>` whenever the XML attribute accepts a string value.

For example:

```
<TextView
```

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

# Supporting Different Screens

Android categorizes device screens using two general properties: size and density. You should expect that your app will be installed on devices with screens that range in both size and density. As such, you should include some alternative resources that optimize your app's appearance for different screen sizes and densities.

- There are four generalized sizes: small, normal, large, xlarge
- And four generalized densities: low (ldpi), medium (mdpi), high (hdpi), extra high (xhdpi)

To declare different layouts and bitmaps you'd like to use for different screens, you must place these alternative resources in separate directories, similar to how you do for different language strings.

Also be aware that the screens orientation (landscape or portrait) is considered a variation of screen size, so many apps should revise the layout to optimize the user experience in each orientation.

## THIS LESSON TEACHES YOU TO

1. [Create Different Layouts](#)
2. [Create Different Bitmaps](#)

## YOU SHOULD ALSO READ

- [Designing for Multiple Screens](#)
- [Supporting Multiple Screens](#)
- [Iconography design guide](#)

## Create Different Layouts

To optimize your user experience on different screen sizes, you should create a unique layout XML file for each screen size you want to support. Each layout should be saved into the appropriate resources directory, named with a -<screen\_size> suffix. For example, a unique layout for large screens should be saved under res/layout-large/.

Note: Android automatically scales your layout in order to properly fit the screen. Thus, your layouts for different screen sizes don't need to worry about the absolute size of UI elements but instead focus on the layout structure that affects the user experience (such as the size or position of important views relative to sibling views).

For example, this project includes a default layout and an alternative layout for *large* screens:

```
MyProject/
    res/
        layout/
            main.xml
        layout-large/
            main.xml
```

The file names must be exactly the same, but their contents are different in order to provide an optimized UI for the corresponding screen size.

Simply reference the layout file in your app as usual:

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

The system loads the layout file from the appropriate layout directory based on screen size of the device on which your app is running. More information about how Android selects the appropriate resource is available in the [Providing Resources \(/guide/topics/resources/providing-resources.html#BestMatch\)](#) guide.

As another example, here's a project with an alternative layout for landscape orientation:

```
MyProject/  
    res/  
        layout/  
            main.xml  
        layout-land/  
            main.xml
```

By default, the `layout/main.xml` file is used for portrait orientation.

If you want to provide a special layout for landscape, including while on large screens, then you need to use both the `large` and `land` qualifier:

```
MyProject/  
    res/  
        layout/          # default (portrait)  
            main.xml  
        layout-land/     # landscape  
            main.xml  
        layout-large/    # large (portrait)  
            main.xml  
        layout-large-land/ # large landscape  
            main.xml
```

Note: Android 3.2 and above supports an advanced method of defining screen sizes that allows you to specify resources for screen sizes based on the minimum width and height in terms of density-independent pixels. This lesson does not cover this new technique. For more information, read [Designing for Multiple Screens \(/training/multiscreen/index.html\)](#).

## Create Different Bitmaps

You should always provide bitmap resources that are properly scaled to each of the generalized density buckets: low, medium, high and extra-high density. This helps you achieve good graphical quality and performance on all screen densities.

To generate these images, you should start with your raw resource in vector format and generate the images for each density using the following size scale:

- xhdpi: 2.0
- hdpi: 1.5
- mdpi: 1.0 (baseline)
- ldpi: 0.75

This means that if you generate a 200x200 image for xhdpi devices, you should generate the same resource in 150x150 for hdpi, 100x100 for mdpi, and 75x75 for ldpi devices.

Then, place the files in the appropriate drawable resource directory:

```
MyProject/  
    res/  
        drawable-xhdpi/  
            awesomeimage.png  
        drawable-hdpi/  
            awesomeimage.png  
        drawable-mdpi/  
            awesomeimage.png  
        drawable-ldpi/  
            awesomeimage.png
```

Any time you reference @drawable/awesomeimage, the system selects the appropriate bitmap based on the screen's density.

Note: Low-density (ldpi) resources aren't always necessary. When you provide hdpi assets, the system scales them down by one half to properly fit ldpi screens.

For more tips and guidelines about creating icon assets for your app, see the [Iconography design guide \(/design/style/iconography.html\)](#).

# Supporting Different Platform Versions

While the latest versions of Android often provide great APIs for your app, you should continue to support older versions of Android until more devices get updated. This lesson shows you how to take advantage of the latest APIs while continuing to support older versions as well.

The dashboard for [Platform Versions](#) (<http://developer.android.com/about/dashboards/index.html>) is updated regularly to show the distribution of active devices running each version of Android, based on the number of devices that visit the Google Play Store. Generally, it's a good practice to support about 90% of the active devices, while targeting your app to the latest version.

**Tip:** In order to provide the best features and functionality across several Android versions, you should use the [Android Support Library](#) ([/tools/extras/support-library.html](http://tools/extras/support-library.html)) in your app, which allows you to use several recent platform APIs on older versions.

## THIS LESSON TEACHES YOU TO

1. [Specify Minimum and Target API Levels](#)
2. [Check System Version at Runtime](#)
3. [Use Platform Styles and Themes](#)

## YOU SHOULD ALSO READ

- [Android API Levels](#)
- [Android Support Library](#)

## Specify Minimum and Target API Levels

The [AndroidManifest.xml](#) ([/guide/topics/manifest/manifest-intro.html](http://guide/topics/manifest/manifest-intro.html)) file describes details about your app and identifies which versions of Android it supports. Specifically, the `minSdkVersion` and `targetSdkVersion` attributes for the [uses-sdk](#) ([/guide/topics/manifest/uses-sdk-element.html](http://guide/topics/manifest/uses-sdk-element.html)) element identify the lowest API level with which your app is compatible and the highest API level against which you've designed and tested your app.

For example:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" ... >
  <uses-sdk android:minSdkVersion="4" android:targetSdkVersion="15" />
  ...
</manifest>
```

As new versions of Android are released, some style and behaviors may change. To allow your app to take advantage of these changes and ensure that your app fits the style of each user's device, you should set the [targetSdkVersion](#) ([/guide/topics/manifest/uses-sdk-element.html#target](http://guide/topics/manifest/uses-sdk-element.html#target)) value to match the latest Android version available.

## Check System Version at Runtime

Android provides a unique code for each platform version in the [Build](#) ([/reference/android](http://reference/android)

`/os/Build.html` constants class. Use these codes within your app to build conditions that ensure the code that depends on higher API levels is executed only when those APIs are available on the system.

```
private void setUpActionBar() {  
    // Make sure we're running on Honeycomb or higher to use ActionBar APIs  
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {  
        ActionBar actionBar = getActionBar();  
        actionBar.setDisplayHomeAsUpEnabled(true);  
    }  
}
```

Note: When parsing XML resources, Android ignores XML attributes that aren't supported by the current device. So you can safely use XML attributes that are only supported by newer versions without worrying about older versions breaking when they encounter that code. For example, if you set the `targetSdkVersion="11"`, your app includes the [ActionBar \(/reference/android/app/ActionBar.html\)](#) by default on Android 3.0 and higher. To then add menu items to the action bar, you need to set `android:showAsAction="ifRoom"` in your menu resource XML. It's safe to do this in a cross-version XML file, because the older versions of Android simply ignore the `showAsAction` attribute (that is, you *do not* need a separate version in `res/menu-v11/`).

## Use Platform Styles and Themes

Android provides user experience themes that give apps the look and feel of the underlying operating system. These themes can be applied to your app within the manifest file. By using these built in styles and themes, your app will naturally follow the latest look and feel of Android with each new release.

To make your activity look like a dialog box:

```
<activity android:theme="@android:style/Theme.Dialog">
```

To make your activity have a transparent background:

```
<activity android:theme="@android:style/Theme.Translucent">
```

To apply your own custom theme defined in `/res/values/styles.xml`:

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

To apply a theme to your entire app (all activities), add the `android:theme` attribute to the [`<application>` \(/guide/topics/manifest/application-element.html\)](#) element:

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

For more about creating and using themes, read the [Styles and Themes \(/guide/topics/ui/themes.html\)](#) guide.

# Building a Dynamic UI with Fragments

To create a dynamic and multi-pane user interface on Android, you need to encapsulate UI components and activity behaviors into modules that you can swap into and out of your activities. You can create these modules with the [Fragment](#) (</reference/android/app/Fragment.html>) class, which behaves somewhat like a nested activity that can define its own layout and manage its own lifecycle.

When a fragment specifies its own layout, it can be configured in different combinations with other fragments inside an activity to modify your layout configuration for different screen sizes (a small screen might show one fragment at a time, but a large screen can show two or more).

This class shows you how to create a dynamic user experience with fragments and optimize your app's user experience for devices with different screen sizes, all while continuing to support devices running versions as old as Android 1.6.

## DEPENDENCIES AND PREREQUISITES

- Basic knowledge of the Activity lifecycle (see [Managing the Activity Lifecycle](#))
- Experience building [XML layouts](#)

## YOU SHOULD ALSO READ

- [Fragments](#)
- [Supporting Tablets and Handsets](#)

## TRY IT OUT

[Download the sample](#)

FragmentBasics.zip

## Lessons

### [\*\*Using the Android Support Library\*\*](#)

Learn how to use more recent framework APIs in earlier versions of Android by bundling the Android Support Library into your app.

### [\*\*Creating a Fragment\*\*](#)

Learn how to build a fragment and implement basic behaviors within its callback methods.

### [\*\*Building a Flexible UI\*\*](#)

Learn how to build your app with layouts that provide different fragment configurations for different screens.

### [\*\*Communicating with Other Fragments\*\*](#)

Learn how to set up communication paths from a fragment to the activity and other fragments.

# Using the Support Library

The Android [Support Library](#) ([/tools/extras/support-library.html](#)) provides a JAR file with an API library that allow you to use some of the more recent Android APIs in your app while running on earlier versions of Android. For instance, the Support Library provides a version of the [Fragment](#) ([/reference/android/app/Fragment.html](#)) APIs that you can use on Android 1.6 (API level 4) and higher.

This lesson shows how to set up your app to use the Support Library in order to use fragments to build a dynamic app UI.

## THIS LESSON TEACHES YOU TO

1. [Set Up Your Project With the Support Library](#)
2. [Import the Support Library APIs](#)

## YOU SHOULD ALSO READ

- [Support Library](#)

## Set Up Your Project With the Support Library

To set up your project:

1. Download the Android Support package using the SDK Manager
2. Create a `libs` directory at the top level of your Android project.
3. Locate the JAR file for the library you want to use and copy it into the `libs/` directory.

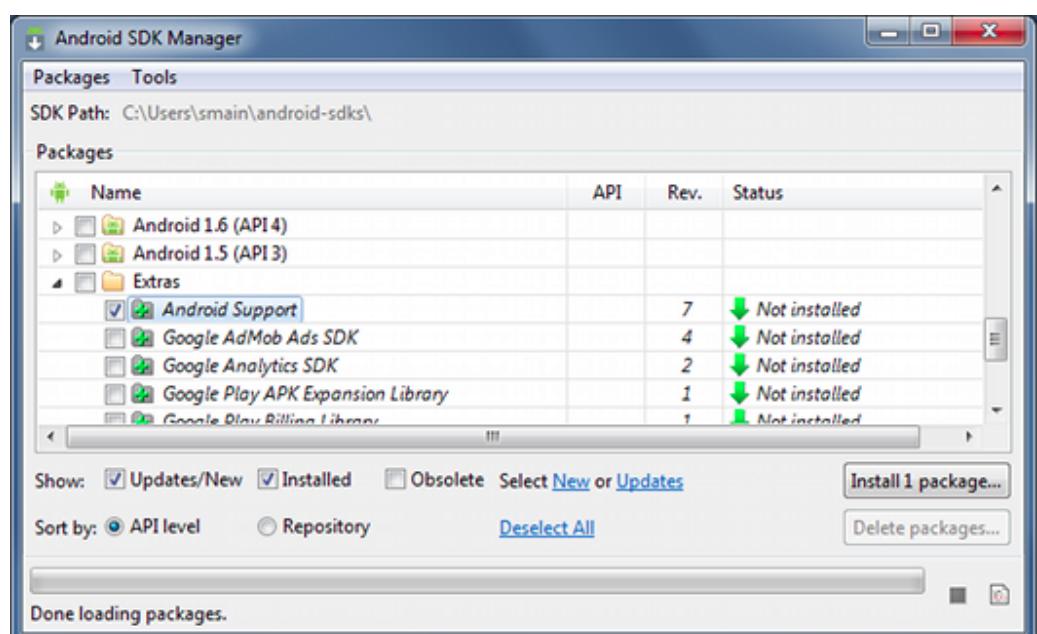


Figure 1. The Android SDK Manager with the Android Support package selected.

For example, the library that supports API level 4 and up is located at `<sdk>/extras/android/support/v4/android-support-v4.jar`.

4. Update your manifest file to set the minimum API level to 4 and the target API level to the latest release:

```
<uses-sdk android:minSdkVersion="4" android:targetSdkVersion="15" />
```

## Import the Support Library APIs

---

The Support Library includes a variety of APIs that were either added in recent versions of Android or don't exist in the platform at all and merely provide additional support to you when developing specific application features.

You can find all the API reference documentation for the Support Library in the platform docs at [android.support.v4.\\* \(/reference/android/support/v4/app/package-summary.html\)](#).

Warning: To be sure that you don't accidentally use new APIs on an older system version, be certain that you import the [Fragment \(/reference/android/support/v4/app/Fragment.html\)](#) class and related APIs from the [android.support.v4.app \(/reference/android/support/v4/app/package-summary.html\)](#) package:

```
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
...
```

When creating an activity that hosts fragments while using the Support Library, you must also extend the [FragmentActivity \(/reference/android/support/v4/app/FragmentActivity.html\)](#) class instead of the traditional [Activity \(/reference/android/app/Activity.html\)](#) class. You'll see sample code for the fragment and activity in the next lesson.

# Creating a Fragment

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). This lesson shows how to extend the [Fragment \(/reference/android/support/v4/app/Fragment.html\)](#) class using the Support Library so your app remains compatible with devices running system versions as old as Android 1.6.

Note: If you decide for other reasons that the minimum API level your app requires is 11 or higher, you don't need to use the Support Library and can instead use the framework's built in [Fragment \(/reference/android/app/Fragment.html\)](#) class and related APIs. Just be aware that this lesson is focused on using the APIs from the Support Library, which use a specific package signature and sometimes slightly different API names than the versions included in the platform.

## THIS LESSON TEACHES YOU TO

1. [Create a Fragment Class](#)
2. [Add a Fragment to an Activity using XML](#)

## YOU SHOULD ALSO READ

- [Fragments](#)

## TRY IT OUT

[Download the sample](#)

FragmentBasics.zip

## Create a Fragment Class

To create a fragment, extend the [Fragment \(/reference/android/support/v4/app/Fragment.html\)](#) class, then override key lifecycle methods to insert your app logic, similar to the way you would with an [Activity \(/reference/android/app/Activity.html\)](#) class.

One difference when creating a [Fragment \(/reference/android/support/v4/app/Fragment.html\)](#) is that you must use the [onCreateView\(\) \(/reference/android/support/v4/app/Fragment.html#onCreateView\(android.view.LayoutInflater, android.view.ViewGroup, android.os.Bundle\)\)](#) callback to define the layout. In fact, this is the only callback you need in order to get a fragment running. For example, here's a simple fragment that specifies its own layout:

```
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.ViewGroup;

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

```
}
```

Just like an activity, a fragment should implement other lifecycle callbacks that allow you to manage its state as it is added or removed from the activity and as the activity transitions between its lifecycle states. For instance, when the activity's [onPause\(\) \(/reference/android/app/Activity.html#onPause\(\)\)](#) method is called, any fragments in the activity also receive a call to [onPause\(\) \(/reference/android/support/v4/app/Fragment.html#onPause\(\)\)](#).

More information about the fragment lifecycle and callback methods is available in the [Fragments \(/guide/components/fragments.html\)](#) developer guide.

## Add a Fragment to an Activity using XML

While fragments are reusable, modular UI components, each instance of a [Fragment \(/reference/android/support/v4/app/Fragment.html\)](#) class must be associated with a parent [FragmentActivity \(/reference/android/support/v4/app/FragmentActivity.html\)](#). You can achieve this association by defining each fragment within your activity layout XML file.

Note: [FragmentActivity \(/reference/android/support/v4/app/FragmentActivity.html\)](#) is a special activity provided in the Support Library to handle fragments on system versions older than API level 11. If the lowest system version you support is API level 11 or higher, then you can use a regular [Activity \(/reference/android/app/Activity.html\)](#).

Here is an example layout file that adds two fragments to an activity when the device screen is considered "large" (specified by the `large` qualifier in the directory name).

res/layout-large/news\_articles.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="horizontal"  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent">  
  
    <fragment android:name="com.example.android.fragments.HeadlinesFragment"  
        android:id="@+id/headlines_fragment"  
        android:layout_weight="1"  
        android:layout_width="0dp"  
        android:layout_height="match_parent" />  
  
    <fragment android:name="com.example.android.fragments.ArticleFragment"  
        android:id="@+id/article_fragment"  
        android:layout_weight="2"  
        android:layout_width="0dp"  
        android:layout_height="match_parent" />  
  
</LinearLayout>
```

Tip: For more information about creating layouts for different screen sizes, read [Supporting Different Screen Sizes \(/training/multiscreen/screensizes.html\)](#).

Here's how an activity applies this layout:

```
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

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

Note: When you add a fragment to an activity layout by defining the fragment in the layout XML file, you *cannot* remove the fragment at runtime. If you plan to swap your fragments in and out during user interaction, you must add the fragment to the activity when the activity first starts, as shown in the next lesson.

# Building a Flexible UI

When designing your application to support a wide range of screen sizes, 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 device it might be appropriate to display just one fragment at a time for a single-pane user interface. Conversely, you may want to set fragments side-by-side on a tablet which has a wider screen size to display more information to the user.

## THIS LESSON TEACHES YOU TO

1. [Add a Fragment to an Activity at Runtime](#)
2. [Replace One Fragment with Another](#)

## YOU SHOULD ALSO READ

- [Fragments](#)
- [Supporting Tablets and Handsets](#)

## TRY IT OUT

[Download the sample](#)

FragmentBasics.zip

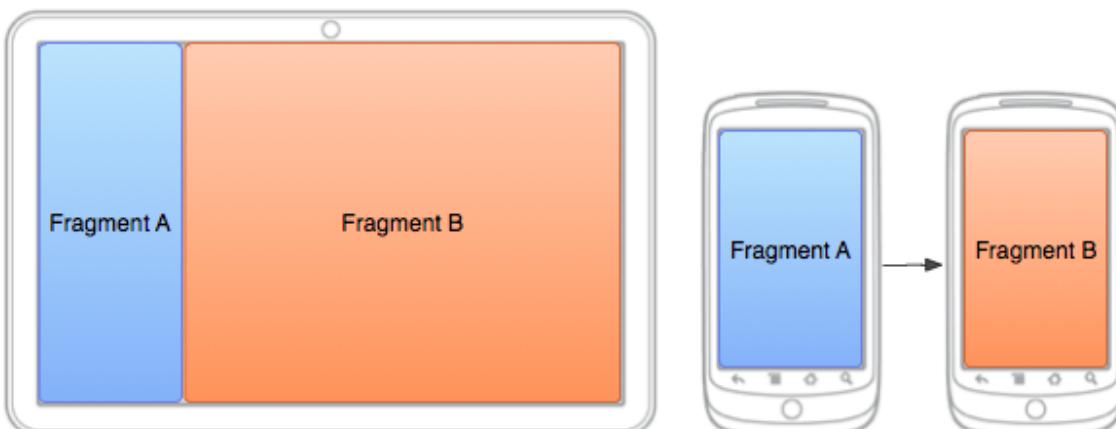


Figure 1. Two fragments, displayed in different configurations for the same activity on different screen sizes. On a large screen, both fragments fit side by side, but on a handset device, only one fragment fits at a time so the fragments must replace each other as the user navigates.

The [FragmentManager](#) (</reference/android/support/v4/app/FragmentManager.html>) class provides methods that allow you to add, remove, and replace fragments to an activity at runtime in order to create a dynamic experience.

## Add a Fragment to an Activity at Runtime

Rather than defining the fragments for an activity in the layout file—as shown in the [previous lesson \(creating.html\)](#) with the `<fragment>` element—you can add a fragment to the activity during the activity runtime. This is necessary if you plan to change fragments during the life of the

activity.

To perform a transaction such as add or remove a fragment, you must use the [FragmentManager](#) ([/reference/android/support/v4/app/FragmentManager.html](#)) to create a [FragmentTransaction](#) ([/reference/android/support/v4/app/FragmentTransaction.html](#)), which provides APIs to add, remove, replace, and perform other fragment transactions.

If your activity allows the fragments to be removed and replaced, you should add the initial fragment(s) to the activity during the activity's [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method.

An important rule when dealing with fragments—especially those that you add at runtime—is that the fragment must have a container [View](#) ([/reference/android/view/View.html](#)) in the layout in which the fragment's layout will reside.

The following layout is an alternative to the layout shown in the [previous lesson \(creating.html\)](#) that shows only one fragment at a time. In order to replace one fragment with another, the activity's layout includes an empty [FrameLayout](#) ([/reference/android/widget/FrameLayout.html](#)) that acts as the fragment container.

Notice that the filename is the same as the layout file in the previous lesson, but the layout directory does *not* have the `large` qualifier, so this layout is used when the device screen is smaller than `large` because the screen does not fit both fragments at the same time.

`res/layout/news_articles.xml:`

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

Inside your activity, call [getSupportFragmentManager\(\)](#) ([/reference/android/support/v4/app/FragmentActivity.html#getSupportFragmentManager\(\)](#)) to get a [FragmentManager](#) ([/reference/android/support/v4/app/FragmentManager.html](#)) using the Support Library APIs. Then call [beginTransaction\(\)](#) ([/reference/android/support/v4/app/FragmentManager.html#beginTransaction\(\)](#)) to create a [FragmentTransaction](#) ([/reference/android/support/v4/app/FragmentTransaction.html](#)) and call [add\(\)](#) ([/reference/android/support/v4/app/FragmentTransaction.html#add\(android.support.v4.app.Fragment, java.lang.String\)](#)) to add a fragment.

You can perform multiple fragment transaction for the activity using the same [FragmentTransaction](#) ([/reference/android/support/v4/app/FragmentTransaction.html](#)). When you're ready to make the changes, you must call [commit\(\)](#) ([/reference/android/support/v4/app/FragmentTransaction.html#commit\(\)](#)).

For example, here's how to add a fragment to the previous layout:

```
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
```

```
public class MainActivity extends FragmentActivity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.news_articles);  
  
        // Check that the activity is using the layout version with  
        // the fragment_container FrameLayout  
        if (findViewById(R.id.fragment_container) != null) {  
  
            // However, if we're being restored from a previous state,  
            // then we don't need to do anything and should return or else  
            // we could end up with overlapping fragments.  
            if (savedInstanceState != null) {  
                return;  
            }  
  
            // Create an instance of ExampleFragment  
            HeadlinesFragment firstFragment = new HeadlinesFragment();  
  
            // In case this activity was started with special instructions from a  
            // pass the Intent's extras to the fragment as arguments  
            firstFragment.setArguments(getIntent().getExtras());  
  
            // Add the fragment to the 'fragment_container' FrameLayout  
            getSupportFragmentManager().beginTransaction()  
                .add(R.id.fragment_container, firstFragment).commit();  
        }  
    }  
}
```

Because the fragment has been added to the [FrameLayout](#) ([/reference/android/widget/FrameLayout.html](#)) container at runtime—instead of defining it in the activity's layout with a <fragment> element—the activity can remove the fragment and replace it with a different one.

## Replace One Fragment with Another

The procedure to replace a fragment is similar to adding one, but requires the [replace\(\)](#) ([/reference/android/support/v4/app/FragmentTransaction.html#replace\(int, android.support.v4.app.Fragment\)](#)) method instead of [add\(\)](#) ([/reference/android/support/v4/app/FragmentTransaction.html#add\(android.support.v4.app.Fragment, java.lang.String\)](#)).

Keep in mind that when you perform fragment transactions, such as replace or remove one, it's often appropriate to allow the user to navigate backward and "undo" the change. To allow the user to navigate backward through the fragment transactions, you must call [addToBackStack\(\)](#) ([/reference/android/support/v4/app/FragmentTransaction.html#addToBackStack\(java.lang.String\)](#)) before you commit the [FragmentTransaction](#) ([/reference/android/support/v4/app/FragmentTransaction.html](#)).

Note: When you remove or replace a fragment and add the transaction to the back stack, the fragment that is removed is stopped (not destroyed). If the user navigates back to restore the fragment, it restarts. If you *do not* add the transaction to the back stack, then the fragment is destroyed when removed or replaced.

Example of replacing one fragment with another:

```
// Create fragment and give it an argument specifying the article it should show
ArticleFragment newFragment = new ArticleFragment();
Bundle args = new Bundle();
args.putInt(ArticleFragment.ARG_POSITION, position);
newFragment.setArguments(args);

FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();

// Replace whatever is in the fragment_container view with this fragment,
// and add the transaction to the back stack so the user can navigate back
transaction.replace(R.id.fragment_container, newFragment);
transaction.addToBackStack(null);

// Commit the transaction
transaction.commit();
```

The [addBackStack\(\)](#) ([/reference/android/support/v4/app/FragmentTransaction.html#addBackStack\(java.lang.String\)](#)) method takes an optional string parameter that specifies a unique name for the transaction. The name isn't needed unless you plan to perform advanced fragment operations using the [FragmentManager.BackStackEntry](#) ([/reference/android/support/v4/app/FragmentManager.BackStackEntry.html](#)) APIs.

# Communicating with Other Fragments

In order to reuse the Fragment UI components, you should build each as a completely self-contained, modular component that defines its own layout and behavior. Once you have defined these reusable Fragments, you can associate them with an Activity and connect them with the application logic to realize the overall composite UI.

Often you will want one Fragment to communicate with another, for example to change the content based on a user event. All Fragment-to-Fragment communication is done through the associated Activity. Two Fragments should never communicate directly.

## Define an Interface

To allow a Fragment to communicate up to its Activity, you can define an interface in the Fragment class and implement it within the Activity. The Fragment captures the interface implementation during its `onAttach()` lifecycle method and can then call the Interface methods in order to communicate with the Activity.

Here is an example of Fragment to Activity communication:

```
public class HeadlinesFragment extends ListFragment {  
    OnHeadlineSelectedListener mCallback;  
  
    // Container Activity must implement this interface  
    public interface OnHeadlineSelectedListener {  
        public void onArticleSelected(int position);  
    }  
  
    @Override  
    public void onAttach(Activity activity) {  
        super.onAttach(activity);  
  
        // This makes sure that the container activity has implemented  
        // the callback interface. If not, it throws an exception  
        try {  
            mCallback = (OnHeadlineSelectedListener) activity;  
        } catch (ClassCastException e) {  
            throw new ClassCastException(activity.toString()  
                + " must implement OnHeadlineSelectedListener");  
    }  
}
```

### THIS LESSON TEACHES YOU TO

1. [Define an Interface](#)
2. [Implement the Interface](#)
3. [Deliver a Message to a Fragment](#)

### YOU SHOULD ALSO READ

- [Fragments](#)

### TRY IT OUT

[Download the sample](#)

FragmentBasics.zip

```
    }  
  
    ...  
}
```

Now the fragment can deliver messages to the activity by calling the `onArticleSelected()` method (or other methods in the interface) using the `mCallback` instance of the `OnHeadlineSelectedListener` interface.

For example, the following method in the fragment is called when the user clicks on a list item. The fragment uses the callback interface to deliver the event to the parent activity.

```
@Override  
public void onListItemClick(ListView l, View v, int position, long id) {  
    // Send the event to the host activity  
    mCallback.onArticleSelected(position);  
}
```

## Implement the Interface

In order to receive event callbacks from the fragment, the activity that hosts it must implement the interface defined in the fragment class.

For example, the following activity implements the interface from the above example.

```
public static class MainActivity extends Activity  
    implements HeadlinesFragment.OnHeadlineSelectedListener{  
  
    ...  
  
    public void onArticleSelected(int position) {  
        // The user selected the headline of an article from the HeadlinesFragmen  
        // Do something here to display that article  
    }  
}
```

## Deliver a Message to a Fragment

The host activity can deliver messages to a fragment by capturing the [Fragment](#) ([/reference/android/support/v4/app/Fragment.html](#)) instance with `findFragmentById()` ([/reference/android/support/v4/app/FragmentManager.html#findFragmentById\(int\)](#)), then directly call the fragment's public methods.

For instance, imagine that the activity shown above may contain another fragment that's used to display the item specified by the data returned in the above callback method. In this case, the activity can pass the information received in the callback method to the other fragment that will display the item:

```
public static class MainActivity extends Activity
    implements HeadlinesFragment.OnHeadlineSelectedListener{
    ...

    public void onArticleSelected(int position) {
        // The user selected the headline of an article from the HeadlinesFragment
        // Do something here to display that article

        ArticleFragment articleFrag = (ArticleFragment)
            getSupportFragmentManager().findFragmentById(R.id.article_fragment);

        if (articleFrag != null) {
            // If article frag is available, we're in two-pane layout...

            // Call a method in the ArticleFragment to update its content
            articleFrag.updateArticleView(position);
        } else {
            // Otherwise, we're in the one-pane layout and must swap frags...

            // Create fragment and give it an argument for the selected article
            ArticleFragment newFragment = new ArticleFragment();
            Bundle args = new Bundle();
            args.putInt(ArticleFragment.ARG_POSITION, position);
            newFragment.setArguments(args);

            FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();

            // Replace whatever is in the fragment_container view with this fragment,
            // and add the transaction to the back stack so the user can navigate
            transaction.replace(R.id.fragment_container, newFragment);
            transaction.addToBackStack(null);

            // Commit the transaction
            transaction.commit();
        }
    }
}
```

# Saving Data

Most Android apps need to save data, even if only to save information about the app state during [onPause\(\)](#) ([/reference/android/app/Activity.html#onPause\(\)](#)) so the user's progress is not lost. Most non-trivial apps also need to save user settings, and some apps must manage large amounts of information in files and databases. This class introduces you to the principal data storage options in Android, including:

- Saving key-value pairs of simple data types in a shared preferences file
- Saving arbitrary files in Android's file system
- Using databases managed by SQLite

## DEPENDENCIES AND PREREQUISITES

- Android 1.6 (API Level 4) or higher
- Familiarity with Map key-value collections
- Familiarity with the Java file I/O API
- Familiarity with SQL databases

## YOU SHOULD ALSO READ

- [Storage Options](#)

## Lessons

---

### [Saving Key-Value Sets](#)

Learn to use a shared preferences file for storing small amounts of information in key-value pairs.

### [Saving Files](#)

Learn to save a basic file, such as to store long sequences of data that are generally read in order.

### [Saving Data in SQL Databases](#)

Learn to use a SQLite database to read and write structured data.

# Saving Key-Value Sets

If you have a relatively small collection of key-values that you'd like to save, you should use the [SharedPreferences](#) ([/reference/android/content/SharedPreferences.html](#)) APIs. A [SharedPreferences](#) ([/reference/android/content/SharedPreferences.html](#)) object points to a file containing key-value pairs and provides simple methods to read and write them. Each [SharedPreferences](#) ([/reference/android/content/SharedPreferences.html](#)) file is managed by the framework and can be private or shared.

This class shows you how to use the [SharedPreferences](#) ([/reference/android/content/SharedPreferences.html](#)) APIs to store and retrieve simple values.

Note: The [SharedPreferences](#) ([/reference/android/content/SharedPreferences.html](#)) APIs are only for reading and writing key-value pairs and you should not confuse them with the [Preference](#) ([/reference/android/preference/Preference.html](#)) APIs, which help you build a user interface for your app settings (although they use [SharedPreferences](#) ([/reference/android/content/SharedPreferences.html](#)) as their implementation to save the app settings). For information about using the [Preference](#) ([/reference/android/preference/Preference.html](#)) APIs, see the [Settings](#) ([/guide/topics/ui/settings.html](#)) guide.

## Get a Handle to a SharedPreferences

You can create a new shared preference file or access an existing one by calling one of two methods:

- [getSharedPreferences\(\)](#) — Use this if you need multiple shared preference files identified by name, which you specify with the first parameter. You can call this from any [Context](#) in your app.
- [getPreferences\(\)](#) — Use this from an [Activity](#) if you need to use only one shared preference file for the activity. Because this retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

For example, the following code is executed inside a [Fragment](#) ([/reference/android/app/Fragment.html](#)). It accesses the shared preferences file that's identified by the resource string `R.string.preference_file_key` and opens it using the private mode so the file is accessible by only your app.

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
    getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

### THIS LESSON TEACHES YOU TO

1. [Get a Handle to a SharedPreferences](#)
2. [Write to Shared Preferences](#)
3. [Read from Shared Preferences](#)

### YOU SHOULD ALSO READ

- [Using Shared Preferences](#)

When naming your shared preference files, you should use a name that's uniquely identifiable to your app, such as "com.example.myapp.PREFERENCE\_FILE\_KEY"

Alternatively, if you need just one shared preference file for your activity, you can use the `getPreferences()` ([/reference/android/app/Activity.html#getPreferences\(int\)](/reference/android/app/Activity.html#getPreferences(int))) method:

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE)
```

Caution: If you create a shared preferences file with `MODE_WORLD_READABLE` ([/reference/android/content/Context.html#MODE\\_WORLD\\_READABLE](/reference/android/content/Context.html#MODE_WORLD_READABLE)) or `MODE_WORLD_WRITEABLE` ([/reference/android/content/Context.html#MODE\\_WORLD\\_WRITEABLE](/reference/android/content/Context.html#MODE_WORLD_WRITEABLE)), then any other apps that know the file identifier can access your data.

## Write to Shared Preferences

To write to a shared preferences file, create a `SharedPreferences.Editor` (</reference/android/content/SharedPreferences.Editor.html>) by calling `edit()` ([/reference/android/content/SharedPreferences.html#edit\(\)](/reference/android/content/SharedPreferences.html#edit())) on your `SharedPreferences` (</reference/android/content/SharedPreferences.html>).

Pass the keys and values you want to write with methods such as `putInt()` ([/reference/android/content/SharedPreferences.Editor.html#putInt\(java.lang.String, int\)](/reference/android/content/SharedPreferences.Editor.html#putInt(java.lang.String, int))) and `putString()` ([/reference/android/content/SharedPreferences.Editor.html#putString\(java.lang.String, java.lang.String\)](/reference/android/content/SharedPreferences.Editor.html#putString(java.lang.String, java.lang.String))). Then call `commit()` ([/reference/android/content/SharedPreferences.Editor.html#commit\(\)](/reference/android/content/SharedPreferences.Editor.html#commit())) to save the changes. For example:

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE)
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score), newHighScore);
editor.commit();
```

## Read from Shared Preferences

To retrieve values from a shared preferences file, call methods such as `getInt()` ([/reference/android/content/SharedPreferences.html#getInt\(java.lang.String, int\)](/reference/android/content/SharedPreferences.html#getInt(java.lang.String, int))) and `getString()` ([/reference/android/content/SharedPreferences.html#getString\(java.lang.String, java.lang.String\)](/reference/android/content/SharedPreferences.html#getString(java.lang.String, java.lang.String))), providing the key for the value you want, and optionally a default value to return if the key isn't present. For example:

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE)
long default = getResources().getInteger(R.string.saved_high_score_default);
long highScore = sharedPref.getInt(getString(R.string.saved_high_score), default)
```

# Saving Files

Android uses a file system that's similar to disk-based file systems on other platforms. This lesson describes how to work with the Android file system to read and write files with the [File \(/reference/java/io/File.html\)](#) APIs.

A [File \(/reference/java/io/File.html\)](#) object is suited to reading or writing large amounts of data in start-to-finish order without skipping around. For example, it's good for image files or anything exchanged over a network.

This lesson shows how to perform basic file-related tasks in your app. The lesson assumes that you are familiar with the basics of the Linux file system and the standard file input/output APIs in [java.io \(/reference/java/io/package-summary.html\)](#).

## THIS LESSON TEACHES YOU TO

1. [Choose Internal or External Storage](#)
2. [Obtain Permissions for External Storage](#)
3. [Save a File on Internal Storage](#)
4. [Save a File on External Storage](#)
5. [Query Free Space](#)
6. [Delete a File](#)

## YOU SHOULD ALSO READ

- [Using the Internal Storage](#)
- [Using the External Storage](#)

## Choose Internal or External Storage

All Android devices have two file storage areas: "internal" and "external" storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage). Some devices divide the permanent storage space into "internal" and "external" partitions, so even without a removable storage medium, there are always two storage spaces and the API behavior is the same whether the external storage is removable or not. The following lists summarize the facts about each storage space.

### Internal storage:

- It's always available.
- Files saved here are accessible by only your app by default.
- When the user uninstalls your app, the system removes all your app's files from internal storage.

Internal storage is best when you want to be sure that neither the user nor other apps can access your files.

### External storage:

- It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
- It's world-readable, so files saved here may be read outside of your control.
- When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from [getExternalFilesDir\(\)](#).

External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

Tip: Although apps are installed onto the internal storage by default, you can specify the [android:installLocation](#) ([/guide/topics/manifest/manifest-element.html#install](#)) attribute in your manifest so your app may be installed on external storage. Users appreciate this option when the APK size is very large and they have an external storage space that's larger than the internal storage. For more information, see [App Install Location](#) ([/guide/topics/data/install-location.html](#)).

## Obtain Permissions for External Storage

To write to the external storage, you must request the [WRITE\\_EXTERNAL\\_STORAGE](#) ([/reference/android/Manifest.permission.html#WRITE\\_EXTERNAL\\_STORAGE](#)) permission in your [manifest file](#) ([/guide/topics/manifest/manifest-intro.html](#)):

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

Caution: Currently, all apps have the ability to read the external storage without a special permission. However, this will change in a future release. If your app needs to read the external storage (but not write to it), then you will need to declare the [READ\\_EXTERNAL\\_STORAGE](#) ([/reference/android/Manifest.permission.html#READ\\_EXTERNAL\\_STORAGE](#)) permission. To ensure that your app continues to work as expected, you should declare this permission now, before the change takes effect.

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

However, if your app uses the [WRITE\\_EXTERNAL\\_STORAGE](#) ([/reference/android/Manifest.permission.html#WRITE\\_EXTERNAL\\_STORAGE](#)) permission, then it implicitly has permission to read the external storage as well.

You don't need any permissions to save files on the internal storage. Your application always has permission to read and write files in its internal storage directory.

## Save a File on Internal Storage

When saving a file to internal storage, you can acquire the appropriate directory as a [File](#) ([/reference/java/io/File.html](#)) by calling one of two methods:

[getFilesDir\(\)](#)

Returns a [File](#) representing an internal directory for your app.

[getCacheDir\(\)](#)

Returns a [File](#) representing an internal directory for your app's temporary cache files. Be sure to delete each file once it is no longer needed and implement a reasonable size limit for

the amount of memory you use at any given time, such as 1MB. If the system begins running low on storage, it may delete your cache files without warning.

To create a new file in one of these directories, you can use the [File\(\) \(/reference/java/io/File.html#File\(java.io.File, java.lang.String\)\)](#) constructor, passing the [File \(/reference/java/io/File.html\)](#) provided by one of the above methods that specifies your internal storage directory. For example:

```
File file = new File(context.getFilesDir(), filename);
```

Alternatively, you can call [openFileOutput\(\) \(/reference/android/content/Context.html#openFileOutput\(java.lang.String, int\)\)](#) to get a [FileOutputStream \(/reference/java/io/FileOutputStream.html\)](#) that writes to a file in your internal directory. For example, here's how to write some text to a file:

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
    outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
    outputStream.write(string.getBytes());
    outputStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Or, if you need to cache some files, you should instead use [createTempFile\(\) \(/reference/java/io/File.html#createTempFile\(java.lang.String, java.lang.String\)\)](#). For example, the following method extracts the file name from a [URL \(/reference/java/net/URL.html\)](#) and creates a file with that name in your app's internal cache directory:

```
public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile(fileName, null, context.getCacheDir());
    } catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

Note: Your app's internal storage directory is specified by your app's package name in a special location of the Android file system. Technically, another app can read your internal files if you set the file mode to be readable. However, the other app would also need to know your app package name and file names. Other apps cannot browse your internal directories and do not have read

or write access unless you explicitly set the files to be readable or writable. So as long as you use [MODE\\_PRIVATE](#) ([/reference/android/content/Context.html#MODE\\_PRIVATE](#)) for your files on the internal storage, they are never accessible to other apps.

## Save a File on External Storage

Because the external storage may be unavailable—such as when the user has mounted the storage to a PC or has removed the SD card that provides the external storage—you should always verify that the volume is available before accessing it. You can query the external storage state by calling [getExternalStorageState\(\)](#) ([/reference/android/os/Environment.html#getExternalStorageState\(\)](#)). If the returned state is equal to [MEDIA\\_MOUNTED](#) ([/reference/android/os/Environment.html#MEDIA\\_MOUNTED](#)), then you can read and write your files. For example, the following methods are useful to determine the storage availability:

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Although the external storage is modifiable by the user and other apps, there are two categories of files you might save here:

### Public files

Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user.

For example, photos captured by your app or other downloaded files.

### Private files

Files that rightfully belong to your app and should be deleted when the user uninstalls your app. Although these files are technically accessible by the user and other apps because they are on the external storage, they are files that realistically don't provide value to the user outside your app. When the user uninstalls your app, the system deletes all files in your app's external private directory.

For example, additional resources downloaded by your app or temporary media files.

If you want to save public files on the external storage, use the [getExternalStoragePublicDirectory\(\) \(/reference/android/os/Environment.html#getExternalStoragePublicDirectory\(java.lang.String\)\)](#) method to get a [File](#) ([/reference/java/io/File.html](#)) representing the appropriate directory on the external storage. The method takes an argument specifying the type of file you want to save so that they can be logically organized with other public files, such as [DIRECTORY\\_MUSIC \(/reference/android/os/Environment.html#DIRECTORY\\_MUSIC\)](#) or [DIRECTORY\\_PICTURES \(/reference/android/os/Environment.html#DIRECTORY\\_PICTURES\)](#). For example:

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

If you want to save files that are private to your app, you can acquire the appropriate directory by calling [getExternalFilesDir\(\) \(/reference/android/content/Context.html#getExternalFilesDir\(java.lang.String\)\)](#) and passing it a name indicating the type of directory you'd like. Each directory created this way is added to a parent directory that encapsulates all your app's external storage files, which the system deletes when the user uninstalls your app.

For example, here's a method you can use to create a directory for an individual photo album:

```
public File getAlbumStorageDir(Context context, String albumName) {
    // Get the directory for the app's private pictures directory.
    File file = new File(context.getExternalFilesDir(
        Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

If none of the pre-defined sub-directory names suit your files, you can instead call [getExternalFilesDir\(\) \(/reference/android/content/Context.html#getExternalFilesDir\(java.lang.String\)\)](#) and pass null. This returns the root directory for your app's private directory on the external storage.

Remember that [getExternalFilesDir\(\) \(/reference/android/content/Context.html#getExternalFilesDir\(java.lang.String\)\)](#) creates a directory inside a directory that is deleted when the user uninstalls your app. If the files you're saving should remain available after

the user uninstalls your app—such as when your app is a camera and the user will want to keep the photos—you should instead use `getExternalStoragePublicDirectory()` ([/reference/android/os/Environment.html#getExternalStoragePublicDirectory\(java.lang.String\)](#)).

Regardless of whether you use `getExternalStoragePublicDirectory()` ([/reference/android/os/Environment.html#getExternalStoragePublicDirectory\(java.lang.String\)](#)) for files that are shared or `getExternalFilesDir()` ([/reference/android/content/Context.html#getExternalFilesDir\(java.lang.String\)](#)) for files that are private to your app, it's important that you use directory names provided by API constants like `DIRECTORY_PICTURES` ([/reference/android/os/Environment.html#DIRECTORY\\_PICTURES](#)). These directory names ensure that the files are treated properly by the system. For instance, files saved in `DIRECTORY_RINGTONES` ([/reference/android/os/Environment.html#DIRECTORY\\_RINGTONES](#)) are categorized by the system media scanner as ringtones instead of music.

## Query Free Space

---

If you know ahead of time how much data you're saving, you can find out whether sufficient space is available without causing an `IIOException` ([/reference/java/io/IIOException.html](#)) by calling `getFreeSpace()` ([/reference/java/io/File.html#getFreeSpace\(\)](#)) or `getTotalSpace()` ([/reference/java/io/File.html#getTotalSpace\(\)](#)). These methods provide the current available space and the total space in the storage volume, respectively. This information is also useful to avoid filling the storage volume above a certain threshold.

However, the system does not guarantee that you can write as many bytes as are indicated by `getFreeSpace()` ([/reference/java/io/File.html#getFreeSpace\(\)](#)). If the number returned is a few MB more than the size of the data you want to save, or if the file system is less than 90% full, then it's probably safe to proceed. Otherwise, you probably shouldn't write to storage.

Note: You aren't required to check the amount of available space before you save your file. You can instead try writing the file right away, then catch an `IIOException` ([/reference/java/io/IIOException.html](#)) if one occurs. You may need to do this if you don't know exactly how much space you need. For example, if you change the file's encoding before you save it by converting a PNG image to JPEG, you won't know the file's size beforehand.

## Delete a File

---

You should always delete files that you no longer need. The most straightforward way to delete a file is to have the opened file reference call `delete()` ([/reference/java/io/File.html#delete\(\)](#)) on itself.

```
myFile.delete();
```

If the file is saved on internal storage, you can also ask the `Context` ([/reference/android/content/Context.html](#)) to locate and delete a file by calling `deleteFile()` ([/reference/android/content/Context.html#deleteFile\(java.lang.String\)](#)):

```
myContext.deleteFile(fileName);
```

Note: When the user uninstalls your app, the Android system deletes the following:

- All files you saved on internal storage
- All files you saved on external storage using `getExternalFilesDir()`.  
However, you should manually delete all cached files created with `getCacheDir()` ([/reference /android/content/Context.html#getCacheDir\(\)](#)) on a regular basis and also regularly delete other files you no longer need.

# Saving Data in SQL Databases

Saving data to a database is ideal for repeating or structured data, such as contact information. This class assumes that you are familiar with SQL databases in general and helps you get started with SQLite databases on Android. The APIs you'll need to use a database on Android are available in the [android.database.sqlite](#) ([/reference/android/database/sqlite/package-summary.html](#)) package.

## Define a Schema and Contract

One of the main principles of SQL databases is the schema: a formal declaration of how the database is organized. The schema is reflected in the SQL statements that you use to create your database. You may find it helpful to create a companion class, known as a *contract* class, which explicitly specifies the layout of your schema in a systematic and self-documenting way.

A contract class is a container for constants that define names for URIs, tables, and columns. The contract class allows you to use the same constants across all the other classes in the same package. This lets you change a column name in one place and have it propagate throughout your code.

A good way to organize a contract class is to put definitions that are global to your whole database in the root level of the class. Then create an inner class for each table that enumerates its columns.

Note: By implementing the [BaseColumns](#) ([/reference/android/provider/BaseColumns.html](#)) interface, your inner class can inherit a primary key field called `_ID` that some Android classes such as cursor adaptors will expect it to have. It's not required, but this can help your database work harmoniously with the Android framework.

For example, this snippet defines the table name and column names for a single table:

```
public static abstract class FeedEntry implements BaseColumns {
    public static final String TABLE_NAME = "entry";
    public static final String COLUMN_NAME_ENTRY_ID = "entryid";
    public static final String COLUMN_NAME_TITLE = "title";
    public static final String COLUMN_NAME_SUBTITLE = "subtitle";
    ...
}
```

### THIS LESSON TEACHES YOU TO

1. [Define a Schema and Contract](#)
2. [Create a Database Using a SQL Helper](#)
3. [Put Information into a Database](#)
4. [Read Information from a Database](#)
5. [Delete Information from a Database](#)
6. [Update a Database](#)

### YOU SHOULD ALSO READ

- [Using Databases](#)

To prevent someone from accidentally instantiating the contract class, give it an empty constructor.

```
// Prevents the FeedReaderContract class from being instantiated.  
private FeedReaderContract() {}
```

## Create a Database Using a SQL Helper

Once you have defined how your database looks, you should implement methods that create and maintain the database and tables. Here are some typical statements that create and delete a table:

```
private static final String TEXT_TYPE = " TEXT";  
private static final String COMMA_SEP = ",";  
private static final String SQL_CREATE_ENTRIES =  
    "CREATE TABLE " + FeedReaderContract.FeedEntry.TABLE_NAME + " (" +  
    FeedReaderContract.FeedEntry._ID + " INTEGER PRIMARY KEY," +  
    FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + TEXT_TYPE + COMMA_SEP +  
    FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE + TEXT_TYPE + COMMA_SEP +  
    ... // Any other options for the CREATE command  
    " )";  
  
private static final String SQL_DELETE_ENTRIES =  
    "DROP TABLE IF EXISTS " + TABLE_NAME_ENTRIES;
```

Just like files that you save on the device's [internal storage](#) ([/guide/topics/data/data-storage.html#filesInternal](#)), Android stores your database in private disk space that's associated with your application. Your data is secure, because by default this area is not accessible to other applications.

A useful set of APIs is available in the [SQLiteOpenHelper](#) ([/reference/android/database/sqlite/SQLiteOpenHelper.html](#)) class. When you use this class to obtain references to your database, the system performs the potentially long-running operations of creating and updating the database only when needed and *not during app startup*. All you need to do is call [getWritableDatabase\(\)](#) ([/reference/android/database/sqlite/SQLiteOpenHelper.html#getWritableDatabase\(\)](#)) or [getReadableDatabase\(\)](#) ([/reference/android/database/sqlite/SQLiteOpenHelper.html#getReadableDatabase\(\)](#)).

Note: Because they can be long-running, be sure that you call [getWritableDatabase\(\)](#) ([/reference/android/database/sqlite/SQLiteOpenHelper.html#getWritableDatabase\(\)](#)) or [getReadableDatabase\(\)](#) ([/reference/android/database/sqlite/SQLiteOpenHelper.html#getReadableDatabase\(\)](#)) in a background thread, such as with [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) or [IntentService](#) ([/reference/android/app/IntentService.html](#)).

To use [SQLiteOpenHelper](#) ([/reference/android/database/sqlite/SQLiteOpenHelper.html](#)), create a

subclass that overrides the [onCreate\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#onCreate\(android.database.sqlite.SQLiteDatabase\)](#)), [onUpgrade\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#onUpgrade\(android.database.sqlite.SQLiteDatabase, int, int\)](#)) and [onOpen\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#onOpen\(android.database.sqlite.SQLiteDatabase\)](#)) callback methods. You may also want to implement [onDowngrade\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#onDowngrade\(android.database.sqlite.SQLiteDatabase, int, int\)](#)), but it's not required.

For example, here's an implementation of [SQLiteDatabaseOpenHelper](#) ([/reference/android/database/sqlite/SQLiteDatabaseOpenHelper.html](#)) that uses some of the commands shown above:

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database version
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // This database is only a cache for online data, so its upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

To access your database, instantiate your subclass of [SQLiteDatabaseOpenHelper](#) ([/reference/android/database/sqlite/SQLiteDatabaseOpenHelper.html](#)):

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

## Put Information into a Database

Insert data into the database by passing a [ContentValues](#) ([/reference/android/content/ContentValues.html](#)) object to the [insert\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#insert\(java.lang.String, java.lang.String, android.content.ContentValues\)](#)) method:

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new row
long newRowId;
newRowId = db.insert(
        FeedReaderContract.FeedEntry.TABLE_NAME,
        FeedReaderContract.FeedEntry.COLUMN_NAME_NULLABLE,
        values);
```

The first argument for [insert\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#insert\(java.lang.String, java.lang.String, android.content.ContentValues\)](#)) is simply the table name. The second argument provides the name of a column in which the framework can insert NULL in the event that the [ContentValues](#) ([/reference/android/content/ContentValues.html](#)) is empty (if you instead set this to "null", then the framework will not insert a row when there are no values).

## Read Information from a Database

To read from a database, use the [query\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#query\(boolean, java.lang.String, java.lang.String\[\], java.lang.String, java.lang.String\[\], java.lang.String, java.lang.String, java.lang.String, java.lang.String\)](#) method, passing it your selection criteria and desired columns. The method combines elements of [insert\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#insert\(java.lang.String, java.lang.String, android.content.ContentValues\)](#)) and [update\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#update\(java.lang.String, android.content.ContentValues, java.lang.String, java.lang.String\[\]\)](#)), except the column list defines the data you want to fetch, rather than the data to insert. The results of the query are returned to you in a [Cursor](#) ([/reference/android/database/Cursor.html](#)) object.

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
        FeedReaderContract.FeedEntry._ID,
        FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE,
        FeedReaderContract.FeedEntry.COLUMN_NAME_UPDATED,
        ...
}
```

```
};

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedReaderContract.FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedReaderContract.FeedEntry.TABLE_NAME, // The table to query
    projection, // The columns to return
    selection, // The columns for the WHERE clause
    selectionArgs, // The values for the WHERE clause
    null, // don't group the rows
    null, // don't filter by row groups
    sortOrder // The sort order
);
```

To look at a row in the cursor, use one of the [Cursor](#) ([/reference/android/database/Cursor.html](#)) move methods, which you must always call before you begin reading values. Generally, you should start by calling [moveToFirst\(\)](#) ([/reference/android/database/Cursor.html#moveToFirst\(\)](#)), which places the "read position" on the first entry in the results. For each row, you can read a column's value by calling one of the [Cursor](#) ([/reference/android/database/Cursor.html](#)) get methods, such as [getString\(\)](#) ([/reference/android/database/Cursor.html#getString\(int\)](#)) or [getLong\(\)](#) ([/reference/android/database/Cursor.html#getLong\(int\)](#)). For each of the get methods, you must pass the index position of the column you desire, which you can get by calling [getColumnIndex\(\)](#) ([/reference/android/database/Cursor.html#getColumnIndex\(java.lang.String\)](#)) or [getColumnIndexOrThrow\(\)](#) ([/reference/android/database/Cursor.html#getColumnIndexOrThrow\(java.lang.String\)](#)). For example:

```
cursor.moveToFirst();
long itemId = cursor.getLong(
    cursor.getColumnIndexOrThrow(FeedReaderContract.FeedEntry._ID)
);
```

## Delete Information from a Database

To delete rows from a table, you need to provide selection criteria that identify the rows. The database API provides a mechanism for creating selection criteria that protects against SQL injection. The mechanism divides the selection specification into a selection clause and selection arguments. The clause defines the columns to look at, and also allows you to combine column tests. The arguments are values to test against that are bound into the clause. Because the result isn't handled the same as a regular SQL statement, it is immune to SQL injection.

```
// Define 'where' part of query.
String selection = FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { String.valueOf(rowId) };
```

```
// Issue SQL statement.  
db.delete(table_name, mySelection, selectionArgs);
```

## Update a Database

When you need to modify a subset of your database values, use the [update\(\) \(/reference/android/database/sqlite/SQLiteDatabase.html#update\(java.lang.String, android.content.ContentValues, java.lang.String, java.lang.String\[\]\)\)](#) method.

Updating the table combines the content values syntax of [insert\(\) \(/reference/android/database/sqlite/SQLiteDatabase.html#insert\(java.lang.String, java.lang.String, android.content.ContentValues\)\)](#) with the where syntax of [delete\(\) \(/reference/android/database/sqlite/SQLiteDatabase.html#delete\(java.lang.String, java.lang.String, java.lang.String\[\]\)\)](#).

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();  
  
// New value for one column  
ContentValues values = new ContentValues();  
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE, title);  
  
// Which row to update, based on the ID  
String selection = FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";  
String[] selectionArgs = { String.valueOf(rowId) };  
  
int count = db.update(  
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,  
    values,  
    selection,  
    selectionArgs);
```

# Interacting with Other Apps

An Android app typically has several [activities \(/guide/components/activities.html\)](#). Each activity displays a user interface that allows the user to perform a specific task (such as view a map or take a photo). To take the user from one activity to another, your app must use an [Intent \(/reference/android/content/Intent.html\)](#) to define your app's "intent" to do something. When you pass an [Intent \(/reference/android/content/Intent.html\)](#) to the system with a method such as [startActivity\(\) \(/reference/android/app/Activity.html#startActivity\(android.content.Intent\)\)](#), the system uses the [Intent \(/reference/android/content/Intent.html\)](#) to identify and start the appropriate app component. Using intents even allows your app to start an activity that is contained in a separate app.

An [Intent \(/reference/android/content/Intent.html\)](#) can be *explicit* in order to start a specific component (a specific [Activity \(/reference/android/app/Activity.html\)](#) instance) or *implicit* in order to start any component that can handle the intended action (such as "capture a photo").

This class shows you how to use an [Intent \(/reference/android/content/Intent.html\)](#) to perform some basic interactions with other apps, such as start another app, receive a result from that app, and make your app able to respond to intents from other apps.

## Lessons

---

### [\*\*Sending the User to Another App\*\*](#)

Shows how you can create implicit intents to launch other apps that can perform an action.

### [\*\*Getting a Result from an Activity\*\*](#)

Shows how to start another activity and receive a result from the activity.

### [\*\*Allowing Other Apps to Start Your Activity\*\*](#)

Shows how to make activities in your app open for use by other apps by defining intent filters that declare the implicit intents your app accepts.

## DEPENDENCIES AND PREREQUISITES

---

- Basic understanding of the Activity lifecycle (see [Managing the Activity Lifecycle](#))

## YOU SHOULD ALSO READ

---

- [Sharing Content](#)
- [Integrating Application with Intents \(blog post\)](#)
- [Intents and Intent Filters](#)

# Sending the User to Another App

One of Android's most important features is an app's ability to send the user to another app based on an "action" it would like to perform. For example, if your app has the address of a business that you'd like to show on a map, you don't have to build an activity in your app that shows a map. Instead, you can create a request to view the address using an [Intent \(/reference/android/content/Intent.html\)](#). The Android system then starts an app that's able to show the address on a map.

As explained in the first class, [Building Your First App \(/training/basics/firstapp/index.html\)](#), you must use intents to navigate between activities in your own app. You generally do so with an *explicit intent*, which defines the exact class name of the component you want to start. However, when you want to have a separate app perform an action, such as "view a map," you must use an *implicit intent*.

This lesson shows you how to create an implicit intent for a particular action, and how to use it to start an activity that performs the action in another app.

## Build an Implicit Intent

Implicit intents do not declare the class name of the component to start, but instead declare an action to perform. The action specifies the thing you want to do, such as *view*, *edit*, *send*, or *get* something. Intents often also include data associated with the action, such as the address you want to view, or the email message you want to send. Depending on the intent you want to create, the data might be a [Uri \(/reference/android/net/Uri.html\)](#), one of several other data types, or the intent might not need data at all.

If your data is a [Uri \(/reference/android/net/Uri.html\)](#), there's a simple [Intent\(\) \(/reference/android/content/Intent.html#Intent\(java.lang.String, android.net.Uri\)\)](#) constructor you can use define the action and data.

For example, here's how to create an intent to initiate a phone call using the [Uri \(/reference/android/net/Uri.html\)](#) data to specify the telephone number:

```
Uri number = Uri.parse("tel:5551234");
Intent callIntent = new Intent(Intent.ACTION_DIAL, number);
```

When your app invokes this intent by calling [startActivity\(\) \(/reference/android/app/Activity.html#startActivity\(android.content.Intent\)\)](#), the Phone app initiates a call to the given phone number.

### THIS LESSON TEACHES YOU TO

1. [Build an Implicit Intent](#)
2. [Verify There is an App to Receive the Intent](#)
3. [Start an Activity with the Intent](#)
4. [Show an App Chooser](#)

### YOU SHOULD ALSO READ

- [Sharing Content](#)

Here are a couple other intents and their action and [Uri \(/reference/android/net/Uri.html\)](#) data pairs:

- View a map:

```
// Map point based on address  
Uri location = Uri.parse("geo:0,0?q=1600+Amphitheatre+Parkway,+Mountain+View,+California");  
// Or map point based on latitude/longitude  
// Uri location = Uri.parse("geo:37.422219,-122.08364?z=14"); // z param is zoom  
Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);
```

- View a web page:

```
Uri webpage = Uri.parse("http://www.android.com");  
Intent webIntent = new Intent(Intent.ACTION_VIEW, webpage);
```

Other kinds of implicit intents require "extra" data that provide different data types, such as a string. You can add one or more pieces of extra data using the various [putExtra\(\) \(/reference/android/content/Intent.html#putExtra\(java.lang.String, java.lang.String\)\)](#) methods.

By default, the system determines the appropriate MIME type required by an intent based on the [Uri \(/reference/android/net/Uri.html\)](#) data that's included. If you don't include a [Uri \(/reference/android/net/Uri.html\)](#) in the intent, you should usually use [setType\(\) \(/reference/android/content/Intent.html#setType\(java.lang.String\)\)](#) to specify the type of data associated with the intent. Setting the MIME type further specifies which kinds of activities should receive the intent.

Here are some more intents that add extra data to specify the desired action:

- Send an email with an attachment:

```
Intent emailIntent = new Intent(Intent.ACTION_SEND);  
// The intent does not have a URI, so declare the "text/plain" MIME type  
emailIntent.setType(HTTP.PLAIN_TEXT_TYPE);  
emailIntent.putExtra(Intent.EXTRA_EMAIL, new String[] {"jon@example.com"}); // recipient  
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Email subject");  
emailIntent.putExtra(Intent.EXTRA_TEXT, "Email message text");  
emailIntent.putExtra(Intent.EXTRA_STREAM, Uri.parse("content://path/to/email/attachment"));  
// You can also attach multiple items by passing an ArrayList of Uris
```

- Create a calendar event:

```
Intent calendarIntent = new Intent(Intent.ACTION_INSERT, Events.CONTENT_URI);  
Calendar beginTime = Calendar.getInstance().set(2012, 0, 19, 7, 30);  
Calendar endTime = Calendar.getInstance().set(2012, 0, 19, 10, 30);  
calendarIntent.putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME, beginTime.getTimeInMillis());  
calendarIntent.putExtra(CalendarContract.EXTRA_EVENT_END_TIME, endTime.getTimeInMillis());  
calendarIntent.putExtra(Events.TITLE, "Ninja class");  
calendarIntent.putExtra(Events.EVENT_LOCATION, "Secret dojo");
```

Note: This intent for a calendar event is supported only with API level 14 and higher.

Note: It's important that you define your [Intent](#) ([/reference/android/content/Intent.html](#)) to be as specific as possible. For example, if you want to display an image using the [ACTION\\_VIEW](#) ([/reference/android/content/Intent.html#ACTION\\_VIEW](#)) intent, you should specify a MIME type of `image/*`. This prevents apps that can "view" other types of data (like a map app) from being triggered by the intent.

## Verify There is an App to Receive the Intent

Although the Android platform guarantees that certain intents will resolve to one of the built-in apps (such as the Phone, Email, or Calendar app), you should always include a verification step before invoking an intent.

Caution: If you invoke an intent and there is no app available on the device that can handle the intent, your app will crash.

To verify there is an activity available that can respond to the intent, call `queryIntentActivities()` ([/reference/android/content/pm/PackageManager.html#queryIntentActivities\(android.content.Intent, int\)](#)) to get a list of activities capable of handling your [Intent](#) ([/reference/android/content/Intent.html](#)). If the returned [List](#) ([/reference/java/util/List.html](#)) is not empty, you can safely use the intent. For example:

```
PackageManager packageManager = getPackageManager();
List<ResolveInfo> activities = packageManager.queryIntentActivities(intent, 0);
boolean isIntentSafe = activities.size() > 0;
```

If `isIntentSafe` is true, then at least one app will respond to the intent. If it is false, then there aren't any apps to handle the intent.

Note: You should perform this check when your activity first starts in case you need to disable the feature that uses the intent before the user attempts to use it. If you know of a specific app that can handle the intent, you can also provide a link for the user to download the app (see how to [link to your product on Google Play](#) ([/distribute/googleplay/promote/linking.html](#))).

## Start an Activity with the Intent

Once you have created your [Intent](#) ([/reference/android/content/Intent.html](#)) and set the extra info, call `startActivity()` ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#)) to send it to the system. If the system identifies more than one activity that can handle the intent, it displays a dialog for the user to select which app to use, as shown in figure 1. If there is only one activity that handles the intent, the system immediately starts it.

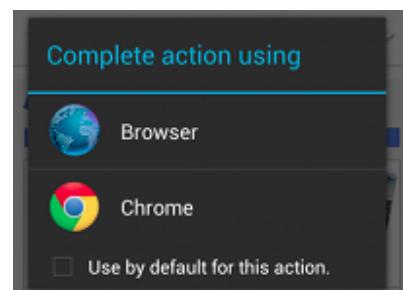


Figure 1. Example of the selection dialog that appears

```
startActivity(intent);
```

when more than one app can handle an intent.

Here's a complete example that shows how to create an intent to view a map, verify that an app exists to handle the intent, then start it:

```
// Build the intent
Uri location = Uri.parse("geo:0,0?q=1600+Amphitheatre+Parkway,+Mountain+View,+California");
Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);

// Verify it resolves
PackageManager packageManager = getPackageManager();
List<ResolveInfo> activities = packageManager.queryIntentActivities(mapIntent, 0);
boolean isIntentSafe = activities.size() > 0;

// Start an activity if it's safe
if (isIntentSafe) {
    startActivity(mapIntent);
}
```

## Show an App Chooser

Notice that when you start an activity by passing your [Intent](#) ([/reference/android/content/Intent.html](#)) to [startActivity\(\)](#) ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#)) and there is more than one app that responds to the intent, the user can select which app to use by default (by selecting a checkbox at the bottom of the dialog; see figure 1). This is nice when performing an action for which the user generally wants to use the same app every time, such as when opening a web page (users likely use just one web browser) or taking a photo (users likely prefer one camera). However, if the action to be performed could be handled by multiple apps and the user might prefer a different app each time—such as a "share" action, for which users might have several apps through which they might share an item—you should explicitly show a chooser dialog, which forces the user to select which app to use for the action every time (the user cannot select a default app for the action).

To show the chooser, create an [Intent](#) ([/reference/android/content/Intent.html](#)) using [createChooser\(\)](#) ([/reference/android/content/Intent.html#createChooser\(android.content.Intent, java.lang.CharSequence\)](#)) and pass it to [startActivity\(\)](#) ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#)). For example:

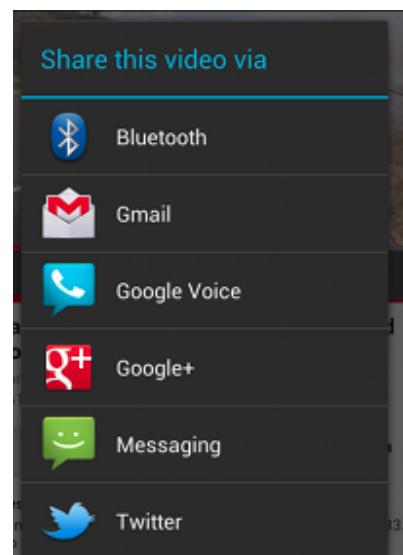


Figure 2. Example of the chooser dialog that appears when you use [createChooser\(\)](#) ([/reference/android/content/Intent.html#createChooser\(android.content.Intent, java.lang.CharSequence\)](#)) to ensure that the user is always

```
Intent intent = new Intent(Intent.ACTION_SEND);  
...  
// Always use string resources for UI text. This says s  
String title = getResources().getText(R.string.chooser_  
// Create and start the chooser  
Intent chooser = Intent.createChooser(intent, title);  
startActivity(chooser);
```

shown a list of apps that respond to your intent.

This displays a dialog with a list of apps that respond to the intent passed to the [createChooser\(\)](#) ([/reference/android/content/Intent.html#createChooser\(android.content.Intent, java.lang.CharSequence\)](#)) method and uses the supplied text as the dialog title.

# Getting a Result from an Activity

Starting another activity doesn't have to be one-way. You can also start another activity and receive a result back. To receive a result, call [startActivityForResult\(\)](#) ([/reference/android/app/Activity.html#startActivityForResult\(android.content.Intent, int\)](#)) (instead of [startActivity\(\)](#) ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#))).

## THIS LESSON TEACHES YOU TO

1. [Start the Activity](#)
2. [Receive the Result](#)

## YOU SHOULD ALSO READ

- [Sharing Content](#)

[/app/Activity.html#startActivityForResult\(android.content.Intent, int\)](#) (instead of [startActivity\(\)](#) ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#))).

For example, your app can start a camera app and receive the captured photo as a result. Or, you might start the People app in order for the user to select a contact and you'll receive the contact details as a result.

Of course, the activity that responds must be designed to return a result. When it does, it sends the result as another [Intent](#) ([/reference/android/content/Intent.html](#)) object. Your activity receives it in the [onActivityResult\(\)](#) ([/reference/android/app/Activity.html#onActivityResult\(int, int, android.content.Intent\)](#)) callback.

Note: You can use explicit or implicit intents when you call [startActivityForResult\(\)](#) ([/reference/android/app/Activity.html#startActivityForResult\(android.content.Intent, int\)](#)). When starting one of your own activities to receive a result, you should use an explicit intent to ensure that you receive the expected result.

## Start the Activity

There's nothing special about the [Intent](#) ([/reference/android/content/Intent.html](#)) object you use when starting an activity for a result, but you do need to pass an additional integer argument to the [startActivityForResult\(\)](#) ([/reference/android/app/Activity.html#startActivityForResult\(android.content.Intent, int\)](#)) method.

The integer argument is a "request code" that identifies your request. When you receive the result [Intent](#) ([/reference/android/content/Intent.html](#)), the callback provides the same request code so that your app can properly identify the result and determine how to handle it.

For example, here's how to start an activity that allows the user to pick a contact:

```
static final int PICK_CONTACT_REQUEST = 1; // The request code
...
private void pickContact() {
    Intent pickContactIntent = new Intent(Intent.ACTION_PICK, new Uri("content://"))
```

```
pickContactIntent.setType(Phone.CONTENT_TYPE); // Show user only contacts w/  
startActivityForResult(pickContactIntent, PICK_CONTACT_REQUEST);  
}
```

## Receive the Result

When the user is done with the subsequent activity and returns, the system calls your activity's `onActivityResult()` ([/reference/android/app/Activity.html#onActivityResult\(int, int, android.content.Intent\)](/reference/android/app/Activity.html#onActivityResult(int, int, android.content.Intent))) method. This method includes three arguments:

- The request code you passed to `startActivityForResult()`.
- A result code specified by the second activity. This is either `RESULT_OK` if the operation was successful or `RESULT_CANCELED` if the user backed out or the operation failed for some reason.
- An `Intent` that carries the result data.

For example, here's how you can handle the result for the "pick a contact" intent:

```
@Override  
protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
    // Check which request we're responding to  
    if (requestCode == PICK_CONTACT_REQUEST) {  
        // Make sure the request was successful  
        if (resultCode == RESULT_OK) {  
            // The user picked a contact.  
            // The Intent's data Uri identifies which contact was selected.  
  
            // Do something with the contact here (bigger example below)  
        }  
    }  
}
```

In this example, the result `Intent` (</reference/android/content/Intent.html>) returned by Android's Contacts or People app provides a content `Uri` (</reference/android/net/Uri.html>) that identifies the contact the user selected.

In order to successfully handle the result, you must understand what the format of the result `Intent` (</reference/android/content/Intent.html>) will be. Doing so is easy when the activity returning a result is one of your own activities. Apps included with the Android platform offer their own APIs that you can count on for specific result data. For instance, the People app (Contacts app on some older versions) always returns a result with the content URI that identifies the selected contact, and the Camera app returns a `Bitmap` (</reference/android/graphics/Bitmap.html>) in the "data" extra (see the class about [Capturing Photos](#) (</training/camera/index.html>)).

### Bonus: Read the contact data

The code above showing how to get a result from the People app doesn't go into details about how to actually read the data from the result, because it requires more advanced discussion about

[content providers](#) ([/guide/topics/providers/content-providers.html](#)). However, if you're curious, here's some more code that shows how to query the result data to get the phone number from the selected contact:

```
@Override  
protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
    // Check which request it is that we're responding to  
    if (requestCode == PICK_CONTACT_REQUEST) {  
        // Make sure the request was successful  
        if (resultCode == RESULT_OK) {  
            // Get the URI that points to the selected contact  
            Uri contactUri = data.getData();  
            // We only need the NUMBER column, because there will be only one row  
            String[] projection = {Phone.NUMBER};  
  
            // Perform the query on the contact to get the NUMBER column  
            // We don't need a selection or sort order (there's only one result if  
            // CAUTION: The query() method should be called from a separate thread  
            // your app's UI thread. (For simplicity of the sample, this code does not  
            // Consider using CursorLoader to perform the query.  
            Cursor cursor = getContentResolver()  
                .query(contactUri, projection, null, null, null);  
            cursor.moveToFirst();  
  
            // Retrieve the phone number from the NUMBER column  
            int column = cursor.getColumnIndex(Phone.NUMBER);  
            String number = cursor.getString(column);  
  
            // Do something with the phone number...  
        }  
    }  
}
```

Note: Before Android 2.3 (API level 9), performing a query on the [Contacts Provider](#) ([/reference/android/provider/ContactsContract.Contacts.html](#)) (like the one shown above) requires that your app declare the [READ\\_CONTACTS](#) ([/reference/android/Manifest.permission.html#READ\\_CONTACTS](#)) permission (see [Security and Permissions](#) ([/guide/topics/security/security.html](#))). However, beginning with Android 2.3, the Contacts/People app grants your app a temporary permission to read from the Contacts Provider when it returns you a result. The temporary permission applies only to the specific contact requested, so you cannot query a contact other than the one specified by the intent's [Uri](#) ([/reference/android/net/Uri.html](#)), unless you do declare the [READ\\_CONTACTS](#) ([/reference/android/Manifest.permission.html#READ\\_CONTACTS](#)) permission.

# Allowing Other Apps to Start Your Activity

The previous two lessons focused on one side of the story: starting another app's activity from your app. But if your app can perform an action that might be useful to another app, your app should be prepared to respond to action requests from other apps. For instance, if you build a social app that can share messages or photos with the user's friends, it's in your best interest to support the [ACTION\\_SEND](#) ([/reference/android/content/Intent.html#ACTION\\_SEND](#)) intent so users can initiate a "share" action from another app and launch your app to perform the action.

To allow other apps to start your activity, you need to add an [`<intent-filter>`](#) ([/guide/topics/manifest/intent-filter-element.html](#)) element in your manifest file for the corresponding [`<activity>`](#) ([/guide/topics/manifest/activity-element.html](#)) element.

When your app is installed on a device, the system identifies your intent filters and adds the information to an internal catalog of intents supported by all installed apps. When an app calls [`startActivity\(\)`](#) ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#)) or [`startActivityForResult\(\)`](#) ([/reference/android/app/Activity.html#startActivityForResult\(android.content.Intent, int\)](#)), with an implicit intent, the system finds which activity (or activities) can respond to the intent.

## Add an Intent Filter

In order to properly define which intents your activity can handle, each intent filter you add should be as specific as possible in terms of the type of action and data the activity accepts.

The system may send a given [`Intent`](#) ([/reference/android/content/Intent.html](#)) to an activity if that activity has an intent filter fulfills the following criteria of the [`Intent`](#) ([/reference/android/content/Intent.html](#)) object:

### Action

A string naming the action to perform. Usually one of the platform-defined values such as [ACTION\\_SEND](#) or [ACTION\\_VIEW](#).

Specify this in your intent filter with the [`<action>`](#) ([/guide/topics/manifest/action-element.html](#)) element. The value you specify in this element must be the full string name for the action, instead of the API constant (see the examples below).

### Data

A description of the data associated with the intent.

Specify this in your intent filter with the [`<data>`](#) ([/guide/topics/manifest/data-element.html](#)) element. Using one or more attributes in this element, you can specify just the MIME type, just a URI prefix, just a URI scheme, or a combination of these and others that indicate the data type

### THIS LESSON TEACHES YOU TO

1. [Add an Intent Filter](#)
2. [Handle the Intent in Your Activity](#)
3. [Return a Result](#)

### YOU SHOULD ALSO READ

- [Sharing Content](#)

accepted.

Note: If you don't need to declare specifics about the data [Uri \(/reference/android/net/Uri.html\)](#) (such as when your activity handles to other kind of "extra" data, instead of a URI), you should specify only the android:mimeType attribute to declare the type of data your activity handles, such as text/plain or image/jpeg.

## Category

Provides an additional way to characterize the activity handling the intent, usually related to the user gesture or location from which it's started. There are several different categories supported by the system, but most are rarely used. However, all implicit intents are defined with [CATEGORY\\_DEFAULT](#) by default.

Specify this in your intent filter with the [<category> \(/guide/topics/manifest/category-element.html\)](#) element.

In your intent filter, you can declare which criteria your activity accepts by declaring each of them with corresponding XML elements nested in the [<intent-filter> \(/guide/topics/manifest/intent-filter-element.html\)](#) element.

For example, here's an activity with an intent filter that handles the [ACTION\\_SEND \(/reference/android/content/Intent.html#ACTION\\_SEND\)](#) intent when the data type is either text or an image:

```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
        <data android:mimeType="image/*"/>
    </intent-filter>
</activity>
```

Each incoming intent specifies only one action and one data type, but it's OK to declare multiple instances of the [<action> \(/guide/topics/manifest/action-element.html\)](#), [<category> \(/guide/topics/manifest/category-element.html\)](#), and [<data> \(/guide/topics/manifest/data-element.html\)](#) elements in each [<intent-filter> \(/guide/topics/manifest/intent-filter-element.html\)](#).

If any two pairs of action and data are mutually exclusive in their behaviors, you should create separate intent filters to specify which actions are acceptable when paired with which data types.

For example, suppose your activity handles both text and images for both the [ACTION\\_SEND \(/reference/android/content/Intent.html#ACTION\\_SEND\)](#) and [ACTION\\_SENDTO \(/reference/android/content/Intent.html#ACTION\\_SENDTO\)](#) intents. In this case, you must define two separate intent filters for the two actions because a [ACTION\\_SENDTO \(/reference/android/content/Intent.html#ACTION\\_SENDTO\)](#) intent must use the data [Uri \(/reference/android/net/Uri.html\)](#) to specify the recipient's address using the send or sendto URI scheme. For example:

```
<activity android:name="ShareActivity">
    <!-- filter for sending text; accepts SENDTO action with sms URI schemes -->
```

```

<intent-filter>
    <action android:name="android.intent.action.SENDTO"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:scheme="sms" />
    <data android:scheme="smsto" />
</intent-filter>
<!-- filter for sending text or images; accepts SEND action and text or image -->
<intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="image/*"/>
    <data android:mimeType="text/plain"/>
</intent-filter>
</activity>

```

Note: In order to receive implicit intents, you must include the [CATEGORY\\_DEFAULT](#) ([/reference/android/content/Intent.html#CATEGORY\\_DEFAULT](#)) category in the intent filter. The methods [startActivity\(\)](#) ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#)) and [startActivityForResult\(\)](#) ([/reference/android/app/Activity.html#startActivityForResult\(android.content.Intent, int\)](#)) treat all intents as if they contained the [CATEGORY\\_DEFAULT](#) ([/reference/android/content/Intent.html#CATEGORY\\_DEFAULT](#)) category. If you do not declare it, no implicit intents will resolve to your activity.

For more information about sending and receiving [ACTION\\_SEND](#) ([/reference/android/content/Intent.html#ACTION\\_SEND](#)) intents that perform social sharing behaviors, see the lesson about [Receiving Content from Other Apps](#) ([/training/sharing/receive.html](#)).

## Handle the Intent in Your Activity

In order to decide what action to take in your activity, you can read the [Intent](#) ([/reference/android/content/Intent.html](#)) that was used to start it.

As your activity starts, call [getIntent\(\)](#) ([/reference/android/app/Activity.html#getIntent\(\)](#)) to retrieve the [Intent](#) ([/reference/android/content/Intent.html](#)) that started the activity. You can do so at any time during the lifecycle of the activity, but you should generally do so during early callbacks such as [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) or [onStart\(\)](#) ([/reference/android/app/Activity.html#onStart\(\)](#)).

For example:

```

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

    setContentView(R.layout.main);

    // Get the intent that started this activity

```

```

Intent intent = getIntent();
Uri data = intent.getData();

// Figure out what to do based on the intent type
if (intent.getType().indexOf("image/") != -1) {
    // Handle intents with image data ...
} else if (intent.getType().equals("text/plain")) {
    // Handle intents with text ...
}
}

```

## Return a Result

If you want to return a result to the activity that invoked yours, simply call [setResult\(\)](#) ([/reference/android/app/Activity.html#setResult\(int, android.content.Intent\)](#)) to specify the result code and result Intent ([/reference/android/content/Intent.html](#)). When your operation is done and the user should return to the original activity, call [finish\(\)](#) ([/reference/android/app/Activity.html#finish\(\)](#)) to close (and destroy) your activity. For example:

```

// Create intent to deliver some kind of result data
Intent result = new Intent("com.example.RESULT_ACTION", Uri.parse("content://resu
 setResult(Activity.RESULT_OK, result);
finish();

```

You must always specify a result code with the result. Generally, it's either [RESULT\\_OK](#) ([/reference/android/app/Activity.html#RESULT\\_OK](#)) or [RESULT\\_CANCELED](#) ([/reference/android/app/Activity.html#RESULT\\_CANCELED](#)). You can then provide additional data with an Intent ([/reference/android/content/Intent.html](#)), as necessary.

Note: The result is set to [RESULT\\_CANCELED](#) ([/reference/android/app/Activity.html#RESULT\\_CANCELED](#)) by default. So, if the user presses the *Back* button before completing the action and before you set the result, the original activity receives the "canceled" result.

If you simply need to return an integer that indicates one of several result options, you can set the result code to any value higher than 0. If you use the result code to deliver an integer and you have no need to include the Intent ([/reference/android/content/Intent.html](#)), you can call [setResult\(\)](#) ([/reference/android/app/Activity.html#setResult\(int\)](#)) and pass only a result code. For example:

```

setResult(RESULT_COLOR_RED);
finish();

```

In this case, there might be only a handful of possible results, so the result code is a locally defined integer (greater than 0). This works well when you're returning a result to an activity in your own app, because the activity that receives the result can reference the public constant to determine the value of the result code.

Note: There's no need to check whether your activity was started with `startActivity()` ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#)) or `startActivityForResult()` ([/reference/android/app/Activity.html#startActivityForResult\(android.content.Intent, int\)](#)). Simply call `setResult()` ([/reference/android/app/Activity.html# setResult\(int, android.content.Intent\)](#)) if the intent that started your activity might expect a result. If the originating activity had called `startActivityForResult()` ([/reference/android/app/Activity.html#startActivityForResult\(android.content.Intent, int\)](#)), then the system delivers it the result you supply to `setResult()` ([/reference/android/app/Activity.html# setResult\(int, android.content.Intent\)](#)); otherwise, the result is ignored.

# Advanced Training

Advanced Training contains a variety of classes that teach you best practices in Android development. These classes simplify the steps required to enhance your app with powerful platform features or effectively optimize your app performance.

What you see now is still the beginning. We plan to add many more classes, expand and refine existing classes, re-organize, and build courses that help you enhance your apps using objective-oriented collections of classes.

# Making Your App Location Aware

Users bring their mobile devices with them almost everywhere. One of the unique features available to mobile applications is location awareness. Knowing the location and using the information wisely can bring a more contextual experience to your users.

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.

## Lessons

---

### **Using the Location Manager**

Learn how to set up your application before it can receive location updates in Android.

### **Obtaining the Current Location**

Learn how to work with underlying location technologies available on the platform to obtain current location.

### **Displaying a Location Address**

Learn how to translate location coordinates into addresses that are readable to users.

---

### **DEPENDENCIES AND PREREQUISITES**

- Android 1.0 or higher (2.3+ for the sample app)

---

### **YOU SHOULD ALSO READ**

- [Location and Maps](#)

---

### **TRY IT OUT**

[Download the sample app](#)

LocationAware.zip

# Using the Location Manager

Before your application can begin receiving location updates, it needs to perform some simple steps to set up access. In this lesson, you'll learn what these steps entail.

## Declare Proper Permissions in Android Manifest

The first step of setting up location update access is to declare proper permissions in the manifest. If permissions are missing, the application will get a [SecurityException \(/reference/java/lang/SecurityException.html\)](#) at runtime.

Depending on the [LocationManager \(/reference/android/location/LocationManager.html\)](#) methods used, either [ACCESS\\_COARSE\\_LOCATION \(/reference/android/Manifest.permission.html#ACCESS\\_COARSE\\_LOCATION\)](#) or [ACCESS\\_FINE\\_LOCATION \(/reference/android/Manifest.permission.html#ACCESS\\_FINE\\_LOCATION\)](#) permission is needed. For example, you need to declare the [ACCESS\\_COARSE\\_LOCATION \(/reference/android/Manifest.permission.html#ACCESS\\_COARSE\\_LOCATION\)](#) permission if your application uses a network-based location provider only. The more accurate GPS requires the [ACCESS\\_FINE\\_LOCATION \(/reference/android/Manifest.permission.html#ACCESS\\_FINE\\_LOCATION\)](#) permission. Note that declaring the [ACCESS\\_FINE\\_LOCATION \(/reference/android/Manifest.permission.html#ACCESS\\_FINE\\_LOCATION\)](#) permission implies [ACCESS\\_COARSE\\_LOCATION \(/reference/android/Manifest.permission.html#ACCESS\\_COARSE\\_LOCATION\)](#) already.

Also, if a network-based location provider is used in the application, you'll need to declare the internet permission as well.

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.INTERNET" />
```

### THIS LESSON TEACHES YOU TO

1. [Declare Proper Permissions in Android Manifest](#)
2. [Get a Reference to LocationManager](#)
3. [Pick a Location Provider](#)
4. [Verify the Location Provider is Enabled](#)

### YOU SHOULD ALSO READ

- [Location and Maps](#)

### TRY IT OUT

[Download the sample app](#)

LocationAware.zip

## Get a Reference to LocationManager

[LocationManager \(/reference/android/location/LocationManager.html\)](#) is the main class through which your application can access location services on Android. Similar to other system services, a reference can be obtained from calling the [getSystemService\(\) \(/reference/android/content/Context.html#getSystemService\(java.lang.String\)\)](#) method. If your application intends to receive location updates in the foreground (within an [Activity \(/reference/android/app/Activity.html\)](#)), you

should usually perform this step in the `onCreate()` ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method.

```
LocationManager locationManager =  
    (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);
```

## Pick a Location Provider

---

While not required, most modern Android-powered devices can receive location updates through multiple underlying technologies, which are abstracted to an application as [LocationProvider](#) ([/reference/android/location/LocationProvider.html](#)) objects. Location providers may have different performance characteristics in terms of time-to-fix, accuracy, monetary cost, power consumption, and so on. Generally, a location provider with a greater accuracy, like the GPS, requires a longer fix time than a less accurate one, such as a network-based location provider.

Depending on your application's use case, you have to choose a specific location provider, or multiple providers, based on similar tradeoffs. For example, a points of interest check-in application would require higher location accuracy than say, a retail store locator where a city level location fix would suffice. The snippet below asks for a provider backed by the GPS.

```
LocationProvider provider =  
    locationManager.getProvider(LocationManager.GPS_PROVIDER);
```

Alternatively, you can provide some input criteria such as accuracy, power requirement, monetary cost, and so on, and let Android decide a closest match location provider. The snippet below asks for a location provider with fine accuracy and no monetary cost. Note that the criteria may not resolve to any providers, in which case a null will be returned. Your application should be prepared to gracefully handle the situation.

```
// Retrieve a list of location providers that have fine accuracy, no monetary cost  
Criteria criteria = new Criteria();  
criteria.setAccuracy(Criteria.ACCURACY_FINE);  
criteria.setCostAllowed(false);  
...  
String providerName = locManager.getBestProvider(criteria, true);  
  
// If no suitable provider is found, null is returned.  
if (providerName != null) {  
    ...  
}
```

## Verify the Location Provider is Enabled

---

Some location providers such as the GPS can be disabled in Settings. It is good practice to check

whether the desired location provider is currently enabled by calling the [isProviderEnabled\(\)](#) ([/reference/android/location/LocationManager.html#isProviderEnabled\(java.lang.String\)](#)) method. If the location provider is disabled, you can offer the user an opportunity to enable it in Settings by firing an [Intent](#) ([/reference/android/content/Intent.html](#)) with the [ACTION\\_LOCATION\\_SOURCE\\_SETTINGS](#) ([/reference/android/provider/Settings.html#ACTION\\_LOCATION\\_SOURCE\\_SETTINGS](#)) action.

```
@Override  
protected void onStart() {  
    super.onStart();  
  
    // This verification should be done during onStart() because the system calls  
    // this method when the user returns to the activity, which ensures the desir  
    // location provider is enabled each time the activity resumes from the stopp  
    LocationManager locationManager =  
        (LocationManager) getSystemService(Context.LOCATION_SERVICE);  
    final boolean gpsEnabled = locationManager.isProviderEnabled(LocationManager.  
  
    if (!gpsEnabled) {  
        // Build an alert dialog here that requests that the user enable  
        // the location services, then when the user clicks the "OK" button,  
        // call enableLocationSettings()  
    }  
}  
  
private void enableLocationSettings() {  
    Intent settingsIntent = new Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS);  
    startActivity(settingsIntent);  
}
```

# Obtaining the Current Location

After setting up your application to work with [LocationManager](#) ([/reference/android/location/LocationManager.html](#)), you can begin to obtain location updates.

## Set Up the Location Listener

The [LocationManager](#) ([/reference/android/location/LocationManager.html](#)) class exposes a number of methods for applications to receive location updates. In its simplest form, you register an event listener, identify the location manager from which you'd like to receive location updates, and specify the minimum time and distance intervals at which to receive location updates. The [onLocationChanged\(\)](#) ([/reference/android/location/LocationListener.html#onLocationChanged\(android.location.Location\)](#)) callback will be invoked with the frequency that correlates with time and distance intervals.

In the sample code snippet below, the location listener is set up to receive notifications at least every 10 seconds and if the device moves by more than 10 meters. The other callback methods notify the application any status change coming from the location provider.

```
private final LocationListener listener = new LocationListener() {  
  
    @Override  
    public void onLocationChanged(Location location) {  
        // A new location update is received. Do something useful with it. In this example we're sending the update to a handler which then updates the UI with the location.  
        Message.obtain(mHandler,  
                      UPDATE_LATLNG,  
                      location.getLatitude() + ", " +  
                      location.getLongitude()).sendToTarget();  
  
        ...  
    }  
};
```

### THIS LESSON TEACHES YOU TO

1. [Set Up the Location Listener](#)
2. [Handle Multiple Sources of Location Updates](#)
3. [Use getLastKnownLocation\(\) Wisely](#)
4. [Terminate Location Updates](#)

### YOU SHOULD ALSO READ

- [Location and Maps](#)

### TRY IT OUT

[Download the sample app](#)

LocationAware.zip

```
mLocationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,  
    10000, // 10-second interval.  
    10, // 10 meters.  
    listener);
```

## Handle Multiple Sources of Location Updates

Generally speaking, a location provider with greater accuracy (GPS) requires a longer fix time than one with lower accuracy (network-based). If you want to display location data as quickly as possible and update it as more accurate data becomes available, a common practice is to register a location listener with both GPS and network providers. In the [onLocationChanged\(\)](#) ([/reference/android/location/LocationListener.html#onLocationChanged\(android.location.Location\)](#)) callback, you'll receive location updates from multiple location providers that may have different timestamps and varying levels of accuracy. You'll need to incorporate logic to disambiguate the location providers and discard updates that are stale and less accurate. The code snippet below demonstrates a sample implementation of this logic.

```
private static final int TWO_MINUTES = 1000 * 60 * 2;  
  
/** Determines whether one Location reading is better than the current Location for  
 * @param location The new Location that you want to evaluate  
 * @param currentBestLocation The current Location fix, to which you want to compare  
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  
    // because the user has likely moved  
    if (isSignificantlyNewer) {  
        return true;  
    // If the new location is more than two minutes older, it must be worse  
    } else if (isSignificantlyOlder) {  
        return false;  
    }  
  
    // Check whether the new location fix is more or less accurate  
    int accuracyDelta = (int) (location.getAccuracy() - currentBestLocation.getAc
```

```
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 (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);
}
```

## Use getLastKnownLocation() Wisely

The setup time for getting a reasonable location fix may not be acceptable for certain applications. You should consider calling the [getLastKnownLocation\(\) \(/reference/android/location/LocationManager.html#getLastKnownLocation\(java.lang.String\)\)](#) method which simply queries Android for the last location update previously received by any location providers. Keep in mind that the returned location may be stale. You should check the timestamp and accuracy of the returned location and decide whether it is useful for your application. If you elect to discard the location update returned from [getLastKnownLocation\(\) \(/reference/android/location/LocationManager.html#getLastKnownLocation\(java.lang.String\)\)](#) and wait for fresh updates from the location provider(s), you should consider displaying an appropriate message before location data is received.

## Terminate Location Updates

When you are done with using location data, you should terminate location update to reduce unnecessary consumption of power and network bandwidth. For example, if the user navigates away from an activity where location updates are displayed, you should stop location update by calling [removeUpdates\(\) \(/reference/android/location/LocationManager.html#removeUpdates\(int\)\)](#)

[/LocationManager.html#removeUpdates\(android.location.LocationListener\)](#)) in [onStop\(\)](#) ([/reference/android/app/Activity.html#onStop\(\)](#)). [\(onStop\(\)\)](#) ([/reference/android/app/Activity.html#onStop\(\)](#)) is called when the activity is no longer visible. If you want to learn more about activity lifecycle, read up on the [Stopping and Restarting an Activity](#) ([/training/basics/activity-lifecycle/stopping.html](#)) lesson.

```
protected void onStop() {  
    super.onStop();  
    mLocationManager.removeUpdates(listener);  
}
```

Note: For applications that need to continuously receive and process location updates like a near-real time mapping application, it is best to incorporate the location update logic in a background service and make use of the system notification bar to make the user aware that location data is being used.

# Displaying the Location Address

As shown in previous lessons, location updates are received in the form of latitude and longitude coordinates. While this format is useful for calculating distance or displaying a pushpin on a map, the decimal numbers make no sense to most end users. If you need to display a location to user, it is much more preferable to display the address instead.

## Perform Reverse Geocoding

Reverse-geocoding is the process of translating latitude longitude coordinates to a human-readable address. The [Geocoder \(/reference/android/location/Geocoder.html\)](#) API is available for this purpose. Note that behind the scene, the API is dependent on a web service. If such service is unavailable on the device, the API will throw a "Service not Available exception" or return an empty list of addresses. A helper method called [isPresent\(\) \(/reference/android/location/Geocoder.html#isPresent\(\)\)](#) was added in Android 2.3 (API level 9) to check for the existence of the service.

The following code snippet demonstrates the use of the [Geocoder \(/reference/android/location/Geocoder.html\)](#) API to perform reverse-geocoding. Since the [getFromLocation\(\) \(/reference/android/location/Geocoder.html#getFromLocation\(double, double, int\)\)](#) method is synchronous, you should not invoke it from the UI thread, hence an [AsyncTask \(/reference/android/os/AsyncTask.html\)](#) is used in the snippet.

```
private final LocationListener listener = new LocationListener() {  
  
    public void onLocationChanged(Location location) {  
        // Bypass reverse-geocoding if the Geocoder service is not available on the  
        // device. The isPresent() convenient method is only available on Gingerbread.  
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD && Geocoder.  
            // Since the geocoding API is synchronous and may take a while. You  
            // up the UI thread. Invoking reverse geocoding in an AsyncTask.  
            (new ReverseGeocodingTask(this)).execute(new Location[] {location});  
    }  
}  
...  
  
// AsyncTask encapsulating the reverse-geocoding API. Since the geocoder API is  
// we do not want to invoke it from the UI thread.  
private class ReverseGeocodingTask extends AsyncTask<Location, Void, Void> {
```

### THIS LESSON TEACHES YOU TO

1. [Perform Reverse Geocoding](#)

### YOU SHOULD ALSO READ

- [Location and Maps](#)

### TRY IT OUT

[Download the sample app](#)

LocationAware.zip

```
Context mContext;

public ReverseGeocodingTask(Context context) {
    super();
    mContext = context;
}

@Override
protected Void doInBackground(Location... params) {
    Geocoder geocoder = new Geocoder(mContext, Locale.getDefault());

    Location loc = params[0];
    List<Address> addresses = null;
    try {
        // Call the synchronous getFromLocation() method by passing in the location.
        addresses = geocoder.getFromLocation(loc.getLatitude(), loc.getLongitude());
    } catch (IOException e) {
        e.printStackTrace();
        // Update UI field with the exception.
        Message.obtain(mHandler, UPDATE_ADDRESS, e.toString()).sendToTarget();
    }
    if (addresses != null && addresses.size() > 0) {
        Address address = addresses.get(0);
        // Format the first line of address (if available), city, and country.
        String addressText = String.format("%s, %s, %s",
                address.getMaxAddressLineIndex() > 0 ? address.getAddressLine(0),
                address.getLocality(),
                address.getCountryName());
        // Update the UI via a message handler.
        Message.obtain(mHandler, UPDATE_ADDRESS, addressText).sendToTarget();
    }
    return null;
}
```

# Performing Network Operations

This class explains the basic tasks involved in connecting to the network, monitoring the network connection (including connection changes), and giving users control over an app's network usage. It also describes how to parse and consume XML data.

This class includes a sample application that illustrates how to perform common network operations. You can download the sample (to the right) and use it as a source of reusable code for your own application.

By going through these lessons, you'll have the fundamental building blocks for creating Android applications that download content and parse data efficiently, while minimizing network traffic.

## Lessons

---

### [Connecting to the Network](#)

Learn how to connect to the network, choose an HTTP client, and perform network operations outside of the UI thread.

### [Managing Network Usage](#)

Learn how to check a device's network connection, create a preferences UI for controlling network usage, and respond to connection changes.

### [Parsing XML Data](#)

Learn how to parse and consume XML data.

---

### DEPENDENCIES AND PREREQUISITES

- Android 1.6 (API level 4) or higher
- A device that is able to connect to mobile and Wi-Fi networks

---

### YOU SHOULD ALSO READ

- [Optimizing Battery Life](#)
- [Transferring Data Without Draining the Battery](#)
- [Web Apps Overview](#)

---

### TRY IT OUT

[Download the sample](#)

NetworkUsage.zip

# Connecting to the Network

This lesson shows you how to implement a simple application that connects to the network. It explains some of the best practices you should follow in creating even the simplest network-connected app.

Note that to perform the network operations described in this lesson, your application manifest must include the following permissions:

```
<uses-permission android:name="android.permission.INTERNET">
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE">
```

## Choose an HTTP Client

Most network-connected Android apps use HTTP to send and receive data. Android includes two HTTP clients:

[HttpURLConnection](#) (/reference/java/net/[HttpURLConnection.html](#)) and Apache [HttpClient](#) (/reference/org/apache/http/client/[HttpClient.html](#)).

Both support HTTPS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling. We recommend using [HttpURLConnection](#) (/reference/java/net/[HttpURLConnection.html](#)) for applications targeted at Gingerbread and higher. For more discussion of this topic, see the blog post [Android's HTTP Clients](#) (<http://android-developers.blogspot.com/2011/09/androids-http-clients.html>).

## Check the Network Connection

Before your app attempts to connect to the network, it should check to see whether a network connection is available using [getActiveNetworkInfo\(\)](#) (/reference/android/net/[ConnectivityManager.html#.getActiveNetworkInfo\(\)](#)) and [isConnected\(\)](#) (/reference/android/net/[NetworkInfo.html#isConnected\(\)](#)). Remember, the device may be out of range of a network, or the user may have disabled both Wi-Fi and mobile data access. For more discussion of this topic, see the lesson [Managing Network Usage](#) (/training/basics/network-ops/managing.html).

```
public void myClickHandler(View view) {
    ...
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    if (networkInfo != null && networkInfo.isConnected()) {
        // fetch data
    }
}
```

### THIS LESSON TEACHES YOU TO

1. [Choose an HTTP Client](#)
2. [Check the Network Connection](#)
3. [Perform Network Operations on a Separate Thread](#)
4. [Connect and Download Data](#)
5. [Convert the InputStream to a String](#)

### YOU SHOULD ALSO READ

- [Optimizing Battery Life](#)
- [Transferring Data Without Draining the Battery](#)
- [Web Apps Overview](#)
- [Application Fundamentals](#)

```
    } else {
        // display error
    }
...
}
```

## Perform Network Operations on a Separate Thread

Network operations can involve unpredictable delays. To prevent this from causing a poor user experience, always perform network operations on a separate thread from the UI. The [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) class provides one of the simplest ways to fire off a new task from the UI thread. For more discussion of this topic, see the blog post [Multithreading For Performance](#) (<http://android-developers.blogspot.com/2010/07/multithreading-for-performance.html>).

In the following snippet, the `myClickHandler()` method invokes new `DownloadWebpageTask().execute(stringUrl)`. The `DownloadWebpageTask` class is a subclass of [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)). `DownloadWebpageTask` implements the following [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) methods:

- [`doInBackground\(\)`](#) executes the method `downloadUrl()`. It passes the web page URL as a parameter. The method `downloadUrl()` fetches and processes the web page content. When it finishes, it passes back a result string.
- [`onPostExecute\(\)`](#) takes the returned string and displays it in the UI.

```
public class HttpExampleActivity extends Activity {
    private static final String DEBUG_TAG = "HttpExample";
    private EditText urlText;
    private TextView textView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        urlText = (EditText) findViewById(R.id.myUrl);
        textView = (TextView) findViewById(R.id.myText);
    }

    // When user clicks button, calls AsyncTask.
    // Before attempting to fetch the URL, makes sure that there is a network connection.
    public void myClickHandler(View view) {
        // Gets the URL from the UI's text field.
        String stringUrl = urlText.getText().toString();
        ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
        if (networkInfo != null && networkInfo.isConnected()) {
            new DownloadWebpageText().execute(stringUrl);
        } else {
    }
```

```
        textView.setText("No network connection available.");
    }

// Uses AsyncTask to create a task away from the main UI thread. This task takes a URL string and uses it to create an HttpURLConnection. Once the connection has been established, the AsyncTask downloads the contents of the webpage as an InputStream. Finally, the InputStream is converted into a string, which is displayed in the UI by the AsyncTask's onPostExecute method.
private class DownloadWebpageText extends AsyncTask {
    @Override
    protected String doInBackground(String... urls) {

        // params comes from the execute() call: params[0] is the url.
        try {
            return downloadUrl(urls[0]);
        } catch (IOException e) {
            return "Unable to retrieve web page. URL may be invalid.";
        }
    }
    // onPostExecute displays the results of the AsyncTask.
    @Override
    protected void onPostExecute(String result) {
        textView.setText(result);
    }
}
...
}
```

The sequence of events in this snippet is as follows:

1. When users click the button that invokes myClickHandler(), the app passes the specified URL to the AsyncTask subclass DownloadWebpageTask.
2. The AsyncTask method doInBackground() calls the downloadUrl() method.
3. The downloadUrl() method takes a URL string as a parameter and uses it to create a URL object.
4. The URL object is used to establish an HttpURLConnection.
5. Once the connection has been established, the HttpURLConnection object fetches the web page content as an InputStream.
6. The InputStream is passed to the readIt() method, which converts the stream to a string.
7. Finally, the AsyncTask's onPostExecute() method displays the string in the main activity's UI.

## Connect and Download Data

In your thread that performs your network transactions, you can use HttpURLConnection (</reference/java/net/HttpURLConnection.html>) to perform a GET and download your data. After you call connect(), you can get an InputStream (</reference/java/io/InputStream.html>) of the data by calling

getInputStream().

In the following snippet, the [doInBackground\(\) \(/reference/android/os/AsyncTask.html#doInBackground\(Params...\)\)](#) method calls the method `downloadUrl()`. The `downloadUrl()` method takes the given URL and uses it to connect to the network via [HttpURLConnection \(/reference/java/net/HttpURLConnection.html\)](#). Once a connection has been established, the app uses the method `getInputStream()` to retrieve the data as an [InputStream \(/reference/java/io/InputStream.html\)](#).

```
// Given a URL, establishes an HttpURLConnection and retrieves
// the web page content as a InputStream, which it returns as
// a string.
private String downloadUrl(String myurl) throws IOException {
    InputStream is = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        // Starts the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        is = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString = readIt(is, len);
        return contentAsString;

        // Makes sure that the InputStream is closed after the app is
        // finished using it.
    } finally {
        if (is != null) {
            is.close();
        }
    }
}
```

Note that the method `getResponseCode()` returns the connection's [status code \(http://www.w3.org/Protocols/HTTP/HTRESP.html\)](#). This is a useful way of getting additional information about the connection. A status code of 200 indicates success.

## Convert the InputStream to a String

An [InputStream](#) (/reference/java/io/InputStream.html) is a readable source of bytes. Once you get an [InputStream](#) (/reference/java/io/InputStream.html), it's common to decode or convert it into a target data type. For example, if you were downloading image data, you might decode and display it like this:

```
InputStream is = null;
...
Bitmap bitmap = BitmapFactory.decodeStream(is);
ImageView imageView = (ImageView) findViewById(R.id.image_view);
imageView.setImageBitmap(bitmap);
```

In the example shown above, the [InputStream](#) (/reference/java/io/InputStream.html) represents the text of a web page. This is how the example converts the [InputStream](#) (/reference/java/io/InputStream.html) to a string so that the activity can display it in the UI:

```
// Reads an InputStream and converts it to a String.
public String readIt(InputStream stream, int len) throws IOException, UnsupportedEncodingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

# Managing Network Usage

This lesson describes how to write applications that have fine-grained control over their usage of network resources. If your application performs a lot of network operations, you should provide user settings that allow users to control your app's data habits, such as how often your app syncs data, whether to perform uploads/downloads only when on Wi-Fi, whether to use data while roaming, and so on. With these controls available to them, users are much less likely to disable your app's access to background data when they approach their limits, because they can instead precisely control how much data your app uses.

For general guidelines on how to write apps that minimize the battery life impact of downloads and network connections, see [Optimizing Battery Life](#) ([/training/monitoring-device-state/index.html](#)) and [Transferring Data Without Draining the Battery](#) ([/training/efficient-downloads/index.html](#)).

## Check a Device's Network Connection

A device can have various types of network connections. This lesson focuses on using either a Wi-Fi or a mobile network connection. For the full list of possible network types, see [ConnectivityManager](#) ([/reference/android/net/ConnectivityManager.html](#)).

Wi-Fi is typically faster. Also, mobile data is often metered, which can get expensive. A common strategy for apps is to only fetch large data if a Wi-Fi network is available.

Before you perform network operations, it's good practice to check the state of network connectivity. Among other things, this could prevent your app from inadvertently using the wrong radio. If a network connection is unavailable, your application should respond gracefully. To check the network connection, you typically use the following classes:

- [ConnectivityManager](#): Answers queries about the state of network connectivity. It also notifies applications when network connectivity changes.
- [NetworkInfo](#): Describes the status of a network interface of a given type (currently either Mobile or Wi-Fi).

This code snippet tests network connectivity for Wi-Fi and mobile. It determines whether these network interfaces are available (that is, whether network connectivity is possible) and/or connected (that is, whether network connectivity exists and if it is possible to establish sockets and pass data):

### THIS LESSON TEACHES YOU TO

1. [Check a Device's Network Connection](#)
2. [Manage Network Usage](#)
3. [Implement a Preferences Activity](#)
4. [Respond to Preference Changes](#)
5. [Detect Connection Changes](#)

### YOU SHOULD ALSO READ

- [Optimizing Battery Life](#)
- [Transferring Data Without Draining the Battery](#)
- [Web Apps Overview](#)

### TRY IT OUT

[Download the sample](#)

NetworkUsage.zip

```
private static final String DEBUG_TAG = "NetworkStatusExample";
...
ConnectivityManager connMgr = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();
networkInfo = connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
```

Note that you should not base decisions on whether a network is "available." You should always check [isConnected\(\)](#) before performing network operations, since [isConnected\(\)](#) handles cases like flaky mobile networks, airplane mode, and restricted background data.

A more concise way of checking whether a network interface is available is as follows. The method [getActiveNetworkInfo\(\)](#) returns a [NetworkInfo](#) instance representing the first connected network interface it can find, or null if none of the interfaces is connected (meaning that an internet connection is not available):

```
public boolean isOnline() {
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    return (networkInfo != null && networkInfo.isConnected());
}
```

To query more fine-grained state you can use [NetworkInfo.DetailedState](#), but this should seldom be necessary.

## Manage Network Usage

You can implement a preferences activity that gives users explicit control over your app's usage of network resources. For example:

- You might allow users to upload videos only when the device is connected to a Wi-Fi network.
- You might sync (or not) depending on specific criteria such as network availability, time interval, and so on.

To write an app that supports network access and managing network usage, your manifest must have the right permissions and intent filters.

- The manifest excerpted below includes the following permissions:
  - [android.permission.INTERNET](#)—Allows applications to open network sockets.
  - [android.permission.ACCESS\\_NETWORK\\_STATE](#)—Allows applications to access information about

networks.

- You can declare the intent filter for the `ACTION_MANAGE_NETWORK_USAGE` action (introduced in Android 4.0) to indicate that your application defines an activity that offers options to control data usage. `ACTION_MANAGE_NETWORK_USAGE` shows settings for managing the network data usage of a specific application. When your app has a settings activity that allows users to control network usage, you should declare this intent filter for that activity. In the sample application, this action is handled by the class `SettingsActivity`, which displays a preferences UI to let users decide when to download a feed.

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

    <uses-sdk android:minSdkVersion="4"
        android:targetSdkVersion="14" />

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        ...>
        ...
        <activity android:label="SettingsActivity" android:name=".SettingsActivity"
            <intent-filter>
                <action android:name="android.intent.action.MANAGE_NETWORK_USAGE"
                    <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

## Implement a Preferences Activity

As you can see in the manifest excerpt above, the sample app's activity `SettingsActivity` has an intent filter for the `ACTION_MANAGE_NETWORK_USAGE` ([/reference/android/content/Intent.html#ACTION\\_MANAGE\\_NETWORK\\_USAGE](#)) action. `SettingsActivity` is a subclass of `PreferenceActivity` ([/reference/android/preference/PreferenceActivity.html](#)). It displays a preferences screen (shown in figure 1) that lets users specify the following:

- Whether to display summaries for each XML feed entry, or just a link for each entry.
- Whether to download the XML feed if any network connection is available, or only if Wi-Fi is available.

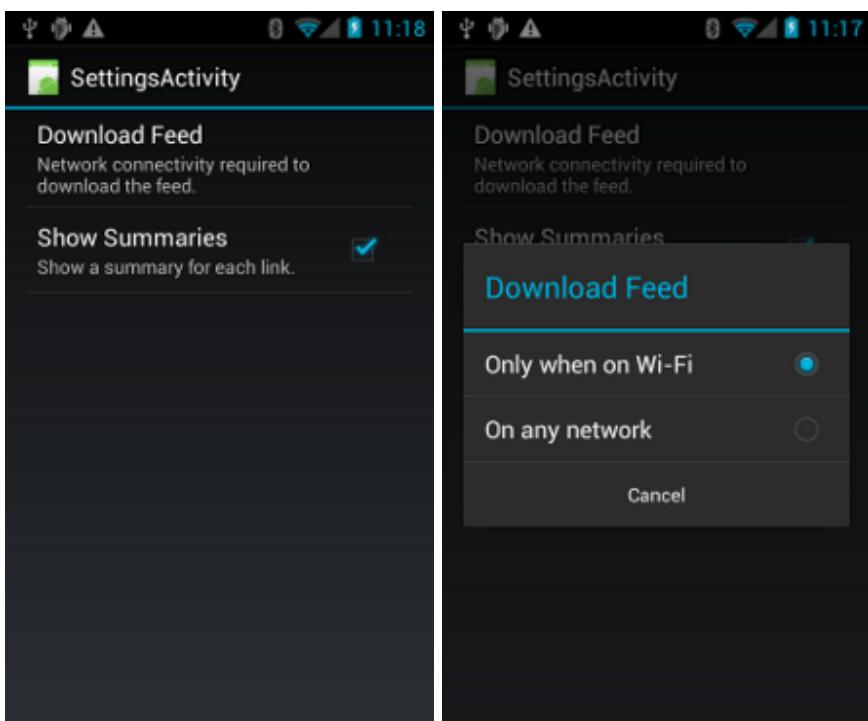


Figure 1. Preferences activity.

Here is `SettingsActivity`. Note that it implements [OnSharedPreferenceChangeListener](#) (`/reference/android/content/SharedPreferences.OnSharedPreferenceChangeListener.html`). When a user changes a preference, it fires [onSharedPreferenceChanged\(\)](#) (`/reference/android/content/SharedPreferences.OnSharedPreferenceChangeListener.html#onSharedPreferenceChanged(android.content.SharedPreferences, java.lang.String)`), which sets `refreshDisplay` to true. This causes the display to refresh when the user returns to the main activity:

```
public class SettingsActivity extends PreferenceActivity implements OnSharedPreferenceChangeListener {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        // Loads the XML preferences file  
        addPreferencesFromResource(R.xml.preferences);  
    }  
  
    @Override  
    protected void onResume() {  
        super.onResume();  
  
        // Registers a listener whenever a key changes  
        getPreferenceScreen().getSharedPreferences().registerOnSharedPreferenceChangeListener(this);  
    }  
  
    @Override  
    protected void onPause() {  
        super.onPause();  
    }  
}
```

```
super.onPause();

// Unregisters the listener set in onResume().
// It's best practice to unregister listeners when your app isn't using them
// unnecessarily system overhead. You do this in onPause().
getPreferenceScreen().getSharedPreferences().unregisterOnSharedPreferenceChangeListener(this);

// When the user changes the preferences selection,
// onSharedPreferenceChanged() restarts the main activity as a new
// task. Sets the refreshDisplay flag to "true" to indicate that
// the main activity should update its display.
// The main activity queries the PreferenceManager to get the latest settings

@Override
public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String key) {
    // Sets refreshDisplay to true so that when the user returns to the main
    // activity, the display refreshes to reflect the new settings.
    NetworkActivity.refreshDisplay = true;
}
}
```

## Respond to Preference Changes

When the user changes preferences in the settings screen, it typically has consequences for the app's behavior. In this snippet, the app checks the preferences settings in `onStart()`. If there is a match between the setting and the device's network connection (for example, if the setting is "Wi-Fi" and the device has a Wi-Fi connection), the app downloads the feed and refreshes the display.

```
public class NetworkActivity extends Activity {
    public static final String WIFI = "Wi-Fi";
    public static final String ANY = "Any";
    private static final String URL = "http://stackoverflow.com/feeds/tag?tagname=";

    // Whether there is a Wi-Fi connection.
    private static boolean wifiConnected = false;
    // Whether there is a mobile connection.
    private static boolean mobileConnected = false;
    // Whether the display should be refreshed.
    public static boolean refreshDisplay = true;

    // The user's current network preference setting.
    public static String sPref = null;

    // The BroadcastReceiver that tracks network connectivity changes.
    private NetworkReceiver receiver = new NetworkReceiver();
}
```

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

    // Registers BroadcastReceiver to track network connection changes.
    IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_A
receiver = new NetworkReceiver();
    this.registerReceiver(receiver, filter);
}

@Override
public void onDestroy() {
    super.onDestroy();
    // Unregisters BroadcastReceiver when app is destroyed.
    if (receiver != null) {
        this.unregisterReceiver(receiver);
    }
}

// Refreshes the display if the network connection and the
// pref settings allow it.

@Override
public void onStart () {
    super.onStart();

    // Gets the user's network preference settings
    SharedPreferences sharedPrefs = PreferenceManager.getDefaultSharedPrefere

    // Retrieves a string value for the preferences. The second parameter
    // is the default value to use if a preference value is not found.
    sPref = sharedPrefs.getString("listPref", "Wi-Fi");

    updateConnectedFlags();

    if(refreshDisplay){
        loadPage();
    }
}

// Checks the network connection and sets the wifiConnected and mobileConnect
// variables accordingly.
public void updateConnectedFlags() {
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);

    NetworkInfo activeInfo = connMgr.getActiveNetworkInfo();
    if (activeInfo != null && activeInfo.isConnected()) {
```

```
wifiConnected = activeInfo.getType() == ConnectivityManager.TYPE_WIFI  
mobileConnected = activeInfo.getType() == ConnectivityManager.TYPE_MO  
} else {  
    wifiConnected = false;  
    mobileConnected = false;  
}  
}  
  
// Uses AsyncTask subclass to download the XML feed from stackoverflow.com.  
public void loadPage() {  
    if (((sPref.equals(ANY)) && (wifiConnected || mobileConnected))  
        || ((sPref.equals(WIFI)) && (wifiConnected))) {  
        // AsyncTask subclass  
        new DownloadXmlTask().execute(URL);  
    } else {  
        showErrorPage();  
    }  
}  
...  
}
```

## Detect Connection Changes

The final piece of the puzzle is the [BroadcastReceiver](#) ([/reference/android/content/BroadcastReceiver.html](#)) subclass, NetworkReceiver. When the device's network connection changes, NetworkReceiver intercepts the action [CONNECTIVITY ACTION](#) ([/reference/android/net/ConnectivityManager.html#CONNECTIVITY ACTION](#)), determines what the network connection status is, and sets the flags wifiConnected and mobileConnected to true/false accordingly. The upshot is that the next time the user returns to the app, the app will only download the latest feed and update the display if NetworkActivity.refreshDisplay is set to true.

Setting up a BroadcastReceiver that gets called unnecessarily can be a drain on system resources. The sample application registers the [BroadcastReceiver](#) ([/reference/android/content/BroadcastReceiver.html](#)) NetworkReceiver in [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)), and it unregisters it in [onDestroy\(\)](#) ([/reference/android/app/Activity.html#onDestroy\(\)](#)). This is more lightweight than declaring a `<receiver>` in the manifest. When you declare a `<receiver>` in the manifest, it can wake up your app at any time, even if you haven't run it for weeks. By registering and unregistering NetworkReceiver within the main activity, you ensure that the app won't be woken up after the user leaves the app. If you do declare a `<receiver>` in the manifest and you know exactly where you need it, you can use [setComponentEnabledSetting\(\)](#) ([/reference/android/content/pm/PackageManager.html#setComponentEnabledSetting\(android.content.ComponentName, int, int\)](#)) to enable and disable it as appropriate.

Here is NetworkReceiver:

```
public class NetworkReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        ConnectivityManager conn = (ConnectivityManager)
            context.getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = conn.getActiveNetworkInfo();

        // Checks the user prefs and the network connection. Based on the result, dec-
        // to refresh the display or keep the current display.
        // If the userpref is Wi-Fi only, checks to see if the device has a Wi-Fi con-
        if (WIFI.equals(sPref) && networkInfo != null && networkInfo.getType() == Con-
            // If device has its Wi-Fi connection, sets refreshDisplay
            // to true. This causes the display to be refreshed when the user
            // returns to the app.
            refreshDisplay = true;
            Toast.makeText(context, R.string.wifi_connected, Toast.LENGTH_SHORT).show

        // If the setting is ANY network and there is a network connection
        // (which by process of elimination would be mobile), sets refreshDisplay to
        } else if (ANY.equals(sPref) && networkInfo != null) {
            refreshDisplay = true;

        // Otherwise, the app can't download content--either because there is no netw-
        // connection (mobile or Wi-Fi), or because the pref setting is WIFI, and the
        // is no Wi-Fi connection.
        // Sets refreshDisplay to false.
        } else {
            refreshDisplay = false;
            Toast.makeText(context, R.string.lost_connection, Toast.LENGTH_SHORT).show
        }
    }
}
```

# Parsing XML Data

Extensible Markup Language (XML) is a set of rules for encoding documents in machine-readable form. XML is a popular format for sharing data on the internet. Websites that frequently update their content, such as news sites or blogs, often provide an XML feed so that external programs can keep abreast of content changes. Uploading and parsing XML data is a common task for network-connected apps. This lesson explains how to parse XML documents and use their data.

## Choose a Parser

We recommend [XmlPullParser \(/reference/org/xmlpull/v1/XmlPullParser.html\)](#), which is an efficient and maintainable way to parse XML on Android. Historically Android has had two implementations of this interface:

- [KXmlParser](#) via `KXmlParserFactory.newPullParser()`.
- [ExpatPullParser](#), via `Xml.newPullParser()`.

Either choice is fine. The example in this section uses [ExpatPullParser](#), via `Xml.newPullParser() (/reference/android/util/Xml.html#newPullParser())`.

## Analyze the Feed

The first step in parsing a feed is to decide which fields you're interested in. The parser extracts data for those fields and ignores the rest.

Here is an excerpt from the feed that's being parsed in the sample app. Each post to [StackOverflow.com](#) (<http://stackoverflow.com>) appears in the feed as an entry tag that contains several nested tags:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:creativeCommons="http://backend.u
<title type="text">newest questions tagged android - Stack Overflow</title>
...
<entry>
  ...
</entry>
<entry>
  <id>http://stackoverflow.com/q/9439999</id>
```

### THIS LESSON TEACHES YOU TO

1. [Choose a Parser](#)
2. [Analyze the Feed](#)
3. [Instantiate the Parser](#)
4. [Read the Feed](#)
5. [Parse XML](#)
6. [Skip Tags You Don't Care About](#)
7. [Consume XML Data](#)

### YOU SHOULD ALSO READ

- [Web Apps Overview](#)

### TRY IT OUT

[Download the sample](#)

NetworkUsage.zip

```
<re:rank scheme="http://stackoverflow.com">0</re:rank>
<title type="text">Where is my data file?</title>
<category scheme="http://stackoverflow.com/feeds/tag?tagnames=android&sor
<category scheme="http://stackoverflow.com/feeds/tag?tagnames=android&sor
<author>
    <name>cliff2310</name>
    <uri>http://stackoverflow.com/users/1128925</uri>
</author>
<link rel="alternate" href="http://stackoverflow.com/questions/9439999/wh
<published>2012-02-25T00:30:54Z</published>
<updated>2012-02-25T00:30:54Z</updated>
<summary type="html">
    <p>I have an Application that requires a data file...</p>

    </summary>
</entry>
<entry>
...
</entry>
...
</feed>
```

The sample app extracts data for the entry tag and its nested tags title, link, and summary.

## Instantiate the Parser

The next step is to instantiate a parser and kick off the parsing process. In this snippet, a parser is initialized to not process namespaces, and to use the provided [InputStream](#) ([/reference/java/io/InputStream.html](#)) as its input. It starts the parsing process with a call to [nextTag\(\)](#) ([/reference/org/xmlpull/v1/XmlPullParser.html#nextTag\(\)](#)) and invokes the [readFeed\(\)](#) method, which extracts and processes the data the app is interested in:

```
public class StackOverflowXmlParser {
    // We don't use namespaces
    private static final String ns = null;

    public List parse(InputStream in) throws XmlPullParserException, IOException
        try {
            XmlPullParser parser = Xml.newPullParser();
            parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
            parser.setInput(in, null);
            parser.nextTag();
            return readFeed(parser);
        } finally {
            in.close();
        }
}
```

...  
}

## Read the Feed

The `readFeed()` method does the actual work of processing the feed. It looks for elements tagged "entry" as a starting point for recursively processing the feed. If a tag isn't an entry tag, it skips it. Once the whole feed has been recursively processed, `readFeed()` returns a [List](#) ([/reference/java/util/List.html](#)) containing the entries (including nested data members) it extracted from the feed. This [List](#) ([/reference/java/util/List.html](#)) is then returned by the parser.

```
private List readFeed(XmlPullParser parser) throws XmlPullParserException, IOException {
    List entries = new ArrayList();

    parser.require(XmlPullParser.START_TAG, ns, "feed");
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
            continue;
        }
        String name = parser.getName();
        // Starts by looking for the entry tag
        if (name.equals("entry")) {
            entries.add(readEntry(parser));
        } else {
            skip(parser);
        }
    }
    return entries;
}
```

## Parse XML

The steps for parsing an XML feed are as follows:

1. As described in [Analyze the Feed](#), identify the tags you want to include in your app. This example extracts data for the entry tag and its nested tags title, link, and summary.
2. Create the following methods:
  - o A "read" method for each tag you're interested in. For example, `readEntry()`, `readTitle()`, and so on. The parser reads tags from the input stream. When it encounters a tag named entry, title, link or summary, it calls the appropriate method for that tag. Otherwise, it skips the tag.
  - o Methods to extract data for each different type of tag and to advance the parser to the next tag. For example:
    - For the title and summary tags, the parser calls `readText()`. This method extracts data for these tags by calling `parser.getText()`.

- For the link tag, the parser extracts data for links by first determining if the link is the kind it's interested in. Then it uses `parser.getAttributeValue()` to extract the link's value.
- For the entry tag, the parser calls `readEntry()`. This method parses the entry's nested tags and returns an `Entry` object with the data members title, link, and summary.
- A helper `skip()` method that's recursive. For more discussion of this topic, see [Skip Tags You Don't Care About](#).

This snippet shows how the parser parses entries, titles, links, and summaries.

```
public static class Entry {  
    public final String title;  
    public final String link;  
    public final String summary;  
  
    private Entry(String title, String summary, String link) {  
        this.title = title;  
        this.summary = summary;  
        this.link = link;  
    }  
}  
  
// Parses the contents of an entry. If it encounters a title, summary, or link ta  
// to their respective "read" methods for processing. Otherwise, skips the tag.  
private Entry readEntry(XmlPullParser parser) throws XmlPullParserException, IOException {  
    parser.require(XmlPullParser.START_TAG, ns, "entry");  
    String title = null;  
    String summary = null;  
    String link = null;  
    while (parser.next() != XmlPullParser.END_TAG) {  
        if (parser.getEventType() != XmlPullParser.START_TAG) {  
            continue;  
        }  
        String name = parser.getName();  
        if (name.equals("title")) {  
            title = readTitle(parser);  
        } else if (name.equals("summary")) {  
            summary = readSummary(parser);  
        } else if (name.equals("link")) {  
            link = readLink(parser);  
        } else {  
            skip(parser);  
        }  
    }  
    return new Entry(title, summary, link);  
}  
  
// Processes title tags in the feed.  
private String readTitle(XmlPullParser parser) throws IOException, XmlPullParserException {  
    parser.require(XmlPullParser.START_TAG, ns, "title");  
}
```

```
String title = readText(parser);
parser.require(XmlPullParser.END_TAG, ns, "title");
return title;
}

// Processes link tags in the feed.
private String readLink(XmlPullParser parser) throws IOException, XmlPullParserException {
String link = "";
parser.require(XmlPullParser.START_TAG, ns, "link");
String tag = parser.getName();
String relType = parser.getAttributeValue(null, "rel");
if (tag.equals("link")) {
    if (relType.equals("alternate")){
        link = parser.getAttributeValue(null, "href");
        parser.nextTag();
    }
}
parser.require(XmlPullParser.END_TAG, ns, "link");
return link;
}

// Processes summary tags in the feed.
private String readSummary(XmlPullParser parser) throws IOException, XmlPullParserException {
parser.require(XmlPullParser.START_TAG, ns, "summary");
String summary = readText(parser);
parser.require(XmlPullParser.END_TAG, ns, "summary");
return summary;
}

// For the tags title and summary, extracts their text values.
private String readText(XmlPullParser parser) throws IOException, XmlPullParserException {
String result = "";
if (parser.next() == XmlPullParser.TEXT) {
    result = parser.getText();
    parser.nextTag();
}
return result;
}
...
}
```

## Skip Tags You Don't Care About

One of the steps in the XML parsing described above is for the parser to skip tags it's not interested in. Here is the parser's `skip()` method:

```
private void skip(XmlPullParser parser) throws XmlPullParserException, IOException {
```

```
if (parser.getEventType() != XmlPullParser.START_TAG) {
    throw new IllegalStateException();
}
int depth = 1;
while (depth != 0) {
    switch (parser.next()) {
        case XmlPullParser.END_TAG:
            depth--;
            break;
        case XmlPullParser.START_TAG:
            depth++;
            break;
    }
}
```

This is how it works:

- It throws an exception if the current event isn't a START\_TAG.
- It consumes the START\_TAG, and all events up to and including the matching END\_TAG.
- To make sure that it stops at the correct END\_TAG and not at the first tag it encounters after the original START\_TAG, it keeps track of the nesting depth.

Thus if the current element has nested elements, the value of depth won't be 0 until the parser has consumed all events between the original START\_TAG and its matching END\_TAG. For example, consider how the parser skips the <author> element, which has 2 nested elements, <name> and <uri>:

- The first time through the while loop, the next tag the parser encounters after <author> is the START\_TAG for <name>. The value for depth is incremented to 2.
- The second time through the while loop, the next tag the parser encounters is the END\_TAG </name>. The value for depth is decremented to 1.
- The third time through the while loop, the next tag the parser encounters is the START\_TAG <uri>. The value for depth is incremented to 2.
- The fourth time through the while loop, the next tag the parser encounters is the END\_TAG </uri>. The value for depth is decremented to 1.
- The fifth time and final time through the while loop, the next tag the parser encounters is the END\_TAG </author>. The value for depth is decremented to 0, indicating that the <author> element has been successfully skipped.

## Consume XML Data

The example application fetches and parses the XML feed within an [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)). This takes the processing off the main UI thread. When processing is complete, the app updates the UI in the main activity (NetworkActivity).

In the excerpt shown below, the `loadPage()` method does the following:

- Initializes a string variable with the URL for the XML feed.

- If the user's settings and the network connection allow it, invokes new `DownloadXmlTask().execute(url)`. This instantiates a new `DownloadXmlTask` object (`AsyncTask` subclass) and runs its `execute()` method, which downloads and parses the feed and returns a string result to be displayed in the UI.

```
public class NetworkActivity extends Activity {  
    public static final String WIFI = "Wi-Fi";  
    public static final String ANY = "Any";  
    private static final String URL = "http://stackoverflow.com/feeds/tag?tagname  
  
    // Whether there is a Wi-Fi connection.  
    private static boolean wifiConnected = false;  
    // Whether there is a mobile connection.  
    private static boolean mobileConnected = false;  
    // Whether the display should be refreshed.  
    public static boolean refreshDisplay = true;  
    public static String sPref = null;  
  
    ...  
  
    // Uses AsyncTask to download the XML feed from stackoverflow.com.  
    public void loadPage() {  
  
        if(sPref.equals(ANY) && (wifiConnected || mobileConnected)) {  
            new DownloadXmlTask().execute(URL);  
        }  
        else if ((sPref.equals(WIFI)) && (wifiConnected)) {  
            new DownloadXmlTask().execute(URL);  
        } else {  
            // show error  
        }  
    }  
}
```

The [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) subclass shown below, `DownloadXmlTask`, implements the following [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) methods:

- `doInBackground()` executes the method `loadXmlFromNetwork()`. It passes the feed URL as a parameter. The method `loadXmlFromNetwork()` fetches and processes the feed. When it finishes, it passes back a result string.
- `onPostExecute()` takes the returned string and displays it in the UI.

```
// Implementation of AsyncTask used to download XML feed from stackoverflow.com.  
private class DownloadXmlTask extends AsyncTask<String, Void, String> {  
    @Override  
    protected String doInBackground(String... urls) {  
        try {  
            return loadXmlFromNetwork(urls[0]);  
        } catch (IOException e) {  
            return getResources().getString(R.string.connection_error);  
        }  
    }  
}
```

```
        } catch (XmlPullParserException e) {
            return getResources().getString(R.string.xml_error);
        }
    }

@Override
protected void onPostExecute(String result) {
    setContentView(R.layout.main);
    // Displays the HTML string in the UI via a WebView
    WebView myWebView = (WebView) findViewById(R.id.webview);
    myWebView.loadData(result, "text/html", null);
}
}
```

Below is the method `loadXmlFromNetwork()` that is invoked from `DownloadXmlTask`. It does the following:

1. Instantiates a `StackOverflowXmlParser`. It also creates variables for a `List` of `Entry` objects (`entries`), and `title`, `url`, and `summary`, to hold the values extracted from the XML feed for those fields.
2. Calls `downloadUrl()`, which fetches the feed and returns it as an `InputStream`.
3. Uses `StackOverflowXmlParser` to parse the `InputStream`. `StackOverflowXmlParser` populates a `List` of `entries` with data from the feed.
4. Processes the entries `List`, and combines the feed data with HTML markup.
5. Returns an HTML string that is displayed in the main activity UI by the `AsyncTask` method `onPostExecute()`.

```
// Uploads XML from stackoverflow.com, parses it, and combines it with
// HTML markup. Returns HTML string.
private String loadXmlFromNetwork(String urlString) throws XmlPullParserException
{
    InputStream stream = null;
    // Instantiate the parser
    StackOverflowXmlParser stackOverflowXmlParser = new StackOverflowXmlParser();
    List<Entry> entries = null;
    String title = null;
    String url = null;
    String summary = null;
    Calendar rightNow = Calendar.getInstance();
    DateFormat formatter = new SimpleDateFormat("MMM dd h:mm:ss");

    // Checks whether the user set the preference to include summary text
    SharedPreferences sharedPrefs = PreferenceManager.getDefaultSharedPreferences();
    boolean pref = sharedPrefs.getBoolean("summaryPref", false);

    StringBuilder htmlString = new StringBuilder();
    htmlString.append("<h3>" + getResources().getString(R.string.page_title) + "</h3>");
    htmlString.append("<em>" + getResources().getString(R.string.updated) + " " +
                    formatter.format(rightNow.getTime()) + "</em>");
}
```

```
try {
    stream = downloadUrl(urlString);
    entries = stackOverflowXmlParser.parse(stream);
    // Makes sure that the InputStream is closed after the app is
    // finished using it.
} finally {
    if (stream != null) {
        stream.close();
    }
}

// StackOverflowXmlParser returns a List (called "entries") of Entry objects.
// Each Entry object represents a single post in the XML feed.
// This section processes the entries list to combine each entry with HTML ma
// Each entry is displayed in the UI as a link that optionally includes
// a text summary.
for (Entry entry : entries) {
    htmlString.append("<p><a href='");
    htmlString.append(entry.link);
    htmlString.append("'" + entry.title + "</a></p>");
    // If the user set the preference to include summary text,
    // adds it to the display.
    if (pref) {
        htmlString.append(entry.summary);
    }
}
return htmlString.toString();
}

// Given a string representation of a URL, sets up a connection and gets
// an input stream.
private InputStream downloadUrl(String urlString) throws IOException {
    URL url = new URL(urlString);
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setReadTimeout(10000 /* milliseconds */);
    conn.setConnectTimeout(15000 /* milliseconds */);
    conn.setRequestMethod("GET");
    conn.setDoInput(true);
    // Starts the query
    conn.connect();
    InputStream stream = conn.getInputStream();
}
```

# Transferring Data Without Draining the Battery

In this class you will learn to minimize the battery life impact of downloads and network connections, particularly in relation to the wireless radio.

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.

## DEPENDENCIES AND PREREQUISITES

- Android 2.0 (API Level 5) or higher

## YOU SHOULD ALSO READ

- [Optimizing Battery Life](#)

## Lessons

### **Optimizing Downloads for Efficient Network Access**

This lesson introduces the wireless radio state machine, explains how your app's connectivity model interacts with it, and how you can minimize your data connection and use prefetching and bundling to minimize the battery drain associated with your data transfers.

### **Minimizing the Effect of Regular Updates**

This lesson will examine how your refresh frequency can be varied to best mitigate the effect of background updates on the underlying wireless radio state machine.

### **Redundant Downloads are Redundant**

The most fundamental way to reduce your downloads is to download only what you need. This lesson introduces some best practices to eliminate redundant downloads.

### **Modifying your Download Patterns Based on the Connectivity Type**

When it comes to impact on battery life, not all connection types are created equal. Not only does the Wi-Fi radio use significantly less battery than its wireless radio counterparts, but the radios used in different wireless radio technologies have different battery implications.

# Optimizing Downloads for Efficient Network Access

Using the wireless radio to transfer data is potentially one of your app's most significant sources of battery drain. To minimize the battery drain associated with network activity, it's critical that you understand how your connectivity model will affect the underlying radio hardware.

This lesson introduces the wireless radio state machine and explains how your app's connectivity model interacts with it. It goes on to propose ways to minimize your data connections, use prefetching, and bundle your transfers in order to minimize the battery drain associated with your data transfers.

## The Radio State Machine

A fully active wireless radio consumes significant power, so it transitions between different energy states in order to conserve power when not in use, while attempting to minimize latency associated with "powering up" the radio when it's required.

The state machine for a typical 3G network radio consists of three energy states:

1. **Full power:** Used when a connection is active, allowing the device to transfer data at its highest possible rate.
2. **Low power:** An intermediate state that uses around 50% of the battery power at the full state.
3. **Standby:** The minimal energy state during which no network connection is active or required.

While the low and idle states drain significantly less battery, they also introduce significant latency to network requests. Returning to full power from the low state takes around 1.5 seconds, while moving from idle to full can take over 2 seconds.

To minimize latency, the state machine uses a delay to postpone the transition to lower energy states. Figure 1 uses AT&T's timings for a typical 3G radio.

### THIS LESSON TEACHES YOU TO

1. [Understand the radio state machine](#)
2. [Understand how apps can impact the radio state machine](#)
3. [Efficiently prefetch data](#)
4. [Batch transfers and connections](#)
5. [Reduce the number of connections you use](#)
6. [Use the DDMS Network Traffic Tool to identify areas of concern](#)

### YOU SHOULD ALSO READ

- [Optimizing Battery Life](#)

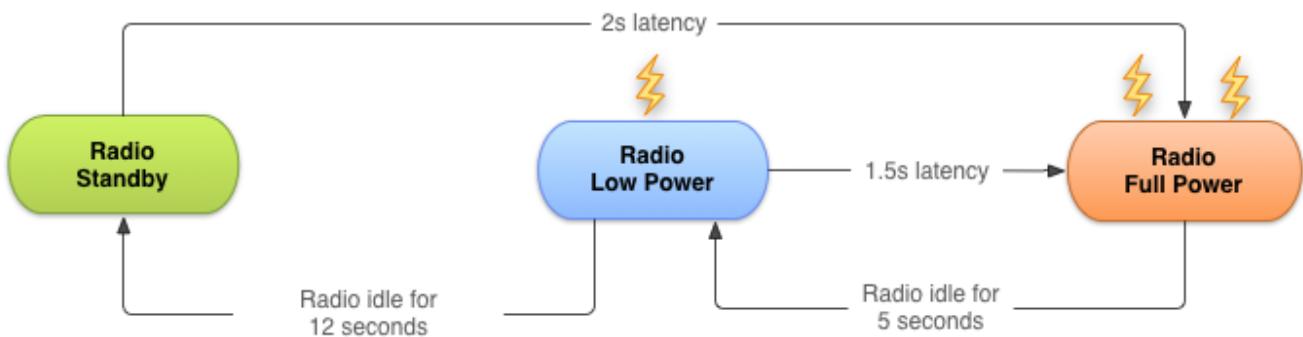


Figure 1. Typical 3G wireless radio state machine.

The radio state machine on each device, particularly the associated transition delay ("tail time") and startup latency, will vary based on the wireless radio technology employed (2G, 3G, LTE, etc.) and is defined and configured by the carrier network over which the device is operating.

This lesson describes a representative state machine for a typical 3G wireless radio, based on [data provided by AT&T](http://www.research.att.com/articles/featured_stories/2011_03_201102_Energy_efficient?fbid=1zObBOMOZSB) ([http://www.research.att.com/articles/featured\\_stories/2011\\_03\\_201102\\_Energy\\_efficient?fbid=1zObBOMOZSB](http://www.research.att.com/articles/featured_stories/2011_03_201102_Energy_efficient?fbid=1zObBOMOZSB)). However, the general principles and resulting best practices are applicable for all wireless radio implementations.

This approach is particularly effective for typical web browsing as it prevents unwelcome latency while users browse the web. The relatively low tail-time also ensures that once a browsing session has finished, the radio can move to a lower energy state.

Unfortunately, this approach can lead to inefficient apps on modern smartphone OSs like Android, where apps run both in the foreground (where latency is important) and in the background (where battery life should be prioritized).

## How Apps Impact the Radio State Machine

Every time you create a new network connection, the radio transitions to the full power state. In the case of the typical 3G radio state machine described above, it will remain at full power for the duration of your transfer—plus an additional 5 seconds of tail time—followed by 12 seconds at the low energy state. So for a typical 3G device, every data transfer session will cause the radio to draw energy for almost 20 seconds.

In practice, this means an app that transfers unbundled data for 1 second every 18 seconds will keep the wireless radio perpetually active, moving it back to high power just as it was about to become idle. As a result, every minute it will consume battery at the high power state for 18 seconds, and at the low power state for the remaining 42 seconds.

By comparison, the same app that bundles transfers of 3 seconds of every minute will keep the radio in the high power state for only 8 seconds, and will keep it in the low power state for only an additional 12 seconds.

The second example allows the radio to be idle for an additional 40 second every minute, resulting in a massive reduction in battery consumption.

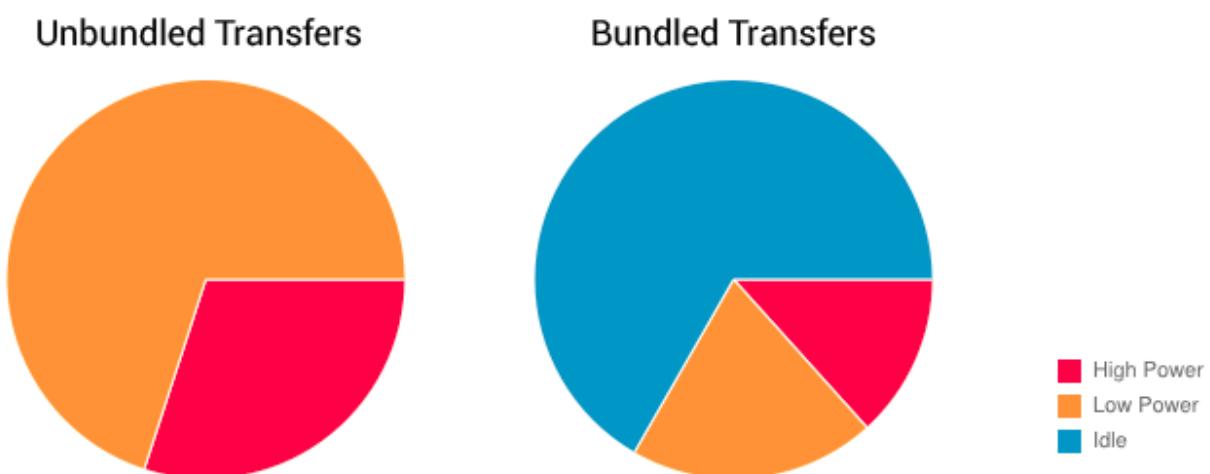


Figure 2. Relative wireless radio power use for bundled versus unbundled transfers.

## Prefetch Data

---

Prefetching data is an effective way to reduce the number of independent data transfer sessions. Prefetching allows you to download all the data you are likely to need for a given time period in a single burst, over a single connection, at full capacity.

By front loading your transfers, you reduce the number of radio activations required to download the data. As a result you not only conserve battery life, but also improve the latency, lower the required bandwidth, and reduce download times.

Prefetching also provides an improved user experience by minimizing in-app latency caused by waiting for downloads to complete before performing an action or viewing data.

However, used too aggressively, prefetching introduces the risk of increasing battery drain and bandwidth use—as well as download quota—by downloading data that isn't used. It's also important to ensure that prefetching doesn't delay application startup while the app waits for the prefetch to complete. In practical terms that might mean processing data progressively, or initiating consecutive transfers prioritized such that the data required for application startup is downloaded and processed first.

How aggressively you prefetch depends on the size of the data being downloaded and the likelihood of it being used. As a rough guide, based on the state machine described above, for data that has a 50% chance of being used within the current user session, you can typically prefetch for around 6 seconds (approximately 1-2 Mb) before the potential cost of downloading unused data matches the potential savings of not downloading that data to begin with.

Generally speaking, it's good practice to prefetch data such that you will only need to initiate another download every 2 to 5 minutes, and in the order of 1 to 5 megabytes.

Following this principle, large downloads—such as video files—should be downloaded in chunks at regular intervals (every 2 to 5 minutes), effectively prefetching only the video data likely to be viewed in the next few minutes.

Note that further downloads should be bundled, as described in the next section, [Batch Transfers](#)

and Connections (#BatchTransfers), and that these approximations will vary based on the connection type and speed, as discussed in [Modify your Download Patterns Based on the Connectivity Type \(connectivity\\_patterns.html\)](#).

Let's look at some practical examples:

### A music player

You could choose to prefetch an entire album, however should the user stop listening after the first song, you've wasted a significant amount of bandwidth and battery life.

A better approach would be to maintain a buffer of one song in addition to the one being played. For streaming music, rather than maintaining a continuous stream that keeps the radio active at all times, consider using HTTP live streaming to transmit the audio stream in bursts, simulating the prefetching approach described above.

### A news reader

Many news apps attempt to reduce bandwidth by downloading headlines only after a category has been selected, full articles only when the user wants to read them, and thumbnails just as they scroll into view.

Using this approach, the radio will be forced to remain active for the majority of users' news-reading session as they scroll headlines, change categories, and read articles. Not only that, but the constant switching between energy states will result in significant latency when switching categories or reading articles.

A better approach would be to prefetch a reasonable amount of data at startup, beginning with the first set of news headlines and thumbnails—ensuring a low latency startup time—and continuing with the remaining headlines and thumbnails, as well as the article text for each article available from at least the primary headline list.

Another alternative is to prefetch every headline, thumbnail, article text, and possibly even full article pictures—typically in the background on a predetermined schedule. This approach risks spending significant bandwidth and battery life downloading content that's never used, so it should be implemented with caution.

One solution is to schedule the full download to occur only when connected to Wi-Fi, and possibly only when the device is charging. This is investigated in more detail in [Modify your Download Patterns Based on the Connectivity Type \(connectivity\\_patterns.html\)](#).

## Batch Transfers and Connections

---

Every time you initiate a connection—irrespective of the size of the associated data transfer—you potentially cause the radio to draw power for nearly 20 seconds when using a typical 3G wireless radio.

An app that pings the server every 20 seconds, just to acknowledge that the app is running and visible to the user, will keep the radio powered on indefinitely, resulting in a significant battery cost for almost no actual data transfer.

With that in mind it's important to bundle your data transfers and create a pending transfer queue.

Done correctly, you can effectively phase-shift transfers that are due to occur within a similar time window, to make them all happen simultaneously—ensuring that the radio draws power for as short a duration as possible.

The underlying philosophy of this approach is to transfer as much data as possible during each transfer session in an effort to limit the number of sessions you require.

That means you should batch your transfers by queuing delay tolerant transfers, and preempting scheduled updates and prefetches, so that they are all executed when time-sensitive transfers are required. Similarly, your scheduled updates and regular prefetching should initiate the execution of your pending transfer queue.

For a practical example, let's return to the earlier examples from [Prefetch Data \(#PrefetchData\)](#).

Take a news application that uses the prefetching routine described above. The news reader collects analytics information to understand the reading patterns of its users and to rank the most popular stories. To keep the news fresh, it checks for updates every hour. To conserve bandwidth, rather than download full photos for each article, it prefetches only thumbnails and downloads the full photos when they are selected.

In this example, all the analytics information collected within the app should be bundled together and queued for download, rather than being transmitted as it's collected. The resulting bundle should be transferred when either a full-sized photo is being downloaded, or when an hourly update is being performed.

Any time-sensitive or on-demand transfer—such as downloading a full-sized image—should preempt regularly scheduled updates. The planned update should be executed at the same time as the on-demand transfer, with the next update scheduled to occur after the set interval. This approach mitigates the cost of performing a regular update by piggy-backing on the necessary time-sensitive photo download.

## Reduce Connections

---

It's generally more efficient to reuse existing network connections than to initiate new ones. Reusing connections also allows the network to more intelligently react to congestion and related network data issues.

Rather than creating multiple simultaneous connections to download data, or chaining multiple consecutive GET requests, where possible you should bundle those requests into a single GET.

For example, it would be more efficient to make a single request for every news article to be returned in a single request / response than to make multiple queries for several news categories. The wireless radio needs to become active in order to transmit the termination / termination acknowledgement packets associated with server and client timeout, so it's also good practice to close your connections when they aren't in use, rather than waiting for these timeouts.

That said, closing a connection too early can prevent it from being reused, which then requires additional overhead for establishing a new connection. A useful compromise is not to close the connection immediately, but to still close it before the inherent timeout expires.

## Use the DDMS Network Traffic Tool to Identify Areas of Concern

The Android [DDMS \(Dalvik Debug Monitor Server\)](#) ([/tools/debugging/ddms.html](#)) includes a Detailed Network Usage tab that makes it possible to track when your application is making network requests. Using this tool, you can monitor how and when your app transfers data and optimize the underlying code appropriately.

Figure 3 shows a pattern of transferring small amounts of data roughly 15 seconds apart, suggesting that efficiency could be dramatically improved by prefetching each request or bundling the uploads.



Figure 3. Tracking network usage with DDMS.

By monitoring the frequency of your data transfers, and the amount of data transferred during each connection, you can identify areas of your application that can be made more battery-efficient. Generally, you will be looking for short spikes that can be delayed, or that should cause a later transfer to be preempted.

To better identify the cause of transfer spikes, the Traffic Stats API allows you to tag the data transfers occurring within a thread using the `TrafficStats.setThreadStatsTag()` method, followed by manually tagging (and untagging) individual sockets using `tagSocket()` and `untagSocket()`. For example:

```
TrafficStats.setThreadStatsTag(0xF00D);
TrafficStats.tagSocket(outputSocket);
// Transfer data using socket
TrafficStats.untagSocket(outputSocket);
```

The Apache HttpClient and URLConnection libraries automatically tag sockets based on the current `getThreadStatsTag()` value. These libraries also tag and untag sockets when recycled through keep-alive pools.

```
TrafficStats.setThreadStatsTag(0xF00D);
try {
    // Make network request using HttpClient.execute()
} finally {
    TrafficStats.clearThreadStatsTag();
}
```

Socket tagging is supported in Android 4.0, but real-time stats will only be displayed on devices running Android 4.0.3 or higher.

# Minimizing the Effect of Regular Updates

The optimal frequency of regular updates will vary based on device state, network connectivity, user behavior, and explicit user preferences.

[Optimizing Battery Life \(/training/monitoring-device-state/index.html\)](#) discusses how to build battery-efficient apps that modify their refresh frequency based on the state of the host device. That includes disabling background service updates when you lose connectivity and reducing the rate of updates when the battery level is low.

This lesson will examine how your refresh frequency can be varied to best mitigate the effect of background updates on the underlying wireless radio state machine.

## THIS LESSON TEACHES YOU TO

1. [Use Google Cloud Messaging as an alternative to polling](#)
2. [Optimize polling with inexact repeating alarms and exponential back-offs](#)

## YOU SHOULD ALSO READ

- [Optimizing Battery Life](#)
- [Google Cloud Messaging for Android](#)

## Use Google Cloud Messaging as an Alternative to Polling

Every time your app polls your server to check if an update is required, you activate the wireless radio, drawing power unnecessarily, for up to 20 seconds on a typical 3G connection.

[Google Cloud Messaging for Android \(GCM\) \(/guide/google/gcm/index.html\)](#) is a lightweight mechanism used to transmit data from a server to a particular app instance. Using GCM, your server can notify your app running on a particular device that there is new data available for it.

Compared to polling, where your app must regularly ping the server to query for new data, this event-driven model allows your app to create a new connection only when it knows there is data to download.

The result is a reduction in unnecessary connections, and a reduced latency for updated data within your application.

GCM is implemented using a persistent TCP/IP connection. While it's possible to implement your own push service, it's best practice to use GCM. This minimizes the number of persistent connections and allows the platform to optimize bandwidth and minimize the associated impact on battery life.

## Optimize Polling with Inexact Repeating Alarms and Exponential Backoffs

Where polling is required, it's good practice to set the default data refresh frequency of your app as low as possible without detracting from the user experience.

A simple approach is to offer preferences to allow users to explicitly set their required update rate, allowing them to define their own balance between data freshness and battery life.

When scheduling updates, use inexact repeating alarms that allow the system to "phase shift" the exact moment each alarm triggers.

```
int alarmType = AlarmManager.ELAPSED_REALTIME;
long interval = AlarmManager.INTERVAL_HOUR;
long start = System.currentTimeMillis() + interval;

alarmManager.setInexactRepeating(alarmType, start, interval, pi);
```

If several alarms are scheduled to trigger at similar times, this phase-shifting will cause them to be triggered simultaneously, allowing each update to piggyback on top of a single active radio state change.

Wherever possible, set your alarm type to ELAPSED\_REALTIME or RTC rather than to their \_WAKEUP equivalents. This further reduces battery impact by waiting until the phone is no longer in standby mode before the alarm triggers.

You can further reduce the impact of these scheduled alarms by opportunistically reducing their frequency based on how recently your app was used.

One approach is to implement an exponential back-off pattern to reduce the frequency of your updates (and / or the degree of prefetching you perform) if the app hasn't been used since the previous update. It's often useful to assert a minimum update frequency and to reset the frequency whenever the app is used, for example:

```
SharedPreferences sp =
    context.getSharedPreferences(PREFS, Context.MODE_WORLD_READABLE);

boolean appUsed = sp.getBoolean(PREFS_APPUSED, false);
long updateInterval = sp.getLong(PREFS_INTERVAL, DEFAULT_REFRESH_INTERVAL);

if (!appUsed)
    if ((updateInterval *= 2) > MAX_REFRESH_INTERVAL)
        updateInterval = MAX_REFRESH_INTERVAL;

Editor spEdit = sp.edit();
spEdit.putBoolean(PREFS_APPUSED, false);
spEdit.putLong(PREFS_INTERVAL, updateInterval);
spEdit.apply();

rescheduleUpdates(updateInterval);
executeUpdateOrPrefetch();
```

You can use a similar exponential back-off pattern to reduce the effect of failed connections and download errors.

The cost of initiating a network connection is the same whether you are able to contact your server and download data or not. For time-sensitive transfers where successful completion is important, an exponential back-off algorithm can be used to reduce the frequency of retries in order to

minimize the associated battery impact, for example:

```
private void retryIn(long interval) {  
    boolean success = attemptTransfer();  
  
    if (!success) {  
        retryIn(interval*2 < MAX_RETRY_INTERVAL ?  
                interval*2 : MAX_RETRY_INTERVAL);  
    }  
}
```

Alternatively, for transfers that are failure tolerant (such as regular updates), you can simply ignore failed connection and transfer attempts.

# Redundant Downloads are Redundant

The most fundamental way to reduce your downloads is to download only what you need. In terms of data, that means implementing REST APIs that allow you to specify query criteria that limit the returned data by using parameters such as the time of your last update.

Similarly, when downloading images, it's good practice to reduce the size of the images server-side, rather than downloading full-sized images that are reduced on the client.

## THIS LESSON TEACHES YOU TO

1. [Cache files locally](#)
2. [Use the HttpURLConnection response cache](#)

## YOU SHOULD ALSO READ

- [Optimizing Battery Life](#)

## Cache Files Locally

Another important technique is to avoid downloading duplicate data. You can do this by aggressive caching. Always cache static resources, including on-demand downloads such as full size images, for as long as reasonably possible. On-demand resources should be stored separately to enable you to regularly flush your on-demand cache to manage its size.

To ensure that your caching doesn't result in your app displaying stale data, be sure to extract the time at which the requested content was last updated, and when it expires, from within the HTTP response headers. This will allow you to determine when the associated content should be refreshed.

```
long currentTime = System.currentTimeMillis();

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

long expires = conn.getHeaderFieldDate("Expires", currentTime);
long lastModified = conn.getHeaderFieldDate("Last-Modified", currentTime);

setDataExpirationDate(expires);

if (lastModified < lastUpdateTime) {
    // Skip update
} else {
    // Parse update
}
```

Using this approach, you can also effectively cache dynamic content while ensuring it doesn't result in your application displaying stale information.

You can cache non-sensitive data in the unmanaged external cache directory:

```
Context.getExternalCacheDir();
```

Alternatively, you can use the managed / secure application cache. Note that this internal cache may be flushed when the system is running low on available storage.

```
Context.getCache();
```

Files stored in either cache location will be erased when the application is uninstalled.

## Use the HttpURLConnection Response Cache

Android 4.0 added a response cache to HttpURLConnection. You can enable HTTP response caching on supported devices using reflection as follows:

```
private void enableHttpServletResponseCache() {
    try {
        long httpCacheSize = 10 * 1024 * 1024; // 10 MiB
        File httpCacheDir = new File(getCacheDir(), "http");
        Class.forName("android.net.http.HttpResponseCache")
            .getMethod("install", File.class, long.class)
            .invoke(null, httpCacheDir, httpCacheSize);
    } catch (Exception httpResponseCacheNotAvailable) {
        Log.d(TAG, "HTTP response cache is unavailable.");
    }
}
```

This sample code will turn on the response cache on Android 4.0+ devices without affecting earlier releases.

With the cache installed, fully cached HTTP requests can be served directly from local storage, eliminating the need to open a network connection. Conditionally cached responses can validate their freshness from the server, eliminating the bandwidth cost associated with the download.

Uncached responses get stored in the response cache for future requests.

# Modifying your Download Patterns Based on the Connectivity Type

When it comes to impact on battery life, not all connection types are created equal. Not only does the Wi-Fi radio use significantly less battery than its wireless radio counterparts, but the radios used in different wireless radio technologies have different battery implications.

## Use Wi-Fi

In most cases a Wi-Fi radio will offer greater bandwidth at a significantly lower battery cost. As a result, you should endeavor to perform data transfers when connected over Wi-Fi whenever possible.

You can use a broadcast receiver to listen for connectivity changes that indicate when a Wi-Fi connection has been established to execute significant downloads, preempt scheduled updates, and potentially even temporarily increase the frequency of regular updates as described in [Optimizing Battery Life \(/training/monitoring-device-state/index.html\)](#) lesson [Determining and Monitoring the Connectivity Status \(/training/monitoring-device-state/connectivity-monitoring.html\)](#).

## Use Greater Bandwidth to Download More Data Less Often

When connected over a wireless radio, higher bandwidth generally comes at the price of higher battery cost. Meaning that LTE typically consumes more energy than 3G, which is in turn more expensive than 2G.

This means that while the underlying radio state machine varies based on the radio technology, generally speaking the relative battery impact of the state change tail-time is greater for higher bandwidth radios.

At the same time, the higher bandwidth means you can prefetch more aggressively, downloading more data over the same time. Perhaps less intuitively, because the tail-time battery cost is relatively higher, it's also more efficient to keep the radio active for longer periods during each transfer session to reduce the frequency of updates.

For example, if an LTE radio has double the bandwidth and double the energy cost of 3G, you should download 4 times as much data during each session—or potentially as much as 10mb. When downloading this much data, it's important to consider the effect of your prefetching on the available local storage and flush your prefetch cache regularly.

You can use the connectivity manager to determine the active wireless radio, and modify your prefetching routines accordingly:

### THIS LESSON TEACHES YOU TO

1. [Use Wi-Fi](#)
2. [Use greater bandwidth to download more data less often](#)

### YOU SHOULD ALSO READ

- [Optimizing Battery Life](#)

```
ConnectivityManager cm =
(ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);

TelephonyManager tm =
(TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);

NetworkInfo activeNetwork = cm.getActiveNetworkInfo();

int PrefetchCacheSize = DEFAULT_PREFETCH_CACHE;

switch (activeNetwork.getType()) {
    case (ConnectivityManager.TYPE_WIFI):
        PrefetchCacheSize = MAX_PREFETCH_CACHE; break;
    case (ConnectivityManager.TYPE_MOBILE): {
        switch (tm.getNetworkType()) {
            case (TelephonyManager.NETWORK_TYPE_LTE |
                TelephonyManager.NETWORK_TYPE_HSPAP):
                PrefetchCacheSize *= 4;
                break;
            case (TelephonyManager.NETWORK_TYPE_EDGE |
                TelephonyManager.NETWORK_TYPE_GPRS):
                PrefetchCacheSize /= 2;
                break;
            default: break;
        }
        break;
    }
    default: break;
}
```

# Syncing to the Cloud

By providing powerful APIs for internet connectivity, the Android framework helps you build rich cloud-enabled apps that sync their data to a remote web service, making sure all your devices always stay in sync, and your valuable data is always backed up 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.

## Lessons

---

### Using the Backup API

Learn how to integrate the Backup API into your Android Application, so that user data such as preferences, notes, and high scores update seamlessly across all of a user's devices

### Making the Most of Google Cloud Messaging

Learn how to efficiently send multicast messages, react intelligently to incoming Google Cloud Messaging (GCM) messages, and use GCM messages to efficiently sync with the server.

# Using the Backup API

When a user purchases a new device or resets their existing one, they might expect that when Google Play restores your app back to their device during the initial setup, the previous data associated with the app restores as well. By default, that doesn't happen and all the user's accomplishments or settings in your app are lost.

For situations where the volume of data is relatively light (less than a megabyte), like the user's preferences, notes, game high scores or other stats, the Backup API provides a lightweight solution. This lesson walks you through integrating the Backup API into your application, and restoring data to new devices using the Backup API.

## THIS LESSON TEACHES YOU TO

1. [Register for the Android Backup Service](#)
2. [Configure Your Manifest](#)
3. [Write Your Backup Agent](#)
4. [Request a Backup](#)
5. [Restore from a Backup](#)

## YOU SHOULD ALSO READ

- [Data Backup](#)

## Register for the Android Backup Service

This lesson requires the use of the [Android Backup Service](http://code.google.com/android/backup/index.html) (<http://code.google.com/android/backup/index.html>), which requires registration. Go ahead and [register here](http://code.google.com/android/backup/signup.html) (<http://code.google.com/android/backup/signup.html>). Once that's done, the service pre-populates an XML tag for insertion in your Android Manifest, which looks like this:

```
<meta-data android:name="com.google.android.backup.api_key"  
        android:value="ABcDe1FGHij2KlmN3oPQRs4TUvW5xYZ" />
```

Note that each backup key works with a specific package name. If you have different applications, register separate keys for each one.

## Configure Your Manifest

Use of the Android Backup Service requires two additions to your application manifest. First, declare the name of the class that acts as your backup agent, then add the snippet above as a child element of the Application tag. Assuming your backup agent is going to be called TheBackupAgent, here's an example of what the manifest looks like with this tag included:

```
<application android:label="MyApp"  
            android:backupAgent="TheBackupAgent">  
    ...  
    <meta-data android:name="com.google.android.backup.api_key"  
              android:value="ABcDe1FGHij2KlmN3oPQRs4TUvW5xYZ" />
```

```
...  
</application>
```

## Write Your Backup Agent

The easiest way to create your backup agent is by extending the wrapper class [BackupAgentHelper](#) ([/reference/android/app/backup/BackupAgentHelper.html](#)). Creating this helper class is actually a very simple process. Just create a class with the same name as you used in the manifest in the previous step (in this example, `TheBackupAgent`), and extend `BackupAgentHelper`. Then override the [onCreate\(\)](#) ([/reference/android/app/backup/BackupAgent.html#onCreate\(\)](#)).

Inside the [onCreate\(\)](#) ([/reference/android/app/backup/BackupAgent.html#onCreate\(\)](#)) method, create a [BackupHelper](#) ([/reference/android/app/backup/BackupHelper.html](#)). These helpers are specialized classes for backing up certain kinds of data. The Android framework currently includes two such helpers: [FileBackupHelper](#) ([/reference/android/app/backup/FileBackupHelper.html](#)) and [SharedPreferencesBackupHelper](#) ([/reference/android/app/backup/SharedPreferencesBackupHelper.html](#)). After you create the helper and point it at the data you want to back up, just add it to the `BackupAgentHelper` using the [addHelper\(\)](#) ([/reference/android/app/backup/BackupAgentHelper.html#addHelper\(java.lang.String, android.app.backup.BackupHelper\)](#)) method, adding a key which is used to retrieve the data later. In most cases the entire implementation is perhaps 10 lines of code.

Here's an example that backs up a high scores file.

```
import android.app.backup.BackupAgentHelper;  
import android.app.backup.FileBackupHelper;  
  
public class TheBackupAgent extends BackupAgentHelper {  
    // The name of the SharedPreferences file  
    static final String HIGH_SCORES_FILENAME = "scores";  
  
    // 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  
    @Override  
    void onCreate() {  
        FileBackupHelper helper = new FileBackupHelper(this, HIGH_SCORES_FILENAME);  
        addHelper(FILES_BACKUP_KEY, helper);  
    }  
}
```

For added flexibility, [FileBackupHelper](#) ([/reference/android/app/backup/FileBackupHelper.html](#))'s constructor can take a variable number of filenames. You could just as easily have backed up both a high scores file and a game progress file just by adding an extra parameter, like this:

```
@Override  
void onCreate() {  
    FileBackupHelper helper = new FileBackupHelper(this, HIGH_SCORES_FILENAME  
        addHelper(FILES_BACKUP_KEY, helper);  
}
```

Backing up preferences is similarly easy. Create a [SharedPreferencesBackupHelper](#) ([/reference/android/app/backup/SharedPreferencesBackupHelper.html](#)) the same way you did a [FileBackupHelper](#) ([/reference/android/app/backup/FileBackupHelper.html](#)). In this case, instead of adding filenames to the constructor, add the names of the shared preference groups being used by your application. Here's an example of how your backup agent helper might look if high scores are implemented as preferences instead of a flat file:

```
import android.app.backup.BackupAgentHelper;  
import android.app.backup.SharedPreferencesBackupHelper;  
  
public class TheBackupAgent extends BackupAgentHelper {  
    // The names of the SharedPreferences groups that the application maintains.  
    // are the same strings that are passed to getSharedPreferences(String, int)  
    static final String PREFS_DISPLAY = "displayprefs";  
    static final String PREFS_SCORES = "highscores";  
  
    // An arbitrary string used within the BackupAgentHelper implementation to  
    // identify the SharedPreferencesBackupHelper's data.  
    static final String MY_PREFS_BACKUP_KEY = "myprefs";  
  
    // Simply allocate a helper and install it  
    void onCreate() {  
        SharedPreferencesBackupHelper helper =  
            new SharedPreferencesBackupHelper(this, PREFS_DISPLAY, PREFS_SCORES);  
        addHelper(MY_PREFS_BACKUP_KEY, helper);  
    }  
}
```

You can add as many backup helper instances to your backup agent helper as you like, but remember that you only need one of each type. One [FileBackupHelper](#) ([/reference/android/app/backup/FileBackupHelper.html](#)) handles all the files that you need to back up, and one [SharedPreferencesBackupHelper](#) ([/reference/android/app/backup/SharedPreferencesBackupHelper.html](#)) handles all the shared preference groups you need backed up.

## Request a Backup

In order to request a backup, just create an instance of the [BackupManager](#) ([/reference/android/app/backup/BackupManager.html](#)), and call its [dataChanged\(\)](#) ([/reference/android/app/backup/BackupManager.html#dataChanged\(\)](#)) method.

```
import android.app.backup.BackupManager;  
...  
  
public void requestBackup() {  
    BackupManager bm = new BackupManager(this);  
    bm.dataChanged();  
}
```

This call notifies the backup manager that there is data ready to be backed up to the cloud. At some point in the future, the backup manager then calls your backup agent's [onBackup\(\)](#) ([\(/reference/android/app/backup/BackupAgent.html#onBackup\(android.os.ParcelFileDescriptor, android.app.backup.BackupDataOutput, android.os.ParcelFileDescriptor\)\)](#)) method. You can make the call whenever your data has changed, without having to worry about causing excessive network activity. If you request a backup twice before a backup occurs, the backup only occurs once.

## Restore from a Backup

---

Typically you shouldn't ever have to manually request a restore, as it happens automatically when your application is installed on a device. However, if it *is* necessary to trigger a manual restore, just call the [requestRestore\(\)](#) ([\(/reference/android/app/backup/BackupManager.html#requestRestore\(android.app.backup.RestoreObserver\)\)](#)) method.

# Making the Most of Google Cloud Messaging

Google Cloud Messaging (GCM) is a free service for sending messages to Android devices. GCM messaging can greatly enhance the user experience. Your application can stay up to date without wasting battery power on waking up the radio and polling the server when there are no updates. Also, GCM allows you to attach up to 1,000 recipients to a single message, letting you easily contact large user bases quickly when appropriate, while minimizing the work load on your server.

This lesson covers some of the best practices for integrating GCM into your application, and assumes you are already familiar with basic implementation of this service. If this is not the case, you can read the [GCM demo app tutorial \(/guide/google/gcm/demo.html\)](#).

## THIS LESSON TEACHES YOU TO

1. [Send Multicast Messages Efficiently](#)
2. [Collapse Messages that can Be Replaced](#)
3. [Embed Data Directly in the GCM Message](#)
4. [React Intelligently to GCM Messages](#)

## YOU SHOULD ALSO READ

- [Google Cloud Messaging for Android](#)

## Send Multicast Messages Efficiently

One of the most useful features in GCM is support for up to 1,000 recipients for a single message. This capability makes it much easier to send out important messages to your entire user base. For instance, let's say you had a message that needed to be sent to 1,000,000 of your users, and your server could handle sending out about 500 messages per second. If you send each message with only a single recipient, it would take  $1,000,000/500 = 2,000$  seconds, or around half an hour. However, attaching 1,000 recipients to each message, the total time required to send a message out to 1,000,000 recipients becomes  $(1,000,000/1,000) / 500 = 2$  seconds. This is not only useful, but important for timely data, such as natural disaster alerts or sports scores, where a 30 minute interval might render the information useless.

Taking advantage of this functionality is easy. If you're using the [GCM helper library](#) (<http://developer.android.com/guide/google/gcm/gs.html#libs>) for Java, simply provide a `List` collection of registration IDs to the `send` or `sendNoRetry` method, instead of a single registration ID.

```
// This method name is completely fabricated, but you get the idea.
List regIds = whoShouldISendThisTo(message);

// If you want the SDK to automatically retry a certain number of times, use the
// standard send method.
MulticastResult result = sender.send(message, regIds, 5);

// Otherwise, use sendNoRetry.
MulticastResult result = sender.sendNoRetry(message, regIds);
```

For those implementing GCM support in a language other than Java, construct an HTTP POST

request with the following headers:

- Authorization: key=YOUR\_API\_KEY
- Content-type: application/json

Then encode the parameters you want into a JSON object, listing all the registration IDs under the key `registration_ids`. The snippet below serves as an example. All parameters except `registration_ids` are optional, and the items nested in `data` represent the user-defined payload, not GCM-defined parameters. The endpoint for this HTTP POST message will be <https://android.googleapis.com/gcm/send>.

```
{ "collapse_key": "score_update",
  "time_to_live": 108,
  "delay_while_idle": true,
  "data": {
    "score": "4 x 8",
    "time": "15:16.2342"
  },
  "registration_ids": ["4", "8", "15", "16", "23", "42"]
}
```

For a more thorough overview of the format of multicast GCM messages, see the [Sending Messages](#) (<http://developer.android.com/guide/google/gcm/gcm.html#send-msg>) section of the GCM guide.

## Collapse Messages that Can Be Replaced

GCM messages are often a tickle, telling the mobile application to contact the server for fresh data. In GCM, it's possible (and recommended) to create collapsible messages for this situation, wherein new messages replace older ones. Let's take the example of sports scores. If you send out a message to all users following a certain game with the updated score, and then 15 minutes later an updated score message goes out, the earlier one no longer matters. For any users who haven't received the first message yet, there's no reason to send both, and force the device to react (and possibly alert the user) twice when only one of the messages is still important.

When you define a collapse key, when multiple messages are queued up in the GCM servers for the same user, only the last one with any given collapse key is delivered. For a situation like with sports scores, this saves the device from doing needless work and potentially over-notifying the user. For situations that involve a server sync (like checking email), this can cut down on the number of syncs the device has to do. For instance, if there are 10 emails waiting on the server, and ten "new email" GCM tickles have been sent to the device, it only needs one, since it should only sync once.

In order to use this feature, just add a collapse key to your outgoing message. If you're using the GCM helper library, use the `Message` class's `collapseKey(String key)` method.

```
Message message = new Message.Builder(regId)
  .collapseKey("game4_scores") // The key for game 4.
```

```
.ttl(600) // Time in seconds to keep message queued if device offline.  
.delayWhileIdle(true) // Wait for device to become active before sending.  
.addPayload("key1", "value1")  
.addPayload("key2", "value2")  
.build();
```

If not using the helper library, simply add a variable to the POST header you're constructing, with collapse\_key as the field name, and the string you're using for that set of updates as the value.

## Embed Data Directly in the GCM Message

---

Often, GCM messages are meant to be a tickle, or indication to the device that there's fresh data waiting on a server somewhere. However, a GCM message can be up to 4kb in size, so sometimes it makes sense to simply send the data within the GCM message itself, so that the device doesn't need to contact the server at all. Consider this approach for situations where all of the following statements are true:

- The total data fits inside the 4kb limit.
- Each message is important, and should be preserved.
- It doesn't make sense to collapse multiple GCM messages into a single "new data on the server" tickle.

For instance, short messages or encoded player moves in a turn-based network game are examples of good use-cases for data to embed directly into a GCM message. Email is an example of a bad use-case, since messages are often larger than 4kb, and users don't need a GCM message for each email waiting for them on the server.

Also consider this approach when sending multicast messages, so you don't tell every device across your user base to hit your server for updates simultaneously.

This strategy isn't appropriate for sending large amounts of data, for a few reasons:

- Rate limits are in place to prevent malicious or poorly coded apps from spamming an individual device with messages.
- Messages aren't guaranteed to arrive in-order.
- Messages aren't guaranteed to arrive as fast as you send them out. Even if the device receives one GCM message a second, at a max of 1K, that's 8kbps, or about the speed of home dial-up internet in the early 1990's. Your app rating on Google Play will reflect having done that to your users.

When used appropriately, directly embedding data in the GCM message can speed up the perceived speediness of your application, by letting it skip a round trip to the server.

## React Intelligently to GCM Messages

---

Your application should not only react to incoming GCM messages, but react *intelligently*. How to react depends on the context.

### Don't be irritating

When it comes to alerting your user of fresh data, it's easy to cross the line from "useful" to "annoying". If your application uses status bar notifications, [update your existing notification](http://developer.android.com/guide/topics/ui/notifiers/notifications.html#Updating) (<http://developer.android.com/guide/topics/ui/notifiers/notifications.html#Updating>) instead of creating a second one. If you beep or vibrate to alert the user, consider setting up a timer. Don't let the application alert more than once a minute, lest users be tempted to uninstall your application, turn the device off, or toss it in a nearby river.

## **Sync smarter, not harder**

When using GCM as an indicator to the device that data needs to be downloaded from the server, remember you have 4kb of metadata you can send along to help your application be smart about it. For instance, if you have a feed reading app, and your user has 100 feeds that they follow, help the device be smart about what it downloads from the server! Look at the following examples of what metadata is sent to your application in the GCM payload, and how the application can react:

- refresh — Your app basically got told to request a dump of every feed it follows. Your app would either need to send feed requests to 100 different servers, or if you have an aggregator on your server, send a request to retrieve, bundle and transmit recent data from 100 different feeds, every time one updates.
- refresh, feedID — Better: Your app knows to check a specific feed for updates.
- refresh, feedID, timestamp — Best: If the user happened to manually refresh before the GCM message arrived, the application can compare timestamps of the most recent post, and determine that it *doesn't need to do anything*.

# Designing for Multiple Screens

Android powers hundreds of device types with several different screen sizes, ranging from small phones to large TV sets. Therefore, it's important that you design your application to be compatible with all screen sizes so it's available to as many users as possible.

But being compatible with different device types is not enough. Each screen size offers different possibilities and challenges for user interaction, so in order to truly satisfy and impress your users, your application must go beyond merely *supporting* multiple screens: it must *optimize* the user experience for each screen configuration.

This class shows you how to implement a user interface that's optimized for several screen configurations.

The code in each lesson comes from a sample application that demonstrates best practices in optimizing for multiple screens. You can download the sample (to the right) and use it as a source of reusable code for your own application.

Note: This class and the associated sample use the [support library](#) ([/tools/extras/support-library.html](#)) in order to use the [Fragment](#) ([/reference/android/app/Fragment.html](#)) APIs on versions lower than Android 3.0. You must download and add the library to your application in order to use all APIs in this class.

## Lessons

### **Supporting Different Screen Sizes**

This lesson walks you through how to design layouts that adapt several different screen sizes (using flexible dimensions for views, [RelativeLayout](#), screen size and orientation qualifiers, alias filters, and nine-patch bitmaps).

### **Supporting Different Screen Densities**

This lesson shows you how to support screens that have different pixel densities (using density-independent pixels and providing bitmaps appropriate for each density).

### **Implementing Adaptive UI Flows**

This lesson shows you how to implement your UI flow in a way that adapts to several screen size/density combinations (run-time detection of active layout, reacting according to current layout, handling screen configuration changes).

### **DEPENDENCIES AND PREREQUISITES**

- Android 1.6 or higher (2.1+ for the sample app)
- Basic knowledge of [Activities](#) and [Fragments](#)
- Experience building an [Android User Interface](#)
- Several features require the use of the [support library](#)

### **YOU SHOULD ALSO READ**

- [Supporting Multiple Screens](#)

### **TRY IT OUT**

[Download the sample app](#)

NewsReader.zip

# Supporting Different Screen Sizes

This lesson shows you how to support different screen sizes by:

- Ensuring your layout can be adequately resized to fit the screen
- Providing appropriate UI layout according to screen configuration
- Ensuring the correct layout is applied to the correct screen
- Providing bitmaps that scale correctly

## Use "wrap\_content" and "match\_parent"

To ensure that your layout is flexible and adapts to different screen sizes, you should use "wrap\_content" and "match\_parent" for the width and height of some view components. If you use "wrap\_content", the width or height of the view is set to the minimum size necessary to fit the content within that view, while "match\_parent" (also known as "fill\_parent" before API level 8) makes the component expand to match the size of its parent view.

By using the "wrap\_content" and "match\_parent" size values instead of hard-coded sizes, your views either use only the space required for that view or expand to fill the available space, respectively. For example:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
    <LinearLayout android:layout_width="match_parent"  
        android:id="@+id/linearLayout1"  
        android:gravity="center"  
        android:layout_height="50dp">  
        <ImageView android:id="@+id/imageView1"  
            android:layout_height="wrap_content"  
            android:layout_width="wrap_content"  
            android:src="@drawable/logo"  
            android:paddingRight="30dp"  
            android:layout_gravity="left"  
            android:layout_weight="0" />  
        <View android:layout_height="wrap_content"
```

### THIS LESSON TEACHES YOU TO

1. [Use "wrap\\_content" and "match\\_parent"](#)
2. [Use RelativeLayout](#)
3. [Use Size Qualifiers](#)
4. [Use the Smallest-width Qualifier](#)
5. [Use Layout Aliases](#)
6. [Use Orientation Qualifiers](#)
7. [Use Nine-patch Bitmaps](#)

### YOU SHOULD ALSO READ

- [Supporting Multiple Screens](#)

### TRY IT OUT

[Download the sample app](#)

NewsReader.zip

```
        android:id="@+id/view1"
        android:layout_width="wrap_content"
        android:layout_weight="1" />
    <Button android:id="@+id/categorybutton"
        android:background="@drawable/button_bg"
        android:layout_height="match_parent"
        android:layout_weight="0"
        android:layout_width="120dp"
        style="@style/CategoryButtonStyle"/>
</LinearLayout>

<fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />
</LinearLayout>
```

Notice how the sample uses "wrap\_content" and "match\_parent" for component sizes rather than specific dimensions. This allows the layout to adapt correctly to different screen sizes and orientations.

For example, this is what this layout looks like in portrait and landscape mode. Notice that the sizes of the components adapt automatically to the width and height:

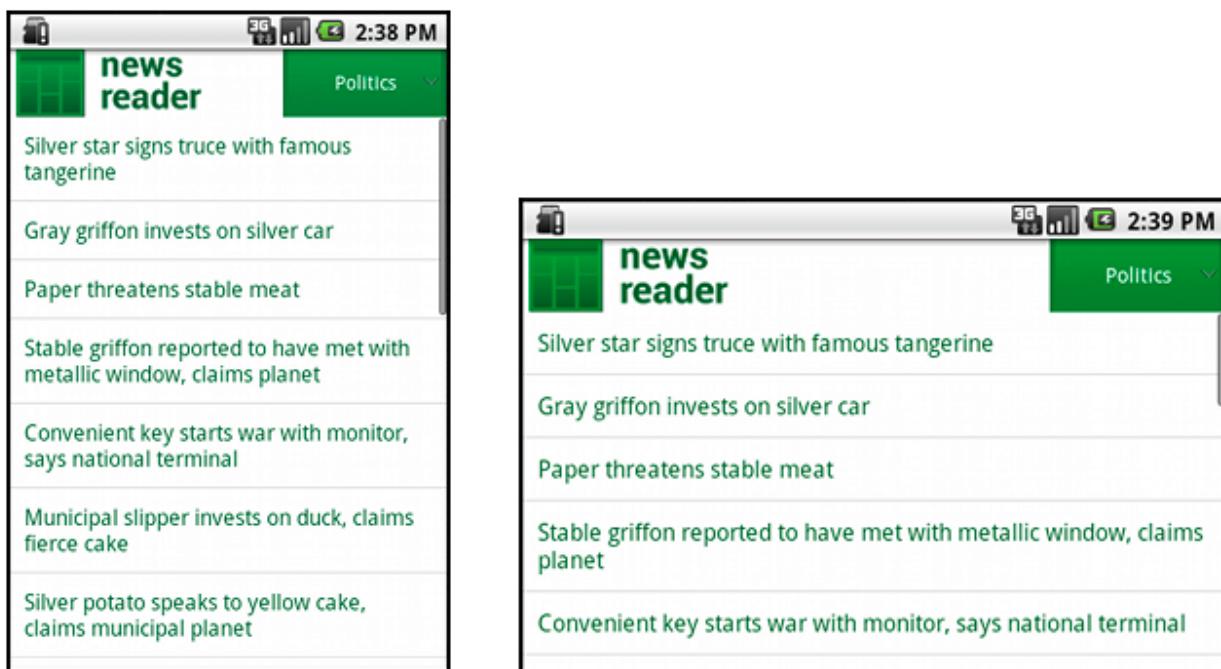


Figure 1. The News Reader sample app in portrait (left) and landscape (right).

## Use RelativeLayout

You can construct fairly complex layouts using nested instances of [LinearLayout](#) (/reference)

[/android/widget/LinearLayout.html](#) and combinations of "wrap\_content" and "match\_parent" sizes. However, [LinearLayout](#) does not allow you to precisely control the spatial relationships of child views; views in a [LinearLayout](#) simply line up side-by-side. If you need child views to be oriented in variations other than a straight line, a better solution is often to use a [RelativeLayout](#), which allows you to specify your layout in terms of the spatial relationships between components. For instance, you can align one child view on the left side and another view on the right side of the screen.

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/label"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Type here:"/>
    <EditText
        android:id="@+id/entry"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/label"/>
    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/entry"
        android:layout_alignParentRight="true"
        android:layout_marginLeft="10dp"
        android:text="OK" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@+id/ok"
        android:layout_alignTop="@+id/ok"
        android:text="Cancel" />
</RelativeLayout>
```

Figure 2 shows how this layout appears on a QVGA screen.



Figure 2. Screenshot on a QVGA screen (small screen).

Figure 3 shows how it appears on a larger screen.



Figure 3. Screenshot on a WSVGA screen (large screen).

Notice that although the size of the components changed, their spatial relationships are preserved as specified by the [RelativeLayout.LayoutParams](#) ([/reference/android/widget/RelativeLayout.LayoutParams.html](#)).

## Use Size Qualifiers

There's only so much mileage you can get from a flexible layout or relative layout like the one in the previous sections. While those layouts adapt to different screens by stretching the space within and around components, they may not provide the best user experience for each screen size. Therefore, your application should not only implement flexible layouts, but should also provide several alternative layouts to target different screen configurations. You do so by using [configuration qualifiers](http://developer.android.com/guide/practices/screens_support.html#qualifiers) ([http://developer.android.com/guide/practices/screens\\_support.html#qualifiers](http://developer.android.com/guide/practices/screens_support.html#qualifiers)), which allows the runtime to automatically select the appropriate resource based on the current device's configuration (such as a different layout design for different screen sizes).

For example, many applications implement the "two pane" pattern for large screens (the app might show a list of items on one pane and the content on another pane). Tablets and TVs are large enough for both panes to fit simultaneously on screen, but phone screens have to show them separately. So, to implement these layouts, you could have the following files:

- res/layout/main.xml, single-pane (default) layout:

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

    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />
</LinearLayout>
```

- res/layout-large/main.xml, two-pane layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="400dp"
        android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.ArticleFragment"
        android:layout_width="fill_parent" />
</LinearLayout>
```

Notice the `large` qualifier in the directory name of the second layout. This layout will be selected on devices with screens classified as large (for example, 7" tablets and above). The other layout (without qualifiers) will be selected for smaller devices.

## Use the Smallest-width Qualifier

One of the difficulties developers had in pre-3.2 Android devices was the "large" screen size bin, which encompasses the Dell Streak, the original Galaxy Tab, and 7" tablets in general. However, many applications may want to show different layouts for different devices in this category (such as for 5" and 7" devices), even though they are all considered to be "large" screens. That's why Android introduced the "Smallest-width" qualifier (amongst others) in Android 3.2.

The Smallest-width qualifier allows you to target screens that have a certain minimum width given in dp. For example, the typical 7" tablet has a minimum width of 600 dp, so if you want your UI to have two panes on those screens (but a single list on smaller screens), you can use the same two layouts from the previous section for single and two-pane layouts, but instead of the large size qualifier, use sw600dp to indicate the two-pane layout is for screens on which the smallest-width is 600 dp:

- res/layout/main.xml, single-pane (default) layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <fragment android:id="@+id/headlines"  
        android:layout_height="fill_parent"  
        android:name="com.example.android.newsreader.HeadlinesFragment"  
        android:layout_width="match_parent" />  
/<LinearLayout>
```

- res/layout-sw600dp/main.xml, two-pane layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent"  
    android:orientation="horizontal">  
    <fragment android:id="@+id/headlines"  
        android:layout_height="fill_parent"  
        android:name="com.example.android.newsreader.HeadlinesFragment"  
        android:layout_width="400dp"  
        android:layout_marginRight="10dp"/>  
    <fragment android:id="@+id/article"  
        android:layout_height="fill_parent"  
        android:name="com.example.android.newsreader.ArticleFragment"  
        android:layout_width="fill_parent" />  
/<LinearLayout>
```

This means that devices whose smallest width is greater than or equal to 600dp will select the layout-sw600dp/main.xml (two-pane) layout, while smaller screens will select the layout/main.xml (single-pane) layout.

However, this won't work well on pre-3.2 devices, because they don't recognize sw600dp as a size

qualifier, so you still have to use the large qualifier as well. So, you should have a file named res/layout-large/main.xml which is identical to res/layout-sw600dp/main.xml. In the next section you'll see a technique that allows you to avoid duplicating the layout files this way.

## Use Layout Aliases

---

The smallest-width qualifier is available only on Android 3.2 and above. Therefore, you should also still use the abstract size bins (small, normal, large and xlarge) to be compatible with earlier versions. For example, if you want to design your UI so that it shows a single-pane UI on phones but a multi-pane UI on 7" tablets, TVs and other large devices, you'd have to supply these files:

- res/layout/main.xml: single-pane layout
- res/layout-large: multi-pane layout
- res/layout-sw600dp: multi-pane layout

The last two files are identical, because one of them will be matched by Android 3.2 devices, and the other one is for the benefit of tablets and TVs with earlier versions of Android.

To avoid this duplication of the same file for tablets and TVs (and the maintenance headache resulting from it), you can use alias files. For example, you can define the following layouts:

- res/layout/main.xml, single-pane layout
- res/layout/main\_twopanes.xml, two-pane layout

And add these two files:

- res/values-large/layout.xml:

```
<resources>
    <item name="main" type="layout">@layout/main_twopanes</item>
</resources>
```

- res/values-sw600dp/layout.xml:

```
<resources>
    <item name="main" type="layout">@layout/main_twopanes</item>
</resources>
```

These latter two files have identical content, but they don't actually define the layout. They merely set up main to be an alias to main\_twopanes. Since these files have large and sw600dp selectors, they are applied to tablets and TVs regardless of Android version (pre-3.2 tablets and TVs match large, and post-3.2 will match sw600dp).

## Use Orientation Qualifiers

---

Some layouts work well in both landscape and portrait orientations, but most of them can benefit from adjustments. In the News Reader sample app, here is how the layout behaves in each screen size and orientation:

- **small screen, portrait:** single pane, with logo
- **small screen, landscape:** single pane, with logo
- **7" tablet, portrait:** single pane, with action bar
- **7" tablet, landscape:** dual pane, wide, with action bar
- **10" tablet, portrait:** dual pane, narrow, with action bar
- **10" tablet, landscape:** dual pane, wide, with action bar
- **TV, landscape:** dual pane, wide, with action bar

So each of these layouts is defined in an XML file in the res/layout/ directory. To then assign each layout to the various screen configurations, the app uses layout aliases to match them to each configuration:

res/layout/onepane.xml:

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

    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />
</LinearLayout>
```

res/layout/onepane\_with\_bar.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout android:layout_width="match_parent"
        android:id="@+id/linearLayout1"
        android:gravity="center"
        android:layout_height="50dp">
        <ImageView android:id="@+id/imageView1"
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:src="@drawable/logo"
            android:paddingRight="30dp"
            android:layout_gravity="left"
            android:layout_weight="0" />
        <View android:layout_height="wrap_content"
            android:id="@+id/view1"
            android:layout_width="wrap_content"
            android:layout_weight="1" />
        <Button android:id="@+id/categorybutton"
            android:background="@drawable/button_bg"
```

```
        android:layout_height="match_parent"
        android:layout_weight="0"
        android:layout_width="120dp"
        style="@style/CategoryButtonStyle"/>
    
```

```
</LinearLayout>

<fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="match_parent" />

```

```
</LinearLayout>
```

res/layout/twopanes.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="400dp"
        android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.ArticleFragment"
        android:layout_width="fill_parent" />

```

```
</LinearLayout>
```

res/layout/twopanes\_narrow.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="200dp"
        android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.ArticleFragment"
        android:layout_width="fill_parent" />

```

```
</LinearLayout>
```

Now that all possible layouts are defined, it's just a matter of mapping the correct layout to each

configuration using the configuration qualifiers. You can now do it using the layout alias technique:

res/values/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane_with_bar</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

res/values-sw600dp-land/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

res/values-sw600dp-port/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/onepane</item>
    <bool name="has_two_panes">false</bool>
</resources>
```

res/values-large-land/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

res/values-large-port/layouts.xml:

```
<resources>
    <item name="main_layout" type="layout">@layout/twopanes_narrow</item>
    <bool name="has_two_panes">true</bool>
</resources>
```

## Use Nine-patch Bitmaps

Supporting different screen sizes usually means that your image resources must also be capable of adapting to different sizes. For example, a button background must fit whichever button shape it is applied to.

If you use simple images on components that can change size, you will quickly notice that the

results are somewhat less than impressive, since the runtime will stretch or shrink your images uniformly. The solution is using nine-patch bitmaps, which are specially formatted PNG files that indicate which areas can and cannot be stretched.

Therefore, when designing bitmaps that will be used on components with variable size, always use nine-patches. To convert a bitmap into a nine-patch, you can start with a regular image (figure 4, shown with 4x zoom for clarity).



Figure 4. button.png

And then run it through the `draw9patch` utility of the SDK (which is located in the `tools/` directory), in which you can mark the areas that should be stretched by drawing pixels along the left and top borders. You can also mark the area that should hold the content by drawing pixels along the right and bottom borders, resulting in figure 5.

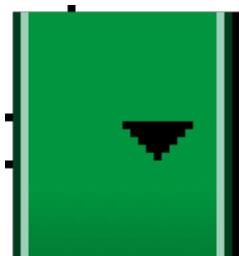


Figure 5. button.9.png

Notice the black pixels along the borders. The ones on the top and left borders indicate the places where the image can be stretched, and the ones on the right and bottom borders indicate where the content should be placed.

Also, notice the `.9.png` extension. You must use this extension, since this is how the framework detects that this is a nine-patch image, as opposed to a regular PNG image.

When you apply this background to a component (by setting `android:background="@drawable/button"`), the framework stretches the image correctly to accommodate the size of the button, as shown in various sizes in figure 6.

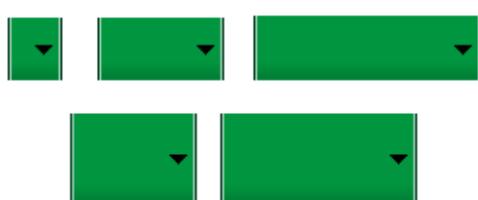


Figure 6. A button using the button.9.png nine-patch in various sizes.

# Supporting Different Densities

This lesson shows you how to support different screen densities by providing different resources and using resolution-independent units of measurements.

## Use Density-independent Pixels

One common pitfall you must avoid when designing your layouts is using absolute pixels to define distances or sizes. Defining layout dimensions with pixels is a problem because different screens have different pixel densities, so the same number of pixels may correspond to different physical sizes on different devices.

Therefore, when specifying dimensions, always use either dp or sp units. A dp is a density-independent pixel that corresponds to the physical size of a pixel at 160 dpi. An sp is the same base unit, but is scaled by the user's preferred text size (it's a scale-independent pixel), so you should use this measurement unit when defining text size (but never for layout sizes).

For example, when you specify spacing between two views, use dp rather than px:

```
<Button android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="@string/clickme"  
        android:layout_marginTop="20dp" />
```

When specifying text size, always use sp:

```
<TextView android:layout_width="match_parent"  
         android:layout_height="wrap_content"  
         android:textSize="20sp" />
```

## Provide Alternative Bitmaps

Since Android runs in devices with a wide variety of screen densities, you should always provide your bitmap resources tailored to each of the generalized density buckets: low, medium, high and extra-high density. This will help you achieve good graphical quality and performance on all screen densities.

To generate these images, you should start with your raw resource in vector format and generate

### THIS LESSON TEACHES YOU TO

1. [Use Density-independent Pixels](#)
2. [Provide Alternative Bitmaps](#)

### YOU SHOULD ALSO READ

- [Supporting Multiple Screens](#)
- [Icon Design Guidelines](#)

### TRY IT OUT

[Download the sample app](#)

NewsReader.zip

the images for each density using the following size scale:

- xhdpi: 2.0
- hdpi: 1.5
- mdpi: 1.0 (baseline)
- ldpi: 0.75

This means that if you generate a 200x200 image for xhdpi devices, you should generate the same resource in 150x150 for hdpi, 100x100 for mdpi and finally a 75x75 image for ldpi devices.

Then, place the generated image files in the appropriate subdirectory under res/ and the system will pick the correct one automatically based on the screen density of the device your application is running on:

```
MyProject/  
  res/  
    drawable-xhdpi/  
      awesomeimage.png  
    drawable-hdpi/  
      awesomeimage.png  
    drawable-mdpi/  
      awesomeimage.png  
    drawable-ldpi/  
      awesomeimage.png
```

Then, any time you reference @drawable/awesomeimage, the system selects the appropriate bitmap based on the screen's dpi.

For more tips and guidelines for creating icon assets for your application, see the [Icon Design Guidelines](#) ([/guide/practices/ui\\_guidelines/icon\\_design.html](#)).

# Implementing Adaptive UI Flows

Depending on the layout that your application is currently showing, the UI flow may be different. For example, if your application is in the dual-pane mode, clicking on an item on the left pane will simply display the content on the right pane; if it is in single-pane mode, the content should be displayed on its own (in a different activity).

## Determine the Current Layout

Since your implementation of each layout will be a little different, one of the first things you will probably have to do is determine what layout the user is currently viewing. For example, you might want to know whether the user is in "single pane" mode or "dual pane" mode. You can do that by querying if a given view exists and is visible:

```
public class NewsReaderActivity extends FragmentActivity {
    boolean mIsDualPane;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_layout);

        View articleView = findViewById(R.id.article);
        mIsDualPane = articleView != null &&
                      articleView.getVisibility() == View.VISIBLE;
    }
}
```

Notice that this code queries whether the "article" pane is available or not, which is much more flexible than hard-coding a query for a specific layout.

Another example of how you can adapt to the existence of different components is to check whether they are available before performing an operation on them. For example, in the News Reader sample app, there is a button that opens a menu, but that button only exists when running on versions older than Android 3.0 (because its function is taken over by the [ActionBar](#) ([/reference/android/app/ActionBar.html](#)) on API level 11+). So, to add the event listener for this button, you can do:

```
Button catButton = (Button) findViewById(R.id.categorybutton);
```

### THIS LESSON TEACHES YOU TO

1. [Determine the Current Layout](#)
2. [React According to Current Layout](#)
3. [Reuse Fragments in Other Activities](#)
4. [Handle Screen Configuration Changes](#)

### YOU SHOULD ALSO READ

- [Supporting Tablets and Handsets](#)

### TRY IT OUT

[Download the sample app](#)

NewsReader.zip

```
OnClickListener listener = /* create your listener here */;
if (catButton != null) {
    catButton.setOnClickListener(listener);
}
```

## React According to Current Layout

Some actions may have a different result depending on the current layout. For example, in the News Reader sample, clicking on a headline from the headlines list opens the article in the right hand-side pane if the UI is in dual-pane mode, but will launch a separate activity if the UI is in single-pane mode:

```
@Override
public void onHeadlineSelected(int index) {
    mArtIndex = index;
    if (mIsDualPane) {
        /* display article on the right pane */
        mArticleFragment.displayArticle(mCurrentCat.getArticle(index));
    } else {
        /* start a separate activity */
        Intent intent = new Intent(this, ArticleActivity.class);
        intent.putExtra("catIndex", mCatIndex);
        intent.putExtra("artIndex", index);
        startActivity(intent);
    }
}
```

Likewise, if the app is in dual-pane mode, it should set up the action bar with tabs for navigation, whereas if the app is in single-pane mode, it should set up navigation with a spinner widget. So your code should also check which case is appropriate:

```
final String CATEGORIES[] = { "Top Stories", "Politics", "Economy", "Technology"

public void onCreate(Bundle savedInstanceState) {
    ...
    if (mIsDualPane) {
        /* use tabs for navigation */
        actionBar.setNavigationMode(android.app.ActionBar.NAVIGATION_MODE_TABS);
        int i;
        for (i = 0; i < CATEGORIES.length; i++) {
            actionBar.addTab(actionBar.newTab().setText(
                CATEGORIES[i]).setTabListener(handler));
        }
        actionBar.setSelectedNavigationItem(selTab);
    }
}
```

```
else {
    /* use list navigation (spinner) */
    actionBar.setNavigationMode(android.app.ActionBar.NAVIGATION_MODE_LIST);
    SpinnerAdapter adap = new ArrayAdapter(this,
        R.layout.headline_item, CATEGORIES);
    actionBar.setListNavigationCallbacks(adap, handler);
}
}
```

## Reuse Fragments in Other Activities

A recurring pattern in designing for multiple screens is having a portion of your interface that's implemented as a pane on some screen configurations and as a separate activity on other configurations. For example, in the News Reader sample, the news article text is presented in the right side pane on large screens, but is a separate activity on smaller screens.

In cases like this, you can usually avoid code duplication by reusing the same [Fragment](#) ([/reference/android/app/Fragment.html](#)) subclass in several activities. For example, ArticleFragment is used in the dual-pane layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.HeadlinesFragment"
        android:layout_width="400dp"
        android:layout_marginRight="10dp"/>
    <fragment android:id="@+id/article"
        android:layout_height="fill_parent"
        android:name="com.example.android.newsreader.ArticleFragment"
        android:layout_width="fill_parent" />
</LinearLayout>
```

And reused (without a layout) in the activity layout for smaller screens (ArticleActivity):

```
ArticleFragment frag = new ArticleFragment();
getSupportFragmentManager().beginTransaction().add(android.R.id.content, frag).co
```

Naturally, this has the same effect as declaring the fragment in an XML layout, but in this case an XML layout is unnecessary work because the article fragment is the only component of this activity.

One very important point to keep in mind when designing your fragments is to not create a strong coupling to a specific activity. You can usually do that by defining an interface that abstracts all the ways in which the fragment needs to interact with its host activity, and then the host activity

implements that interface:

For example, the News Reader app's HeadlinesFragment does precisely that:

```
public class HeadlinesFragment extends ListFragment {  
    ...  
    OnHeadlineSelectedListener mHeadlineSelectedListener = null;  
  
    /* Must be implemented by host activity */  
    public interface OnHeadlineSelectedListener {  
        public void onHeadlineSelected(int index);  
    }  
    ...  
  
    public void setOnHeadlineSelectedListener(OnHeadlineSelectedListener listener)  
        mHeadlineSelectedListener = listener;  
    }  
}
```

Then, when the user selects a headline, the fragment notifies the listener specified by the host activity (as opposed to notifying a specific hard-coded activity):

```
public class HeadlinesFragment extends ListFragment {  
    ...  
    @Override  
    public void onItemClick(AdapterView<?> parent,  
                           View view, int position, long id) {  
        if (null != mHeadlineSelectedListener) {  
            mHeadlineSelectedListener.onHeadlineSelected(position);  
        }  
    }  
    ...  
}
```

This technique is discussed further in the guide to [Supporting Tablets and Handsets \(/guide/practices/tabs-and-handsets.html\)](#).

## Handle Screen Configuration Changes

---

If you are using separate activities to implement separate parts of your interface, you have to keep in mind that it may be necessary to react to certain configuration changes (such as a rotation change) in order to keep your interface consistent.

For example, on a typical 7" tablet running Android 3.0 or higher, the News Reader sample uses a separate activity to display the news article when running in portrait mode, but uses a two-pane layout when in landscape mode.

This means that when the user is in portrait mode and the activity for viewing an article is onscreen, you need to detect that the orientation changed to landscape and react appropriately by ending the activity and return to the main activity so the content can display in the two-pane layout:

```
public class ArticleActivity extends FragmentActivity {  
    int mCatIndex, mArtIndex;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        mCatIndex = getIntent().getExtras().getInt("catIndex", 0);  
        mArtIndex = getIntent().getExtras().getInt("artIndex", 0);  
  
        // If should be in two-pane mode, finish to return to main activity  
        if (getResources().getBoolean(R.bool.has_two_panes)) {  
            finish();  
            return;  
        }  
        ...  
    }  
}
```

# 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. The Android SDK includes tools to help you identify problems in your layout performance, which when combined the lessons here, you will be able to implement smooth scrolling interfaces with a minimum memory footprint.

## DEPENDENCIES AND PREREQUISITES

- Android 1.5 (API Level 3) or higher

## YOU SHOULD ALSO READ

- [XML Layouts](#)

## Lessons

---

### Optimizing Layout Hierarchies

In the same way a complex web page can slow down load time, your layout hierarchy if too complex can also cause performance problems. This lesson shows how you can use SDK tools to inspect your layout and discover performance bottlenecks.

### Re-using Layouts with <include/>

If your application UI repeats certain layout constructs in multiple places, this lesson shows you how to create efficient, re-usable layout constructs, then include them in the appropriate UI layouts.

### Loading Views On Demand

Beyond simply including one layout component within another layout, you might want to make the included layout visible only when it's needed, sometime after the activity is running. This lesson shows how you can improve your layout's initialization performance by loading portions of your layout on demand.

### Making ListView Scrolling Smooth

If you've built an instance of [ListView](#) that contains complex or data-heavy content in each list item, the scroll performance of the list might suffer. This lesson provides some tips about how you can make your scrolling performance more smooth.

# Optimizing Layout Hierarchies

It is a common misconception that using the basic layout structures leads to the most efficient layouts. However, each widget and layout you add to your application requires initialization, layout, and drawing. For example, using nested instances of [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)) can lead to an excessively deep view hierarchy. Furthermore, nesting several instances of [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)) that use the `layout_weight` parameter can be especially expensive as each child needs to be measured twice. This is particularly important when the layout is inflated repeatedly, such as when used in a [ListView](#) ([/reference/android/widget/ListView.html](#)) or [GridView](#) ([/reference/android/widget/GridView.html](#)).

In this lesson you'll learn to use [Hierarchy Viewer](#) ([/tools/help/hierarchy-viewer.html](#)) and [Layoutopt](#) ([/tools/help/layoutopt.html](#)) to examine and optimize your layout.

## Inspect Your Layout

The Android SDK tools include a tool called [Hierarchy Viewer](#) ([/tools/help/hierarchy-viewer.html](#)) that allows you to analyze your layout while your application is running. Using this tool helps you discover bottlenecks in the layout performance.

Hierarchy Viewer works by allowing you to select running processes on a connected device or emulator, then display the layout tree. The traffic lights on each block represent its Measure, Layout and Draw performance, helping you identify potential issues.

For example, figure 1 shows a layout that's used as an item in a [ListView](#) ([/reference/android/widget/ListView.html](#)). This layout shows a small bitmap image on the left and two stacked items of text on the right. It is especially important that layouts that will be inflated multiple times—such as this one—are optimized as the performance benefits will be multiplied.



Figure 1. Conceptual layout for an item in a [ListView](#) ([/reference/android/widget/ListView.html](#)).

The hierarchyviewer tool is available in `<sdk>/tools/`. When opened, the Hierarchy Viewer shows a list of available devices and its running components. Click Load View Hierarchy to view the layout hierarchy of the selected component. For example, figure 2 shows the layout for the list item illustrated by figure 1.

### THIS LESSON TEACHES YOU TO

1. [Inspect Your Layout](#)
2. [Revise Your Layout](#)
3. [Use Lint](#)

### YOU SHOULD ALSO READ

- [XML Layouts](#)
- [Layout Resource](#)



Figure 2. Layout hierarchy for the layout in figure 1, using nested instances of [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)).

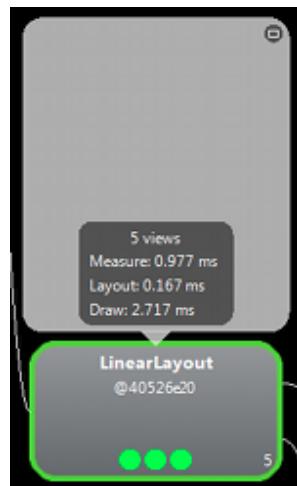


Figure 3. Clicking a hierarchy node shows its performance times.

In figure 2, you can see there is a 3-level hierarchy with some problems laying out the text items. Clicking on the items shows the time taken for each stage of the process (figure 3). It becomes clear which items are taking the longest to measure, layout, and render, and where you should spend time optimizing.

The timings for rendering a complete list item using this layout are:

- Measure: 0.977ms
- Layout: 0.167ms
- Draw: 2.717ms

## Revise Your Layout

Because the layout performance above slows down due to a nested [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)), the performance might improve by flattening the layout—make the layout shallow and wide, rather than narrow and deep. A [RelativeLayout](#) ([/reference/android/widget/RelativeLayout.html](#)) as the root node allows for such layouts. So, when this design is converted to use [RelativeLayout](#) ([/reference/android/widget/RelativeLayout.html](#)), you can see that the layout becomes a 2-level hierarchy. Inspection of the new layout looks like this:

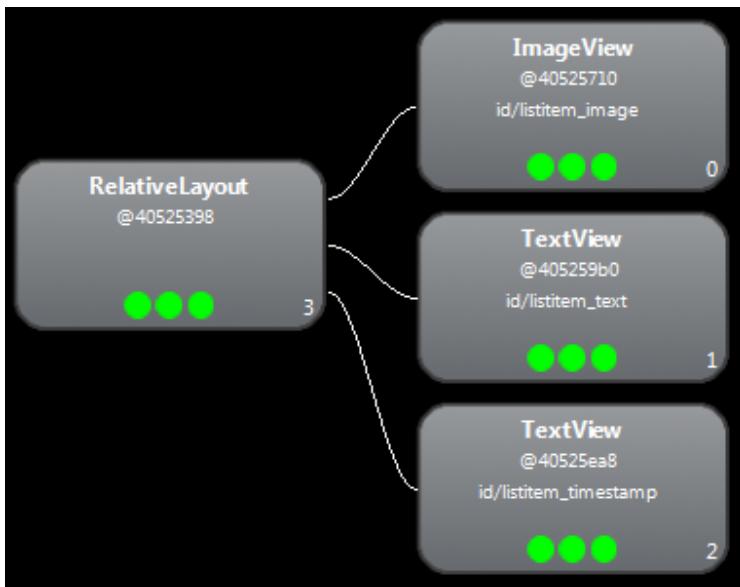


Figure 4. Layout hierarchy for the layout in figure 1, using [RelativeLayout](#) ([/reference/android/widget/RelativeLayout.html](#)).

Now rendering a list item takes:

- Measure: 0.598ms
- Layout: 0.110ms
- Draw: 2.146ms

Might seem like a small improvement, but this time is multiplied several times because this layout is used for every item in a list.

Most of this time difference is due to the use of `layout_weight` in the [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)) design, which can slow down the speed of measurement. It is just one example of how each layout has appropriate uses and you should carefully consider whether using `layout_weight` is necessary.

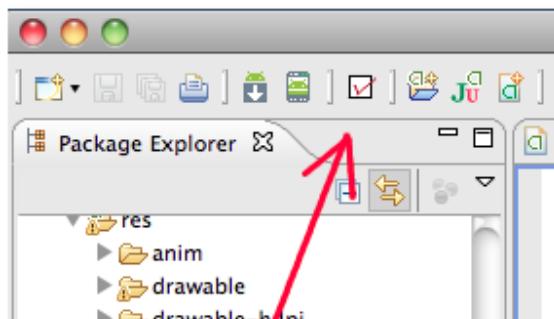
## Use Lint

It is always good practice to run the [Lint](#) (<http://tools.android.com/tips/lint>) tool on your layout files to search for possible view hierarchy optimizations. Lint has replaced the Layoutopt tool and has much greater functionality. Some examples of Lint [rules](#) (<http://tools.android.com/tips/lint-checks>) are:

- Use compound drawables - A [LinearLayout](#) which contains an [ImageView](#) and a [TextView](#) can be more efficiently handled as a compound drawable.
- Merge root frame - If a [FrameLayout](#) is the root of a layout and does not provide background or padding etc, it can be replaced with a merge tag which is slightly more efficient.
- Useless leaf - A layout that has no children or no background can often be removed (since it is invisible) for a flatter and more efficient layout hierarchy.
- Useless parent - A layout with children that has no siblings, is not a [ScrollView](#) or a root layout, and does not have a background, can be removed and have its children moved directly into the parent for a flatter and more efficient layout hierarchy.
- Deep layouts - Layouts with too much nesting are bad for performance. Consider using flatter

layouts such as [RelativeLayout](#) or [GridLayout](#) to improve performance. The default maximum depth is 10.

Another benefit of Lint is that it is integrated into the Android Development Tools for Eclipse (ADT 16+). Lint automatically runs whenever you export an APK, edit and save an XML file or use the Layout Editor. To manually force Lint to run press the Lint button in the Eclipse toolbar.



When used inside Eclipse, Lint has the ability to automatically fix some issues, provide suggestions for others and jump directly to the offending code for review. If you don't use Eclipse for your development, Lint can also be run from the command line. More information about Lint is available at [tools.android.com \(http://tools.android.com/tips/lint\)](http://tools.android.com/tips/lint).

# Re-using Layouts with <include/>

Although Android offers a variety of widgets to provide small and re-usable interactive elements, you might also need to re-use larger components that require a special layout. To efficiently re-use complete layouts, you can use the `<include/>` and `<merge/>` tags to embed another layout inside the current layout.

Reusing layouts is particularly powerful as it allows you create reusable complex layouts. For example, a yes/no button panel, or custom progress bar with description text. It also means that any elements of your application that are common across multiple layouts can be extracted, managed separately, then included in each layout. So while you can create individual UI components by writing a custom [View](#) (`/reference/android/view/View.html`), you can do it even more easily by re-using a layout file.

## Create a Re-usable Layout

If you already know the layout that you want to re-use, create a new XML file and define the layout. For example, here's a layout from the G-Kenya codelab that defines a title bar to be included in each activity (`titlebar.xml`):

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:background="@color/titlebar_bg">  
  
    <ImageView android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:src="@drawable/gafricalogo" />  
/</FrameLayout>
```

The root [View](#) (`/reference/android/view/View.html`) should be exactly how you'd like it to appear in each layout to which you add this layout.

## Use the <include> Tag

Inside the layout to which you want to add the re-usable component, add the `<include/>` tag. For example, here's a layout from the G-Kenya codelab that includes the title bar from above:

Here's the layout file:

### THIS LESSON TEACHES YOU TO

1. [Create a Re-usable Layout](#)
2. [Use the <include> Tag](#)
3. [Use the <merge> Tag](#)

### YOU SHOULD ALSO READ

- [Layout Resources](#)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:background="@color/app_bg"  
    android:gravity="center_horizontal">  
  
    <include layout="@layout/titlebar"/>  
  
    <TextView android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:text="@string/hello"  
        android:padding="10dp" />  
  
    ...  
  
</LinearLayout>
```

You can also override all the layout parameters (any android:layout\_\* attributes) of the included layout's root view by specifying them in the <include/> tag. For example:

```
<include android:id="@+id/news_title"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        layout="@layout/title"/>
```

## Use the <merge> Tag

The <merge /> tag helps eliminate redundant view groups in your view hierarchy when including one layout within another. For example, if your main layout is a vertical [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)) in which two consecutive views can be re-used in multiple layouts, then the re-usable layout in which you place the two views requires its own root view. However, using another [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)) as the root for the re-usable layout would result in a vertical [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)) inside a vertical [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)). The nested [LinearLayout](#) ([/reference/android/widget/LinearLayout.html](#)) serves no real purpose other than to slow down your UI performance.

To avoid including such a redundant view group, you can instead use the <merge> element as the root view for the re-usable layout. For example:

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

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

<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/delete"/>

</merge>
```

Now, when you include this layout in another layout (using the `<include/>` tag), the system ignores the `<merge>` element and places the two buttons directly in the layout, in place of the `<include/>` tag.

# Loading Views On Demand

Sometimes your layout might require complex views that are rarely used. Whether they are item details, progress indicators, or undo messages, you can reduce memory usage and speed up rendering by loading the views only when they are needed.

## Define a ViewStub

[ViewStub](#) (/reference/android/view/ViewStub.html) is a lightweight view with no dimension and doesn't draw anything or participate in the layout. As such, it's cheap to inflate and cheap to leave in a view hierarchy. Each [ViewStub](#) (/reference/android/view/ViewStub.html) simply needs to include the android:layout attribute to specify the layout to inflate.

The following [ViewStub](#) (/reference/android/view/ViewStub.html) is for a translucent progress bar overlay. It should be visible only when new items are being imported into the application.

```
<ViewStub  
    android:id="@+id/stub_import"  
    android:inflatedId="@+id/panel_import"  
    android:layout="@layout/progress_overlay"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:layout_gravity="bottom" />
```

## Load the ViewStub Layout

When you want to load the layout specified by the [ViewStub](#) (/reference/android/view/ViewStub.html), either set it visible by calling [setVisibility\(View.VISIBLE\)](#) (/reference/android/view/View.html#setVisibility(int)) or call [inflate\(\)](#) (/reference/android/view/ViewStub.html#inflate()).

```
((ViewStub) findViewById(R.id.stub_import)).setVisibility(View.VISIBLE);  
// or  
View importPanel = ((ViewStub) findViewById(R.id.stub_import)).inflate();
```

Note: The [inflate\(\)](#) (/reference/android/view/ViewStub.html#inflate()) method returns the inflated [View](#) (/reference/android/view/View.html) once complete. so you don't need to call [findViewById\(\)](#) (/reference/android/app/Activity.html#findViewById(int)) if you need to interact with the layout.

### THIS LESSON TEACHES YOU TO

1. [Define a ViewStub](#)
2. [Load the ViewStub Layout](#)

### YOU SHOULD ALSO READ

- [Optimize with stubs \(blog post\)](#)

Once visible/inflated, the [ViewStub](#) element is no longer part of the view hierarchy. It is replaced by the inflated layout and the ID for the root view of that layout is the one specified by the android:inflatedId attribute of the ViewStub. (The ID android:id specified for the [ViewStub](#) is valid only until the [ViewStub](#) layout is visible/inflated.)

Note: One drawback of [ViewStub](#) is that it doesn't currently support the <merge> tag in the layouts to be inflated.

# Making ListView Scrolling Smooth

The key to a smoothly scrolling [ListView](#) ([/reference/android/widget/ListView.html](#)) is to keep the application's main thread (the UI thread) free from heavy processing. Ensure you do any disk access, network access, or SQL access in a separate thread. To test the status of your app, you can enable [StrictMode](#) ([/reference/android/os/StrictMode.html](#)).

## Use a Background Thread

Using a background thread ("worker thread") removes strain from the main thread so it can focus on drawing the UI. In many cases, using [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) provides a simple way to perform your work outside the main thread. [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) automatically queues up all the `execute()` ([/reference/android/os/AsyncTask.html#execute\(Params...\)](#)) requests and performs them serially. This behavior is global to a particular process and means you don't need to worry about creating your own thread pool.

In the sample code below, an [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) is used to load images in a background thread, then apply them to the UI once finished. It also shows a progress spinner in place of the images while they are loading.

```
// Using an AsyncTask to load the slow images in a background thread
new AsyncTask<ViewHolder, Void, Bitmap>() {
    private ViewHolder v;

    @Override
    protected Bitmap doInBackground(ViewHolder... params) {
        v = params[0];
        return mFakeImageLoader.getImage();
    }

    @Override
    protected void onPostExecute(Bitmap result) {
        super.onPostExecute(result);
        if (v.position == position) {
            // If this item hasn't been recycled already, hide the
            // progress and set and show the image
            v.progress.setVisibility(View.GONE);
            v.icon.setVisibility(View.VISIBLE);
            v.icon.setImageBitmap(result);
        }
    }
}
```

### THIS LESSON TEACHES YOU TO

1. [Use a Background Thread](#)
2. [Hold View Objects in a View Holder](#)

### YOU SHOULD ALSO READ

- [Why is my list black? An Android optimization](#)

```
}.execute(holder);
```

Beginning with Android 3.0 (API level 11), an extra feature is available in [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) so you can enable it to run across multiple processor cores. Instead of calling [execute\(\)](#) ([/reference/android/os/AsyncTask.html#execute\(Params...\)](#)) you can specify [executeOnExecutor\(\)](#) ([/reference/android/os/AsyncTask.html#executeOnExecutor\(java.util.concurrent.Executor, Params...\)](#)) and multiple requests can be executed at the same time depending on the number of cores available.

## Hold View Objects in a View Holder

Your code might call [findViewById\(\)](#) ([/reference/android/app/Activity.html#findViewById\(int\)](#)) frequently during the scrolling of [ListView](#) ([/reference/android/widget/ListView.html](#)), which can slow down performance. Even when the [Adapter](#) ([/reference/android/widget/Adapter.html](#)) returns an inflated view for recycling, you still need to look up the elements and update them. A way around repeated use of [findViewById\(\)](#) ([/reference/android/app/Activity.html#findViewById\(int\)](#)) is to use the "view holder" design pattern.

A ViewHolder object stores each of the component views inside the tag field of the Layout, so you can immediately access them without the need to look them up repeatedly. First, you need to create a class to hold your exact set of views. For example:

```
static class ViewHolder {  
    TextView text;  
    TextView timestamp;  
    ImageView icon;  
    ProgressBar progress;  
    int position;  
}
```

Then populate the ViewHolder and store it inside the layout.

```
ViewHolder holder = new ViewHolder();  
holder.icon = (ImageView) convertView.findViewById(R.id.listitem_image);  
holder.text = (TextView) convertView.findViewById(R.id.listitem_text);  
holder.timestamp = (TextView) convertView.findViewById(R.id.listitem_timestamp);  
holder.progress = (ProgressBar) convertView.findViewById(R.id.progress_spinner);  
convertView.setTag(holder);
```

Now you can easily access each view without the need for the look-up, saving valuable processor cycles.

# Managing Audio Playback

If your app plays audio, it's important that your users can control the audio in a predictable manner. To ensure a great user experience, it's also important that your app manages the audio focus to ensure multiple apps aren't playing audio at the same time.

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.

## DEPENDENCIES AND PREREQUISITES

- Android 2.0 (API level 5) or higher
- Experience with [Media Playback](#)

## YOU SHOULD ALSO READ

- [Services](#)

## Lessons

---

### **Controlling Your App's Volume and Playback**

Learn how to ensure your users can control the volume of your app using the hardware or software volume controls and where available the play, stop, pause, skip, and previous media playback keys.

### **Managing Audio Focus**

With multiple apps potentially playing audio it's important to think about how they should interact. To avoid every music app playing at the same time, Android uses audio focus to moderate audio playback. Learn how to request the audio focus, listen for a loss of audio focus, and how to respond when that happens.

### **Dealing with Audio Output Hardware**

Audio can be played from a number of sources. Learn how to find out where the audio is being played and how to handle a headset being disconnected during playback.

# Controlling Your App's Volume and Playback

A good user experience is a predictable one. If your app plays media it's important that your users can control the volume of your app using the hardware or software volume controls of their device, bluetooth headset, or headphones.

Similarly, where appropriate and available, the play, stop, pause, skip, and previous media playback keys should perform their respective actions on the audio stream used by your app.

## Identify Which Audio Stream to Use

The first step to creating a predictable audio experience is understanding which audio stream your app will use.

Android maintains a separate audio stream for playing music, alarms, notifications, the incoming call ringer, system sounds, in-call volume, and DTMF tones. This is done primarily to allow users to control the volume of each stream independently.

Most of these streams are restricted to system events, so unless your app is a replacement alarm clock, you'll almost certainly be playing your audio using the [STREAM\\_MUSIC](#) ([/reference/android/media/ AudioManager.html#STREAM\\_MUSIC](#)) stream.

## Use Hardware Volume Keys to Control Your App's Audio Volume

By default, pressing the volume controls modify the volume of the active audio stream. If your app isn't currently playing anything, hitting the volume keys adjusts the ringer volume.

If you've got a game or music app, then chances are good that when the user hits the volume keys they want to control the volume of the game or music, even if they're currently between songs or there's no music in the current game location.

You may be tempted to try and listen for volume key presses and modify the volume of your audio stream that way. Resist the urge. Android provides the handy [setVolumeControlStream\(\)](#) ([/reference/android/app/Activity.html#setVolumeControlStream\(int\)](#)) method to direct volume key presses to the audio stream you specify.

Having identified the audio stream your application will be using, you should set it as the volume stream target. You should make this call early in your app's lifecycle—because you only need to call it once during the activity lifecycle, you should typically call it within the `onCreate()` method (of the [Activity](#) ([/reference/android/app/Activity.html](#)) or [Fragment](#) ([/reference/android/app/Fragment.html](#)) that controls your media). This ensures that whenever your app is visible, the

### THIS LESSON TEACHES YOU TO

1. [Identify Which Audio Stream to Use](#)
2. [Use Hardware Volume Keys to Control Your App's Audio Volume](#)
3. [Use Hardware Playback Control Keys to Control Your App's Audio Playback](#)

### YOU SHOULD ALSO READ

- [Media Playback](#)

volume controls function as the user expects.

```
setVolumeControlStream(AudioManager.STREAM_MUSIC);
```

From this point onwards, pressing the volume keys on the device affect the audio stream you specify (in this case “music”) whenever the target activity or fragment is visible.

## Use Hardware Playback Control Keys to Control Your App’s Audio Playback

Media playback buttons such as play, pause, stop, skip, and previous are available on some handsets and many connected or wireless headsets. Whenever a user presses one of these hardware keys, the system broadcasts an intent with the [ACTION\\_MEDIA\\_BUTTON](#) (/reference/android/content/Intent.html#ACTION\_MEDIA\_BUTTON) action.

To respond to media button clicks, you need to register a [BroadcastReceiver](#) (/reference/android/content/BroadcastReceiver.html) in your manifest that listens for this action broadcast as shown below.

```
<receiver android:name=".RemoteControlReceiver">
    <intent-filter>
        <action android:name="android.intent.action.MEDIA_BUTTON" />
    </intent-filter>
</receiver>
```

The receiver implementation itself needs to extract which key was pressed to cause the broadcast. The [Intent](#) (/reference/android/content/Intent.html) includes this under the [EXTRA\\_KEY\\_EVENT](#) (/reference/android/content/Intent.html#EXTRA\_KEY\_EVENT) key, while the [KeyEvent](#) (/reference/android/view(KeyEvent.html)) class includes a list [KEYCODE\\_MEDIA\\_\\*](#) static constants that represents each of the possible media buttons, such as [KEYCODE\\_MEDIA\\_PLAY\\_PAUSE](#) (/reference/android/view(KeyEvent.html#KEYCODE\_MEDIA\_PLAY\_PAUSE) and [KEYCODE\\_MEDIA\\_NEXT](#) (/reference/android/view(KeyEvent.html#KEYCODE\_MEDIA\_NEXT)).

The following snippet shows how to extract the media button pressed and affects the media playback accordingly.

```
public class RemoteControlReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (Intent.ACTION_MEDIA_BUTTON.equals(intent.getAction())) {
            KeyEvent event = (KeyEvent) intent.getParcelableExtra(Intent.EXTRA_KEY_EVENT);
            if (KeyEvent.KEYCODE_MEDIA_PLAY == event.getKeyCode()) {
                // Handle key press.
            }
        }
    }
}
```

```
    }
```

Because multiple applications might want to listen for media button presses, you must also programmatically control when your app should receive media button press events.

The following code can be used within your app to register and de-register your media button event receiver using the [AudioManager](#) ([/reference/android/media/ AudioManager.html](#)). When registered, your broadcast receiver is the exclusive receiver of all media button broadcasts.

```
AudioManager am = mContext.getSystemService(Context.AUDIO_SERVICE);  
...  
// Start listening for button presses  
am.registerMediaButtonEventReceiver(RemoteControlReceiver);  
...  
// Stop listening for button presses  
am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
```

Typically, apps should unregister most of their receivers whenever they become inactive or invisible (such as during the [onStop\(\)](#) ([/reference/android/app/Activity.html#onStop\(\)](#)) callback). However, it's not that simple for media playback apps—in fact, responding to media playback buttons is most important when your application isn't visible and therefore can't be controlled by the on-screen UI.

A better approach is to register and unregister the media button event receiver when your application gains and loses the audio focus. This is covered in detail in the next lesson.

# Managing Audio Focus

With multiple apps potentially playing audio it's important to think about how they should interact. To avoid every music app playing at the same time, Android uses audio focus to moderate audio playback—only apps that hold the audio focus should play audio.

Before your app starts playing audio it should request—and receive—the audio focus. Likewise, it should know how to listen for a loss of audio focus and respond appropriately when that happens.

## THIS LESSON TEACHES YOU TO

1. [Request the Audio Focus](#)
2. [Handle the Loss of Audio Focus](#)
3. [Duck!](#)

## YOU SHOULD ALSO READ

- [Media Playback](#)

## Request the Audio Focus

Before your app starts playing any audio, it should hold the audio focus for the stream it will be using. This is done with a call to [requestAudioFocus\(\)](#) ([/reference/android/media/ AudioManager.html#requestAudioFocus\(android.media.AudioManager.OnAudioFocusChangeListener, int, int\)](#)) which returns [AUDIOFOCUS\\_REQUEST\\_GRANTED](#) ([/reference/android/media/ AudioManager.html#AUDIOFOCUS\\_REQUEST\\_GRANTED](#)) if your request is successful.

You must specify which stream you're using and whether you expect to require transient or permanent audio focus. Request transient focus when you expect to play audio for only a short time (for example when playing navigation instructions). Request permanent audio focus when you plan to play audio for the foreseeable future (for example, when playing music).

The following snippet requests permanent audio focus on the music audio stream. You should request the audio focus immediately before you begin playback, such as when the user presses play or the background music for the next game level begins.

```
AudioManager am = mContext.getSystemService(Context.AUDIO_SERVICE);  
...  
// Request audio focus for playback  
int result = am.requestAudioFocus(afChangeListener,  
                                // Use the music stream.  
                                AudioManager.STREAM_MUSIC,  
                                // Request permanent focus.  
                                AudioManager.AUDIOFOCUS_GAIN);  
  
if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {  
    am.unregisterMediaButtonEventReceiver(remoteControlReceiver);  
    // Start playback.  
}  
}
```

Once you've finished playback be sure to call [abandonAudioFocus\(\)](#) ([/reference/android/media/ AudioManager.html#abandonAudioFocus\(android.media.AudioManager.OnAudioFocusChangeListener\)](#)). This notifies the system that you no longer require focus and unregisters the associated [AudioManager.OnAudioFocusChangeListener](#) ([/reference/android/media/ AudioManager.OnAudioFocusChangeListener.html](#)). In the case of abandoning transient focus, this allows any interrupted app to continue playback.

```
// Abandon audio focus when playback complete  
am.abandonAudioFocus(afChangeListener);
```

When requesting transient audio focus you have an additional option: whether or not you want to enable "ducking." Normally, when a well-behaved audio app loses audio focus it immediately silences its playback. By requesting a transient audio focus that allows ducking you tell other audio apps that it's acceptable for them to keep playing, provided they lower their volume until the focus returns to them.

```
// Request audio focus for playback  
int result = am.requestAudioFocus(afChangeListener,  
                                  // Use the music stream.  
                                  AudioManager.STREAM_MUSIC,  
                                  // Request permanent focus.  
                                  AudioManager.AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK);  
  
if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED) {  
    // Start playback.  
}
```

Ducking is particularly suitable for apps that use the audio stream intermittently, such as for audible driving directions.

Whenever another app requests audio focus as described above, its choice between permanent and transient (with or without support for ducking) audio focus is received by the listener you registered when requesting focus.

## Handle the Loss of Audio Focus

If your app can request audio focus, it follows that it will in turn lose that focus when another app requests it. How your app responds to a loss of audio focus depends on the manner of that loss.

The [onAudioFocusChange\(\)](#) ([/reference/android/media/ AudioManager.OnAudioFocusChangeListener.html#onAudioFocusChange\(int\)](#)) callback method of the audio focus change listener you registered when requesting audio focus receives a parameter that describes the focus change event. Specifically, the possible focus loss events mirror the focus request types from the previous section—permanent loss, transient loss, and transient with ducking permitted.

Generally speaking, a transient (temporary) loss of audio focus should result in your app silencing

it's audio stream, but otherwise maintaining the same state. You should continue to monitor changes in audio focus and be prepared to resume playback where it was paused once you've regained the focus.

If the audio focus loss is permanent, it's assumed that another application is now being used to listen to audio and your app should effectively end itself. In practical terms, that means stopping playback, removing media button listeners—allowing the new audio player to exclusively handle those events—and abandoning your audio focus. At that point, you would expect a user action (pressing play in your app) to be required before you resume playing audio.

In the following code snippet, we pause the playback or our media player object if the audio loss is transient and resume it when we have regained the focus. If the loss is permanent, it unregisters our media button event receiver and stops monitoring audio focus changes.

```
OnAudioFocusChangeListener afChangeListener = new OnAudioFocusChangeListener() {
    public void onAudioFocusChange(int focusChange) {
        if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT
            // Pause playback
        } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {
            // Resume playback
        } else if (focusChange == AudioManager.AUDIOFOCUS_LOSS) {
            am.unregisterMediaButtonEventReceiver(RemoteControlReceiver);
            am.abandonAudioFocus(afChangeListener);
            // Stop playback
        }
    }
};
```

In the case of a transient loss of audio focus where ducking is permitted, rather than pausing playback, you can "duck" instead.

## Duck!

Ducking is the process of lowering your audio stream output volume to make transient audio from another app easier to hear without totally disrupting the audio from your own application.

In the following code snippet lowers the volume on our media player object when we temporarily lose focus, then returns it to its previous level when we regain focus.

```
OnAudioFocusChangeListener afChangeListener = new OnAudioFocusChangeListener() {
    public void onAudioFocusChange(int focusChange) {
        if (focusChange == AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK) {
            // Lower the volume
        } else if (focusChange == AudioManager.AUDIOFOCUS_GAIN) {
            // Raise it back to normal
        }
    }
};
```

```
};
```

A loss of audio focus is the most important broadcast to react to, but not the only one. The system broadcasts a number of intents to alert you to changes in user's audio experience. The next lesson demonstrates how to monitor them to improve the user's overall experience.

# Dealing with Audio Output Hardware

Users have a number of alternatives when it comes to enjoying the audio from their Android devices. Most devices have a built-in speaker, headphone jacks for wired headsets, and many also feature Bluetooth connectivity and support for A2DP audio.

## Check What Hardware is Being Used

How your app behaves might be affected by which hardware its output is being routed to.

You can query the [AudioManager](#) (</reference/android/media/ AudioManager.html>) to determine if the audio is currently being routed to the device speaker, wired headset, or attached Bluetooth device as shown in the following snippet:

```
if (isBluetoothA2dpOn()) {  
    // Adjust output for Bluetooth.  
} else if (isSpeakerphoneOn()) {  
    // Adjust output for Speakerphone.  
} else if (isWiredHeadsetOn()) {  
    // Adjust output for headsets  
} else {  
    // If audio plays and noone can hear it, is it still playing?  
}
```

### THIS LESSON TEACHES YOU TO

1. [Check What Hardware is Being Used](#)
2. [Handle Changes in the Audio Output Hardware](#)

### YOU SHOULD ALSO READ

- [Media Playback](#)

## Handle Changes in the Audio Output Hardware

When a headset is unplugged, or a Bluetooth device disconnected, the audio stream automatically reroutes to the built in speaker. If you listen to your music at as high a volume as I do, that can be a noisy surprise.

Luckily the system broadcasts an [ACTION\\_AUDIO\\_BECOMING\\_NOISY](#) ([/reference/android/media/ AudioManager.html#ACTION\\_AUDIO\\_BECOMING\\_NOISY](/reference/android/media/ AudioManager.html#ACTION_AUDIO_BECOMING_NOISY)) intent when this happens. It's good practice to register a [BroadcastReceiver](#) (</reference/android/content/BroadcastReceiver.html>) that listens for this intent whenever you're playing audio. In the case of music players, users typically expect the playback to be paused—while for games you may choose to significantly lower the volume.

```
private class NoisyAudioStreamReceiver extends BroadcastReceiver {  
    @Override  
    public void onReceive(Context context, Intent intent) {
```

```
        if (AudioManager.ACTION_AUDIO_BECOMING_NOISY.equals(intent.getAction()))
            // Pause the playback
    }
}

private IntentFilter intentFilter = new IntentFilter(AudioManager.ACTION_AUDIO_BE

private void startPlayback() {
    registerReceiver(myNoisyAudioStreamReceiver(), intentFilter);
}

private void stopPlayback() {
    unregisterReceiver(myNoisyAudioStreamReceiver());
}
```

# Optimizing Battery Life

For your app to be a good citizen, it should seek to limit its impact on the battery life of its host device. After this class you will be able to build apps that monitor modify their functionality and behavior based on the state of the host device.

By taking steps such as disabling background service updates when you lose connectivity, or reducing the rate of such updates when the battery level is low, you can ensure that the impact of your app on battery life is minimized, without compromising the user experience.

## DEPENDENCIES AND PREREQUISITES

- Android 2.0 (API level 5) or higher
- Experience with [Intents and Intent Filters](#)

## YOU SHOULD ALSO READ

- [Services](#)

## Lessons

---

### **Monitoring the Battery Level and Charging State**

Learn how to alter your app's update rate by determining, and monitoring, the current battery level and changes in charging state.

### **Determining and Monitoring the Docking State and Type**

Optimal refresh rates can vary based on how the host device is being used. Learn how to determine, and monitor, the docking state and type of dock being used to affect your app's behavior.

### **Determining and Monitoring the Connectivity Status**

Without Internet connectivity you can't update your app from an online source. Learn how to check the connectivity status to alter your background update rate. You'll also learn to check for Wi-Fi or mobile connectivity before beginning high-bandwidth operations.

### **Manipulating Broadcast Receivers On Demand**

Broadcast receivers that you've declared in the manifest can be toggled at runtime to disable those that aren't necessary due to the current device state. Learn to improve efficiency by toggling and cascading state change receivers and delay actions until the device is in a specific state.

# Monitoring the Battery Level and Charging State

When you're altering the frequency of your background updates to reduce the effect of those updates on battery life, checking the current battery level and charging state is a good place to start.

The battery-life impact of performing application updates depends on the battery level and charging state of the device. The impact of performing updates while the device is charging over AC is negligible, so in most cases you can maximize your refresh rate whenever the device is connected to a wall charger. Conversely, if the device is discharging, reducing your update rate helps prolong the battery life.

Similarly, you can check the battery charge level, potentially reducing the frequency of—or even stopping—your updates when the battery charge is nearly exhausted.

## THIS LESSON TEACHES YOU TO

1. [Determine the Current Charging State](#)
2. [Monitor Changes in Charging State](#)
3. [Determine the Current Battery Level](#)
4. [Monitor Significant Changes in Battery Level](#)

## YOU SHOULD ALSO READ

- [Intents and Intent Filters](#)

## Determine the Current Charging State

Start by determining the current charge status. The [BatteryManager \(/reference/android/os/BatteryManager.html\)](#) broadcasts all battery and charging details in a sticky [Intent \(/reference/android/content/Intent.html\)](#) that includes the charging status.

Because it's a sticky intent, you don't need to register a [BroadcastReceiver \(/reference/android/content/BroadcastReceiver.html\)](#)—by simply calling `registerReceiver` passing in `null` as the receiver as shown in the next snippet, the current battery status intent is returned. You could pass in an actual [BroadcastReceiver \(/reference/android/content/BroadcastReceiver.html\)](#) object here, but we'll be handling updates in a later section so it's not necessary.

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
```

You can extract both the current charging status and, if the device is being charged, whether it's charging via USB or AC charger:

```
// Are we charging / charged?
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                     status == BatteryManager.BATTERY_STATUS_FULL;

// How are we charging?
```

```
int chargePlug = battery.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
boolean usbCharge = chargePlug == BATTERY_PLUGGED_USB;
boolean acCharge = chargePlug == BATTERY_PLUGGED_AC;
```

Typically you should maximize the rate of your background updates in the case where the device is connected to an AC charger, reduce the rate if the charge is over USB, and lower it further if the battery is discharging.

## Monitor Changes in Charging State

The charging status can change as easily as a device can be plugged in, so it's important to monitor the charging state for changes and alter your refresh rate accordingly.

The [BatteryManager](#) ([/reference/android/os/BatteryManager.html](#)) broadcasts an action whenever the device is connected or disconnected from power. It's important to receive these events even while your app isn't running—particularly as these events should impact how often you start your app in order to initiate a background update—so you should register a [BroadcastReceiver](#) ([/reference/android/content/BroadcastReceiver.html](#)) in your manifest to listen for both events by defining the [ACTION\\_POWER\\_CONNECTED](#) ([/reference/android/content/Intent.html#ACTION\\_POWER\\_CONNECTED](#)) and [ACTION\\_POWER\\_DISCONNECTED](#) ([/reference/android/content/Intent.html#ACTION\\_POWER\\_DISCONNECTED](#)) within an intent filter.

```
<receiver android:name=".PowerConnectionReceiver">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
        <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
    </intent-filter>
</receiver>
```

Within the associated [BroadcastReceiver](#) ([/reference/android/content/BroadcastReceiver.html](#)) implementation, you can extract the current charging state and method as described in the previous step.

```
public class PowerConnectionReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        int status = intent.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
        boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                            status == BatteryManager.BATTERY_STATUS_FULL;

        int chargePlug = intent.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
        boolean usbCharge = chargePlug == BATTERY_PLUGGED_USB;
        boolean acCharge = chargePlug == BATTERY_PLUGGED_AC;
    }
}
```

## Determine the Current Battery Level

In some cases it's also useful to determine the current battery level. You may choose to reduce the rate of your background updates if the battery charge is below a certain level.

You can find the current battery charge by extracting the current battery level and scale from the battery status intent as shown here:

```
int level = battery.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
int scale = battery.getIntExtra(BatteryManager.EXTRA_SCALE, -1);

float batteryPct = level / (float)scale;
```

## Monitor Significant Changes in Battery Level

You can't easily continually monitor the battery state, but you don't need to.

Generally speaking, the impact of constantly monitoring the battery level has a greater impact on the battery than your app's normal behavior, so it's good practice to only monitor significant changes in battery level—specifically when the device enters or exits a low battery state.

The manifest snippet below is extracted from the intent filter element within a broadcast receiver. The receiver is triggered whenever the device battery becomes low or exits the low condition by listening for [ACTION\\_BATTERY\\_LOW](#) (/reference/android/content/Intent.html#ACTION\_BATTERY\_LOW) and [ACTION\\_BATTERY\\_OKAY](#) (/reference/android/content/Intent.html#ACTION\_BATTERY\_OKAY).

```
<receiver android:name=".BatteryLevelReceiver">
<intent-filter>
    <action android:name="android.intent.action.ACTION_BATTERY_LOW"/>
    <action android:name="android.intent.action.ACTION_BATTERY_OKAY"/>
</intent-filter>
</receiver>
```

It is generally good practice to disable all your background updates when the battery is critically low. It doesn't matter how fresh your data is if the phone turns itself off before you can make use of it.

In many cases, the act of charging a device is coincident with putting it into a dock. The next lesson shows you how to determine the current dock state and monitor for changes in device docking.

# Determining and Monitoring the Docking State and Type

Android devices can be docked into several different kinds of docks. These include car or home docks and digital versus analog docks. The dock-state is typically closely linked to the charging state as many docks provide power to docked devices.

How the dock-state of the phone affects your update rate depends on your app. You may choose to increase the update frequency of a sports center app when it's in the desktop dock, or disable your updates completely if the device is car docked. Conversely, you may choose to maximize your updates while car docked if your background service is updating traffic conditions.

The dock state is also broadcast as a sticky [Intent](#) ([/reference/android/content/Intent.html](#)), allowing you to query if the device is docked or not, and if so, in which kind of dock.

## Determine the Current Docking State

The dock-state details are included as an extra in a sticky broadcast of the [ACTION\\_DOCK\\_EVENT](#) ([/reference/android/content/Intent.html#ACTION\\_DOCK\\_EVENT](#)) action. Because it's sticky, you don't need to register a [BroadcastReceiver](#) ([/reference/android/content/BroadcastReceiver.html](#)). You can simply call [registerReceiver\(\)](#) ([/reference/android/content/Context.html#registerReceiver\(android.content.BroadcastReceiver, android.content.IntentFilter\)](#)) passing in null as the broadcast receiver as shown in the next snippet.

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_DOCK_EVENT);
Intent dockStatus = context.registerReceiver(null, ifilter);
```

You can extract the current docking status from the EXTRA\_DOCK\_STATE extra:

```
int dockState = battery.getIntExtra(EXTRA_DOCK_STATE, -1);
boolean isDocked = dockState != Intent.EXTRA_DOCK_STATE_UNDOCKED;
```

## Determine the Current Dock Type

If a device is docked, it can be docked in any one of four different type of dock:

- Car
- Desk

### THIS LESSON TEACHES YOU TO

1. [Determine the Current Docking State](#)
2. [Determine the Current Dock Type](#)
3. [Monitor for Changes in the Dock State or Type](#)

### YOU SHOULD ALSO READ

- [Intents and Intent Filters](#)

- Low-End (Analog) Desk
- High-End (Digital) Desk

Note that the latter two options were only introduced to Android in API level 11, so it's good practice to check for all three where you are only interested in the type of dock rather than it being digital or analog specifically:

```
boolean isCar = dockState == EXTRA_DOCK_STATE_CAR;  
boolean isDesk = dockState == EXTRA_DOCK_STATE_DESK ||  
    dockState == EXTRA_DOCK_STATE_LE_DESK ||  
    dockState == EXTRA_DOCK_STATE_HE_DESK;
```

## Monitor for Changes in the Dock State or Type

---

Whenever the device is docked or undocked, the [ACTION\\_DOCK\\_EVENT](#) (/reference/android/content/Intent.html#ACTION\_DOCK\_EVENT) action is broadcast. To monitor changes in the device's dock-state, simply register a broadcast receiver in your application manifest as shown in the snippet below:

```
<action android:name="android.intent.action.ACTION_DOCK_EVENT" />
```

You can extract the dock type and state within the receiver implementation using the same techniques described in the previous step.

# Determining and Monitoring the Connectivity Status

Some of the most common uses for repeating alarms and background services is to schedule regular updates of application data from Internet resources, cache data, or execute long running downloads. But if you aren't connected to the Internet, or the connection is too slow to complete your download, why both waking the device to schedule the update at all?

You can use the [ConnectivityManager](#) ([/reference/android/net/ConnectivityManager.html](#)) to check that you're actually connected to the Internet, and if so, what type of connection is in place.

## THIS LESSON TEACHES YOU TO

1. [Determine if you Have an Internet Connection](#)
2. [Determine the Type of your Internet Connection](#)
3. [Monitor for Changes in Connectivity](#)

## YOU SHOULD ALSO READ

- [Intents and Intent Filters](#)

## Determine if You Have an Internet Connection

There's no need to schedule an update based on an Internet resource if you aren't connected to the Internet. The following snippet shows how to use the [ConnectivityManager](#) ([/reference/android/net/ConnectivityManager.html](#)) to query the active network and determine if it has Internet connectivity.

```
ConnectivityManager cm =
    (ConnectivityManager)context.getSystemService(Context.CONNECTIVITY_SERVICE);

NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
boolean isConnected = activeNetwork.isConnectedOrConnecting();
```

## Determine the Type of your Internet Connection

It's also possible to determine the type of Internet connection currently available.

Device connectivity can be provided by mobile data, WiMAX, Wi-Fi, and ethernet connections. By querying the type of the active network, as shown below, you can alter your refresh rate based on the bandwidth available.

```
boolean isWiFi = activeNetwork.getType() == ConnectivityManager.TYPE_WIFI;
```

Mobile data costs tend to be significantly higher than Wi-Fi, so in most cases, your app's update rate should be lower when on mobile connections. Similarly, downloads of significant size should be suspended until you have a Wi-Fi connection.

Having disabled your updates, it's important that you listen for changes in connectivity in order to resume them once an Internet connection has been established.

## Monitor for Changes in Connectivity

---

The [ConnectivityManager](#) ([/reference/android/net/ConnectivityManager.html](#)) broadcasts the [CONNECTIVITY\\_ACTION](#) ([/reference/android/net/ConnectivityManager.html#CONNECTIVITY\\_ACTION](#)) ("`android.net.conn.CONNECTIVITY_CHANGE`") action whenever the connectivity details have changed. You can register a broadcast receiver in your manifest to listen for these changes and resume (or suspend) your background updates accordingly.

```
<action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
```

Changes to a device's connectivity can be very frequent—this broadcast is triggered every time you move between mobile data and Wi-Fi. As a result, it's good practice to monitor this broadcast only when you've previously suspended updates or downloads in order to resume them. It's generally sufficient to simply check for Internet connectivity before beginning an update and, should there be none, suspend further updates until connectivity is restored.

This technique requires toggling broadcast receivers you've declared in the manifest, which is described in the next lesson.

# Manipulating Broadcast Receivers On Demand

The simplest way to monitor device state changes is to create a [BroadcastReceiver](#) ([/reference/android/content/BroadcastReceiver.html](#)) for each state you're monitoring and register each of them in your application manifest. Then within each of these receivers you simply reschedule your recurring alarms based on the current device state.

A side-effect of this approach is that your app will wake the device each time any of these receivers is triggered —potentially much more frequently than required.

A better approach is to disable or enable the broadcast receivers at runtime. That way you can use the receivers you declared in the manifest as passive alarms that are triggered by system events only when necessary.

## Toggle and Cascade State Change Receivers to Improve Efficiency

You can use the [PackageManager](#) ([/reference/android/content/pm/PackageManager.html](#)) to toggle the enabled state on any component defined in the manifest, including whichever broadcast receivers you wish to enable or disable as shown in the snippet below:

```
ComponentName receiver = new ComponentName(context, myReceiver.class);

PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
    PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
    PackageManager.DONT_KILL_APP)
```

Using this technique, if you determine that connectivity has been lost, you can disable all of your receivers except the connectivity-change receiver. Conversely, once you are connected you can stop listening for connectivity changes and simply check to see if you're online immediately before performing an update and rescheduling a recurring update alarm.

You can use the same technique to delay a download that requires higher bandwidth to complete. Simply enable a broadcast receiver that listens for connectivity changes and initiates the download only after you are connected to Wi-Fi.

### THIS LESSON TEACHES YOU TO

1. [Toggle and Cascade State Change Receivers to Improve Efficiency](#)

### YOU SHOULD ALSO READ

- [Intents and Intent Filters](#)

# Creating Custom Views

Improving Layout Performance

The Android framework has [\(/reference/android/view/View.html\)](#) with the user and displaying sometimes your app has user needs not covered by the built-in views. To create your own views to

Managing Audio Playback

Optimizing Battery Life

[Creating Custom Views](#)

Creating a Custom View Class

Implementing Custom Drawing

Making the View Interactive

Optimizing the View

Adding Search Functionality

Remembering Users

Sharing Content

Capturing Photos

Maintaining Multiple APKs

Creating Backward-Compatible UIs

Developing for Enterprise

Monetizing Your App

## Lessons

### [Creating a View Class](#)

Create a class that adds custom attributes and editor.

### [Custom Drawing](#)

Make your view visually appealing using Android graphics system.

### [Making the View Interactive](#)

Users expect a view to react naturally to input gestures. Learn how to use gesture controls and animation to give your view a natural feel.

### [Optimizing the View](#)

No matter how beautiful your view is, it must run at a consistent frame rate. Learn how to use hardware acceleration to make

## DEPENDENCIES AND PREREQUISITES

- Android 2.1 (API level 7) or higher

## YOU SHOULD ALSO READ

- [Custom Components](#)
- [Input Events](#)
- [Property Animation](#)
- [Hardware Acceleration](#)
- [Accessibility](#) developer guide

## TRY IT OUT

[Download the sample](#)

CustomView.zip

feel.

it if it doesn't run at a consistently high frame rate. Learn how to identify performance problems, and how to use hardware acceleration to make your app faster.

# Creating a View Class

A well-designed custom view is much like any other well-designed class. It encapsulates a specific set of functionality with an easy to use interface, it uses CPU and memory efficiently, and so forth. In addition to being a well-designed class, though, a custom view should:

- Conform to Android standards
- Provide custom styleable attributes that work with Android XML layouts
- Send accessibility events
- Be compatible with multiple Android platforms.

The Android framework provides a set of base classes and XML tags to help you create a view that meets all of these requirements. This lesson discusses how to use the Android framework to create the core functionality of a view class.

## THIS LESSON TEACHES YOU TO

1. [Subclass a View](#)
2. [Define Custom Attributes](#)
3. [Apply Custom Attributes to a View](#)
4. [Add Properties and Events](#)
5. [Design For Accessibility](#)

## YOU SHOULD ALSO READ

- [Custom Components](#)

## TRY IT OUT

[Download the sample](#)

CustomView.zip

## Subclass a View

All of the view classes defined in the Android framework extend [View \(/reference/android/view/View.html\)](#). Your custom view can also extend [View \(/reference/android/view/View.html\)](#) directly, or you can save time by extending one of the existing view subclasses, such as [Button \(/reference/android/widget/Button.html\)](#).

To allow the [Android Developer Tools \(/guide/developing/tools/adt.html\)](#) to interact with your view, at a minimum you must provide a constructor that takes a [Context \(/reference/android/content/Context.html\)](#) and an [AttributeSet \(/reference/android/util/AttributeSet.html\)](#) object as parameters. This constructor allows the layout editor to create and edit an instance of your view.

```
class PieChart extends View {  
    public PieChart(Context ctx, AttributeSet attrs) {  
        super(ctx, attrs);  
    }  
}
```

## Define Custom Attributes

To add a built-in [View \(/reference/android/view/View.html\)](#) to your user interface, you specify it in an XML element and control its appearance and behavior with element attributes. Well-written custom views can also be added and styled via XML. To enable this behavior in your custom view, you

must:

- Define custom attributes for your view in a `<declare-styleable>` resource element
- Specify values for the attributes in your XML layout
- Retrieve attribute values at runtime
- Apply the retrieved attribute values to your view

This section discusses how to define custom attributes and specify their values. The next section deals with retrieving and applying the values at runtime.

To define custom attributes, add `<declare-styleable>` resources to your project. It's customary to put these resources into a `res/values/attrs.xml` file. Here's an example of an `attrs.xml` file:

```
<resources>
    <declare-styleable name="PieChart">
        <attr name="showText" format="boolean" />
        <attr name="labelPosition" format="enum">
            <enum name="left" value="0"/>
            <enum name="right" value="1"/>
        </attr>
    </declare-styleable>
</resources>
```

This code declares two custom attributes, `showText` and `labelPosition`, that belong to a styleable entity named `PieChart`. The name of the styleable entity is, by convention, the same name as the name of the class that defines the custom view. Although it's not strictly necessary to follow this convention, many popular code editors depend on this naming convention to provide statement completion.

Once you define the custom attributes, you can use them in layout XML files just like built-in attributes. The only difference is that your custom attributes belong to a different namespace. Instead of belonging to the `http://schemas.android.com/apk/res/android` namespace, they belong to `http://schemas.android.com/apk/res/[your package name]`. For example, here's how to use the attributes defined for `PieChart`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:custom="http://schemas.android.com/apk/res/com.example.customviews">
    <com.example.customviews.charting.PieChart
        custom:showText="true"
        custom:labelPosition="left" />
</LinearLayout>
```

In order to avoid having to repeat the long namespace URI, the sample uses an `xmlns:custom` directive. This directive assigns the alias `custom` to the namespace `http://schemas.android.com/apk/res/com.example.customviews`. You can choose any alias you want for your namespace.

Notice the name of the XML tag that adds the custom view to the layout. It is the fully qualified name of the custom view class. If your view class is an inner class, you must further qualify it with

the name of the view's outer class. further. For instance, the PieChart class has an inner class called PieView. To use the custom attributes from this class, you would use the tag com.example.customviews.charting.PieChart\$PieView.

## Apply Custom Attributes

When a view is created from an XML layout, all of the attributes in the XML tag are read from the resource bundle and passed into the view's constructor as an [AttributeSet](#) ([/reference/android/util/AttributeSet.html](#)). Although it's possible to read values from the [AttributeSet](#) ([/reference/android/util/AttributeSet.html](#)) directly, doing so has some disadvantages:

- Resource references within attribute values are not resolved
- Styles are not applied

Instead, pass the [AttributeSet](#) ([/reference/android/util/AttributeSet.html](#)) to [obtainStyledAttributes\(\)](#) ([/reference/android/content/res/Resources.Theme.html#obtainStyledAttributes\(android.util.AttributeSet, int\[\], int, int\)](#)). This method passes back a [TypedArray](#) ([/reference/android/content/res/TypedArray.html](#)) array of values that have already been dereferenced and styled.

The Android resource compiler does a lot of work for you to make calling [obtainStyledAttributes\(\)](#) ([/reference/android/content/res/Resources.Theme.html#obtainStyledAttributes\(android.util.AttributeSet, int\[\], int, int\)](#)) easier. For each <declare-styleable> resource in the res directory, the generated R.java defines both an array of attribute ids and a set of constants that define the index for each attribute in the array. You use the predefined constants to read the attributes from the [TypedArray](#) ([/reference/android/content/res/TypedArray.html](#)). Here's how the PieChart class reads its attributes:

```
public PieChart(Context ctx, AttributeSet attrs) {
    super(ctx, attrs);
    TypedArray a = context.getTheme().obtainStyledAttributes(
        attrs,
        R.styleable.PieChart,
        0, 0);

    try {
        mShowText = a.getBoolean(R.styleable.PieChart_showText, false);
        mTextPos = a.getInteger(R.styleable.PieChart_labelPosition, 0);
    } finally {
        a.recycle();
    }
}
```

Note that [TypedArray](#) ([/reference/android/content/res/TypedArray.html](#)) objects are a shared resource and must be recycled after use.

## Add Properties and Events

Attributes are a powerful way of controlling the behavior and appearance of views, but they can only be read when the view is initialized. To provide dynamic behavior, expose a property getter and setter pair for each custom attribute. The following snippet shows how PieChart exposes a property called showText:

```
public boolean isShowText() {  
    return mShowText;  
}  
  
public void setShowText(boolean showText) {  
    mShowText = showText;  
    invalidate();  
    requestLayout();  
}
```

Notice that setShowText calls [invalidate\(\)](#) ([/reference/android/view/View.html#invalidate\(\)](#)) and [requestLayout\(\)](#) ([/reference/android/view/View.html#requestLayout\(\)](#)). These calls are crucial to ensure that the view behaves reliably. You have to invalidate the view after any change to its properties that might change its appearance, so that the system knows that it needs to be redrawn. Likewise, you need to request a new layout if a property changes that might affect the size or shape of the view. Forgetting these method calls can cause hard-to-find bugs.

Custom views should also support event listeners to communicate important events. For instance, PieChart exposes a custom event called `OnCurrentItemChanged` to notify listeners that the user has rotated the pie chart to focus on a new pie slice.

It's easy to forget to expose properties and events, especially when you're the only user of the custom view. Taking some time to carefully define your view's interface reduces future maintenance costs. A good rule to follow is to always expose any property that affects the visible appearance or behavior of your custom view.

## Design For Accessibility

Your custom view should support the widest range of users. This includes users with disabilities that prevent them from seeing or using a touchscreen. To support users with disabilities, you should:

- Label your input fields using the `android:contentDescription` attribute
- Send accessibility events by calling `sendAccessibilityEvent()` when appropriate.
- Support alternate controllers, such as D-pad and trackball

For more information on creating accessible views, see [Making Applications Accessible](#) ([/guide/topics/ui/accessibility/apps.html#custom-views](#)) in the Android Developers Guide.

# Custom Drawing

The most important part of a custom view is its appearance. Custom drawing can be easy or complex according to your application's needs. This lesson covers some of the most common operations.

## Override onDraw()

The most important step in drawing a custom view is to override the [onDraw\(\) \(/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)\)](#) method. The parameter to [onDraw\(\) \(/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)\)](#) is a [Canvas \(/reference/android/graphics/Canvas.html\)](#) object that the view can use to draw itself. The [Canvas \(/reference/android/graphics/Canvas.html\)](#) class defines methods for drawing text, lines, bitmaps, and many other graphics primitives. You can use these methods in [onDraw\(\) \(/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)\)](#) to create your custom user interface (UI).

Before you can call any drawing methods, though, it's necessary to create a [Paint \(/reference/android/graphics/Paint.html\)](#) object. The next section discusses [Paint \(/reference/android/graphics/Paint.html\)](#) in more detail.

## Create Drawing Objects

The [android.graphics \(/reference/android/graphics/package-summary.html\)](#) framework divides drawing into two areas:

- What to draw, handled by [Canvas](#)
- How to draw, handled by [Paint](#).

For instance, [Canvas \(/reference/android/graphics/Canvas.html\)](#) provides a method to draw a line, while [Paint \(/reference/android/graphics/Paint.html\)](#) provides methods to define that line's color. [Canvas \(/reference/android/graphics/Canvas.html\)](#) has a method to draw a rectangle, while [Paint \(/reference/android/graphics/Paint.html\)](#) defines whether to fill that rectangle with a color or leave it empty. Simply put, [Canvas \(/reference/android/graphics/Canvas.html\)](#) defines shapes that you can draw on the screen, while [Paint \(/reference/android/graphics/Paint.html\)](#) defines the color, style, font, and so forth of each shape you draw.

So, before you draw anything, you need to create one or more [Paint \(/reference/android/graphics/Paint.html\)](#) objects. The PieChart example does this in a method called `init`, which is called from the constructor:

### THIS LESSON TEACHES YOU TO

1. [Override onDraw\(\)](#)
2. [Create Drawing Objects](#)
3. [Handle Layout Events](#)
4. [Draw!](#)

### YOU SHOULD ALSO READ

- [Canvas and Drawables](#)

### TRY IT OUT

[Download the sample](#)

CustomView.zip

```
private void init() {  
    mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
    mTextPaint.setColor(mTextColor);  
    if (mTextHeight == 0) {  
        mTextHeight = mTextPaint.getTextSize();  
    } else {  
        mTextPaint.setTextSize(mTextHeight);  
    }  
  
    mPiePaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
    mPiePaint.setStyle(Paint.Style.FILL);  
    mPiePaint.setTextSize(mTextHeight);  
  
    mShadowPaint = new Paint(0);  
    mShadowPaint.setColor(0xff101010);  
    mShadowPaint.setMaskFilter(new BlurMaskFilter(8, BlurMaskFilter.Blur.NORMAL));  
  
    ...  
}
```

Creating objects ahead of time is an important optimization. Views are redrawn very frequently, and many drawing objects require expensive initialization. Creating drawing objects within your [onDraw\(\)](#) ([/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)](#)) method significantly reduces performance and can make your UI appear sluggish.

## Handle Layout Events

In order to properly draw your custom view, you need to know what size it is. Complex custom views often need to perform multiple layout calculations depending on the size and shape of their area on screen. You should never make assumptions about the size of your view on the screen. Even if only one app uses your view, that app needs to handle different screen sizes, multiple screen densities, and various aspect ratios in both portrait and landscape mode.

Although [View](#) ([/reference/android/view/View.html](#)) has many methods for handling measurement, most of them do not need to be overridden. If your view doesn't need special control over its size, you only need to override one method: [onSizeChanged\(\)](#) ([/reference/android/view/View.html#onSizeChanged\(int, int, int, int\)](#)).

[onSizeChanged\(\)](#) ([/reference/android/view/View.html#onSizeChanged\(int, int, int, int\)](#)) is called when your view is first assigned a size, and again if the size of your view changes for any reason. Calculate positions, dimensions, and any other values related to your view's size in [onSizeChanged\(\)](#) ([/reference/android/view/View.html#onSizeChanged\(int, int, int, int\)](#)), instead of recalculating them every time you draw. In the PieChart example, [onSizeChanged\(\)](#) ([/reference/android/view/View.html#onSizeChanged\(int, int, int, int\)](#)) is where the PieChart view calculates the bounding rectangle of the pie chart and the relative position of the text label and other visual elements.

When your view is assigned a size, the layout manager assumes that the size includes all of the view's padding. You must handle the padding values when you calculate your view's size. Here's a

snippet from PieChart.onSizeChanged() that shows how to do this:

```
// Account for padding
float xpad = (float)(getPaddingLeft() + getPaddingRight());
float ypad = (float)(getPaddingTop() + getPaddingBottom());

// Account for the label
if (mShowText) xpad += mTextWidth;

float ww = (float)w - xpad;
float hh = (float)h - ypad;

// Figure out how big we can make the pie.
float diameter = Math.min(ww, hh);
```

If you need finer control over your view's layout parameters, implement [onMeasure\(\)](#) ([/reference/android/view/View.html#onMeasure\(int, int\)](#)). This method's parameters are [View.MeasureSpec](#) ([/reference/android/view/View.MeasureSpec.html](#)) values that tell you how big your view's parent wants your view to be, and whether that size is a hard maximum or just a suggestion. As an optimization, these values are stored as packed integers, and you use the static methods of [View.MeasureSpec](#) ([/reference/android/view/View.MeasureSpec.html](#)) to unpack the information stored in each integer.

Here's an example implementation of [onMeasure\(\)](#) ([/reference/android/view/View.html#onMeasure\(int, int\)](#)). In this implementation, PieChart attempts to make its area big enough to make the pie as big as its label:

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    // Try for a width based on our minimum
    int minw = getPaddingLeft() + getPaddingRight() + getSuggestedMinimumWidth();
    int w = resolveSizeAndState(minw, widthMeasureSpec, 1);

    // Whatever the width ends up being, ask for a height that would let the pie
    // get as big as it can
    int minh = MeasureSpec.getSize(w) - (int)mTextWidth + getPaddingBottom() + get
    int h = resolveSizeAndState(MeasureSpec.getSize(w) - (int)mTextWidth, heightMe

    setMeasuredDimension(w, h);
}
```

There are three important things to note in this code:

- The calculations take into account the view's padding. As mentioned earlier, this is the view's responsibility.
- The helper method [resolveSizeAndState\(\)](#) is used to create the final width and height values. This helper returns an appropriate [View.MeasureSpec](#) value by comparing the view's desired size to the spec passed into [onMeasure\(\)](#).
- [onMeasure\(\)](#) has no return value. Instead, the method communicates its results by calling

`setMeasuredDimension()`. Calling this method is mandatory. If you omit this call, the `View` class throws a runtime exception.

## Draw!

Once you have your object creation and measuring code defined, you can implement `onDraw()` ([/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)](#)). Every view implements `onDraw()` ([/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)](#)) differently, but there are some common operations that most views share:

- Draw text using `drawText()`. Specify the typeface by calling `setTypeface()`, and the text color by calling `setColor()`.
- Draw primitive shapes using `drawRect()`, `drawOval()`, and `drawArc()`. Change whether the shapes are filled, outlined, or both by calling `setStyle()`.
- Draw more complex shapes using the `Path` class. Define a shape by adding lines and curves to a `Path` object, then draw the shape using `drawPath()`. Just as with primitive shapes, paths can be outlined, filled, or both, depending on the `setStyle()`.
- Define gradient fills by creating `LinearGradient` objects. Call `setShader()` to use your `LinearGradient` on filled shapes.
- Draw bitmaps using `drawBitmap()`.

For example, here's the code that draws PieChart. It uses a mix of text, lines, and shapes.

```
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    // Draw the shadow
    canvas.drawOval(
        mShadowBounds,
        mShadowPaint
    );

    // Draw the label text
    canvas.drawText(mData.get(mCurrentItem).mLabel, mTextX, mTextY, mTextPaint);

    // Draw the pie slices
    for (int i = 0; i < mData.size(); ++i) {
        Item it = mData.get(i);
        mPiePaint.setShader(it.mShader);
        canvas.drawArc(mBounds,
            360 - it.mEndAngle,
            it.mEndAngle - it.mStartAngle,
            true, mPiePaint);
    }

    // Draw the pointer
    canvas.drawLine(mTextX, mPointerY, mPointerX, mPointerY, mTextPaint);
    canvas.drawCircle(mPointerX, mPointerY, mPointerSize, mTextPaint);
}
```

}

# Making the View Interactive

Drawing a UI is only one part of creating a custom view. You also need to make your view respond to user input in a way that closely resembles the real-world action you're mimicking. Objects should always act in the same way that real objects do. For example, images should not immediately pop out of existence and reappear somewhere else, because objects in the real world don't do that. Instead, images should move from one place to another.

Users also sense subtle behavior or feel in an interface, and react best to subtleties that mimic the real world. For example, when users fling a UI object, they should sense friction at the beginning that delays the motion, and then at the end sense momentum that carries the motion beyond the fling.

This lesson demonstrates how to use features of the Android framework to add these real-world behaviors to your custom view.

## THIS LESSON TEACHES YOU TO

1. [Handle Input Gestures](#)
2. [Create Physically Plausible Motion](#)
3. [Make Your Transitions Smooth](#)

## YOU SHOULD ALSO READ

- [Input Events](#)
- [Property Animation](#)

## TRY IT OUT

[Download the sample](#)

CustomView.zip

## Handle Input Gestures

Like many other UI frameworks, Android supports an input event model. User actions are turned into events that trigger callbacks, and you can override the callbacks to customize how your application responds to the user. The most common input event in the Android system is *touch*, which triggers [onTouchEvent\(android.view.MotionEvent\)](#) ([/reference/android/view/View.html#onTouchEvent\(android.view.MotionEvent\)](#)). Override this method to handle the event:

```
@Override  
public boolean onTouchEvent(MotionEvent event) {  
    return super.onTouchEvent(event);  
}
```

Touch events by themselves are not particularly useful. Modern touch UIs define interactions in terms of gestures such as tapping, pulling, pushing, flinging, and zooming. To convert raw touch events into gestures, Android provides [GestureDetector](#) ([/reference/android/view/GestureDetector.html](#)).

Construct a [GestureDetector](#) ([/reference/android/view/GestureDetector.html](#)) by passing in an instance of a class that implements [GestureDetector.OnGestureListener](#) ([/reference/android/view/GestureDetector.OnGestureListener.html](#)). If you only want to process a few gestures, you can extend [GestureDetector.SimpleOnGestureListener](#) ([/reference/android/view/GestureDetector.SimpleOnGestureListener.html](#)) instead of implementing the

[GestureDetector.OnGestureListener](#) (/reference/android/view/GestureDetector.OnGestureListener.html) interface. For instance, this code creates a class that extends [GestureDetector.SimpleOnGestureListener](#) (/reference/android/view/GestureDetector.SimpleOnGestureListener.html) and overrides [onDown\(MotionEvent\)](#) (/reference/android/view/GestureDetector.SimpleOnGestureListener.html#onDown(android.view MotionEvent)).

```
class mListener extends GestureDetector.SimpleOnGestureListener {  
    @Override  
    public boolean onDown(MotionEvent e) {  
        return true;  
    }  
}  
mDetector = new GestureDetector(PieChart.this.getContext(), new mListener());
```

Whether or not you use [GestureDetector.SimpleOnGestureListener](#) (/reference/android/view/GestureDetector.SimpleOnGestureListener.html), you must always implement an [onDown\(\)](#) (/reference/android/view/GestureDetector.OnGestureListener.html#onDown(android.view MotionEvent)) method that returns true. This step is necessary because all gestures begin with an [onDown\(\)](#) (/reference/android/view/GestureDetector.OnGestureListener.html#onDown(android.view MotionEvent)) message. If you return false from [onDown\(\)](#) (/reference/android/view/GestureDetector.OnGestureListener.html#onDown(android.view MotionEvent)), as [GestureDetector.SimpleOnGestureListener](#) (/reference/android/view/GestureDetector.SimpleOnGestureListener.html) does, the system assumes that you want to ignore the rest of the gesture, and the other methods of [GestureDetector.OnGestureListener](#) (/reference/android/view/GestureDetector.OnGestureListener.html) never get called. The only time you should return false from [onDown\(\)](#) (/reference/android/view/GestureDetector.OnGestureListener.html#onDown(android.view MotionEvent)) is if you truly want to ignore an entire gesture. Once you've implemented [GestureDetector.OnGestureListener](#) (/reference/android/view/GestureDetector.OnGestureListener.html) and created an instance of [GestureDetector](#) (/reference/android/view/GestureDetector.html), you can use your [GestureDetector](#) (/reference/android/view/GestureDetector.html) to interpret the touch events you receive in [onTouchEvent\(\)](#) (/reference/android/view/GestureDetector.html#onTouchEvent(android.view MotionEvent)).

```
@Override  
public boolean onTouchEvent(MotionEvent event) {  
    boolean result = mDetector.onTouchEvent(event);  
    if (!result) {  
        if (event.getAction() == MotionEvent.ACTION_UP) {  
            stopScrolling();  
            result = true;  
        }  
    }  
    return result;  
}
```

When you pass [onTouchEvent\(\)](#) (/reference/android

[/view/GestureDetector.html#onTouchEvent\(android.view.MotionEvent\)](#)) a touch event that it doesn't recognize as part of a gesture, it returns false. You can then run your own custom gesture-detection code.

## Create Physically Plausible Motion

Gestures are a powerful way to control touchscreen devices, but they can be counterintuitive and difficult to remember unless they produce physically plausible results. A good example of this is the *fling* gesture, where the user quickly moves a finger across the screen and then lifts it. This gesture makes sense if the UI responds by moving quickly in the direction of the fling, then slowing down, as if the user had pushed on a flywheel and set it spinning.

However, simulating the feel of a flywheel isn't trivial. A lot of physics and math are required to get a flywheel model working correctly. Fortunately, Android provides helper classes to simulate this and other behaviors. The [Scroller \(/reference/android/widget/Scroller.html\)](#) class is the basis for handling flywheel-style *fling* gestures.

To start a fling, call [fling\(\) \(/reference/android/widget/Scroller.html#fling\(int, int, int, int, int, int\)\)](#) with the starting velocity and the minimum and maximum x and y values of the fling. For the velocity value, you can use the value computed for you by [GestureDetector \(/reference/android/view/GestureDetector.html\)](#).

```
@Override  
public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY)  
    mScroller.fling(currentX, currentY, velocityX / SCALE, velocityY / SCALE, minX  
    postInvalidate();  
}
```

Note: Although the velocity calculated by [GestureDetector \(/reference/android/view/GestureDetector.html\)](#) is physically accurate, many developers feel that using this value makes the fling animation too fast. It's common to divide the x and y velocity by a factor of 4 to 8.

The call to [fling\(\) \(/reference/android/widget/Scroller.html#fling\(int, int, int, int, int, int, int\)\)](#) sets up the physics model for the fling gesture. Afterwards, you need to update the [Scroller \(/reference/android/widget/Scroller.html\)](#) by calling [Scroller.computeScrollOffset\(\) \(/reference/android/widget/Scroller.html#computeScrollOffset\(\)\)](#) at regular intervals. [computeScrollOffset\(\) \(/reference/android/widget/Scroller.html#computeScrollOffset\(\)\)](#) updates the [Scroller \(/reference/android/widget/Scroller.html\)](#) object's internal state by reading the current time and using the physics model to calculate the x and y position at that time. Call [getCurrX\(\) \(/reference/android/widget/Scroller.html#getCurrX\(\)\)](#) and [getCurrY\(\) \(/reference/android/widget/Scroller.html#getCurrY\(\)\)](#) to retrieve these values.

Most views pass the [Scroller \(/reference/android/widget/Scroller.html\)](#) object's x and y position directly to [scrollTo\(\) \(/reference/android/view/View.html#scrollTo\(int, int\)\)](#). The PieChart example is a little different: it uses the current scroll y position to set the rotational angle of the chart.

```
if (!mScroller.isFinished()) {  
    mScroller.computeScrollOffset();  
    setPieRotation(mScroller.getCurrY());  
}
```

The [Scroller](#) ([/reference/android/widget/Scroller.html](#)) class computes scroll positions for you, but it does not automatically apply those positions to your view. It's your responsibility to make sure you get and apply new coordinates often enough to make the scrolling animation look smooth. There are two ways to do this:

- Call [postInvalidate\(\)](#) after calling [fling\(\)](#), in order to force a redraw. This technique requires that you compute scroll offsets in [onDraw\(\)](#) and call [postInvalidate\(\)](#) every time the scroll offset changes.
- Set up a [ValueAnimator](#) to animate for the duration of the fling, and add a listener to process animation updates by calling [addUpdateListener\(\)](#).

The PieChart example uses the second approach. This technique is slightly more complex to set up, but it works more closely with the animation system and doesn't require potentially unnecessary view invalidation. The drawback is that [ValueAnimator](#) ([/reference/android/animation/ValueAnimator.html](#)) is not available prior to API level 11, so this technique cannot be used on devices running Android versions lower than 3.0.

Note: [ValueAnimator](#) ([/reference/android/animation/ValueAnimator.html](#)) isn't available prior to API level 11, but you can still use it in applications that target lower API levels. You just need to make sure to check the current API level at runtime, and omit the calls to the view animation system if the current level is less than 11.

```
mScroller = new Scroller(getContext(), null, true);  
mScrollAnimator = ValueAnimator.ofFloat(0,1);  
mScrollAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener  
    @Override  
    public void onAnimationUpdate(ValueAnimator valueAnimator) {  
        if (!mScroller.isFinished()) {  
            mScroller.computeScrollOffset();  
            setPieRotation(mScroller.getCurrY());  
        } else {  
            mScrollAnimator.cancel();  
            onScrollFinished();  
        }  
    }  
});
```

## Make Your Transitions Smooth

Users expect a modern UI to transition smoothly between states. UI elements fade in and out instead of appearing and disappearing. Motions begin and end smoothly instead of starting and stopping abruptly. The Android [property animation framework](#) ([/guide/topics/graphics/prop-animation.html](#)),

introduced in Android 3.0, makes smooth transitions easy.

To use the animation system, whenever a property changes that will affect your view's appearance, do not change the property directly. Instead, use [ValueAnimator](#) ([/reference/android/animation/ValueAnimator.html](#)) to make the change. In the following example, modifying the currently selected pie slice in PieChart causes the entire chart to rotate so that the selection pointer is centered in the selected slice. [ValueAnimator](#) ([/reference/android/animation/ValueAnimator.html](#)) changes the rotation over a period of several hundred milliseconds, rather than immediately setting the new rotation value.

```
mAutoCenterAnimator = ObjectAnimator.ofInt(PieChart.this, "PieRotation", 0);
mAutoCenterAnimator.setIntValues(targetAngle);
mAutoCenterAnimator.setDuration(AUTOCENTER_ANIM_DURATION);
mAutoCenterAnimator.start();
```

If the value you want to change is one of the base [View](#) ([/reference/android/view/View.html](#)) properties, doing the animation is even easier, because Views have a built-in [ViewPropertyAnimator](#) ([/reference/android/view/ViewPropertyAnimator.html](#)) that is optimized for simultaneous animation of multiple properties. For example:

```
animate().rotation(targetAngle).setDuration(ANIM_DURATION).start();
```

# Optimizing the View

Now that you have a well-designed view that responds to gestures and transitions between states, you need to ensure that the view runs fast. To avoid a UI that feels sluggish or stutters during playback, you must ensure that your animations consistently run at 60 frames per second.

## Do Less, Less Frequently

To speed up your view, eliminate unnecessary code from routines that are called frequently. Start by working on [onDraw\(\) \(/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)\)](#), which will give you the biggest payback. In particular you should eliminate allocations in [onDraw\(\) \(/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)\)](#), because allocations may lead to a garbage collection that would cause a stutter. Allocate objects during initialization, or between animations. Never make an allocation while an animation is running.

In addition to making [onDraw\(\) \(/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)\)](#) leaner, you should also make sure it's called as infrequently as possible. Most calls to [onDraw\(\) \(/reference/android/view/View.html#onDraw\(android.graphics.Canvas\)\)](#) are the result of a call to [invalidate\(\) \(/reference/android/view/View.html#invalidate\(\)\)](#), so eliminate unnecessary calls to [invalidate\(\) \(/reference/android/view/View.html#invalidate\(\)\)](#). When possible, call the four-parameter variant of [invalidate\(\) \(/reference/android/view/View.html#invalidate\(\)\)](#) rather than the version that takes no parameters. The no-parameter variant invalidates the entire view, while the four-parameter variant invalidates only a specified portion of the view. This approach allows draw calls to be more efficient and can eliminate unnecessary invalidation of views that fall outside the invalid rectangle.

Another very expensive operation is traversing layouts. Any time a view calls [requestLayout\(\) \(/reference/android/view/View.html#requestLayout\(\)\)](#), the Android UI system needs to traverse the entire view hierarchy to find out how big each view needs to be. If it finds conflicting measurements, it may need to traverse the hierarchy multiple times. UI designers sometimes create deep hierarchies of nested [ViewGroup \(/reference/android/view/ViewGroup.html\)](#) objects in order to get the UI to behave properly. These deep view hierarchies cause performance problems. Make your view hierarchies as shallow as possible.

If you have a complex UI, you should consider writing a custom [ViewGroup \(/reference/android/view/ViewGroup.html\)](#) to perform its layout. Unlike the built-in views, your custom view can make application-specific assumptions about the size and shape of its children, and thus avoid traversing its children to calculate measurements. The PieChart example shows how to extend [ViewGroup \(/reference/android/view/ViewGroup.html\)](#) as part of a custom view. PieChart has child views, but it never measures them. Instead, it sets their sizes directly according to its own custom layout

### THIS LESSON TEACHES YOU TO

1. [Do Less, Less Frequently](#)
2. [Use Hardware Acceleration](#)

### YOU SHOULD ALSO READ

- [Hardware Acceleration](#)

### TRY IT OUT

[Download the sample](#)

CustomView.zip

algorithm.

## Use Hardware Acceleration

As of Android 3.0, the Android 2D graphics system can be accelerated by the GPU (Graphics Processing Unit) hardware found in most newer Android devices. GPU hardware acceleration can result in a tremendous performance increase for many applications, but it isn't the right choice for every application. The Android framework gives you the ability to finely control which parts of your application are or are not hardware accelerated.

See [Hardware Acceleration \(/guide/topics/graphics/hardware-accel.html\)](#) in the Android Developers Guide for directions on how to enable acceleration at the application, activity, or window level. Notice that in addition to the directions in the developer guide, you must also set your application's target API to 11 or higher by specifying `<uses-sdk android:targetSdkVersion="11"/>` in your `AndroidManifest.xml` file.

Once you've enabled hardware acceleration, you may or may not see a performance increase. Mobile GPUs are very good at certain tasks, such as scaling, rotating, and translating bitmapped images. They are not particularly good at other tasks, such as drawing lines or curves. To get the most out of GPU acceleration, you should maximize the number of operations that the GPU is good at, and minimize the number of operations that the GPU isn't good at.

In the PieChart example, for instance, drawing the pie is relatively expensive. Redrawing the pie each time it's rotated causes the UI to feel sluggish. The solution is to place the pie chart into a child [View \(/reference/android/view/View.html\)](#) and set that [View \(/reference/android/view/View.html\)](#)'s [layer type \(/reference/android/view/View.html#setLayerType\(int, android.graphics.Paint\)\)](#) to [LAYER TYPE HARDWARE \(/reference/android/view/View.html#LAYER\\_TYPE\\_HARDWARE\)](#), so that the GPU can cache it as a static image. The sample defines the child view as an inner class of PieChart, which minimizes the amount of code changes that are needed to implement this solution.

```
private class PieView extends View {

    public PieView(Context context) {
        super(context);
        if (!isInEditMode()) {
            setLayerType(View.LAYER_TYPE_HARDWARE, null);
        }
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);

        for (Item it : mData) {
            mPiePaint.setShader(it.mShader);
            canvas.drawArc(mBounds,
                360 - it.mEndAngle,
                it.mEndAngle - it.mStartAngle,
```

```
        true, mPiePaint);
    }

@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    mBounds = new RectF(0, 0, w, h);
}

RectF mBounds;
}
```

After this code change, `PieChart.PieView.onDraw()` is called only when the view is first shown. During the rest of the application's lifetime, the pie chart is cached as an image, and redrawn at different rotation angles by the GPU. GPU hardware is particularly good at this sort of thing, and the performance difference is immediately noticeable.

There is a tradeoff, though. Caching images as hardware layers consumes video memory, which is a limited resource. For this reason, the final version of `PieChart.PieView` only sets its layer type to [LAYER\\_TYPE\\_HARDWARE](#) ([/reference/android/view/View.html#LAYER\\_TYPE\\_HARDWARE](#)) while the user is actively scrolling. At all other times, it sets its layer type to [LAYER\\_TYPE\\_NONE](#) ([/reference/android/view/View.html#LAYER\\_TYPE\\_NONE](#)), which allows the GPU to stop caching the image.

Finally, don't forget to profile your code. Techniques that improve performance on one view might negatively affect performance on another.

# Adding Search Functionality

Android's built-in search features offer apps an easy way to provide a consistent search experience for all users. There are two ways to implement search in your app depending on the version of Android that is running on the device. This class covers how to add search with [SearchView](#) ([/reference/android/widget/SearchView.html](#)), which was introduced in Android 3.0, while maintaining backward compatibility with older versions of Android by using the default search dialog provided by the system.

## Lessons

---

### **Setting Up the Search Interface**

Learn how to add a search interface to your app and how to configure an activity to handle search queries.

### **Storing and Searching for Data**

Learn a simple way to store and search for data in a SQLite virtual database table.

### **Remaining Backward Compatible**

Learn how to keep search features backward compatible with older devices by using.

---

### **DEPENDENCIES AND PREREQUISITES**

---

- Android 3.0 or later (with some support for Android 2.1)
- Experience building an Android [User Interface](#)

---

### **YOU SHOULD ALSO READ**

---

- [Search](#)
- [Searchable Dictionary Sample App](#)

# Setting Up the Search Interface

Beginning in Android 3.0, using the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) widget as an item in the action bar is the preferred way to provide search in your app. Like with all items in the action bar, you can define the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) to show at all times, only when there is room, or as a collapsible action, which displays the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) as an icon initially, then takes up the entire action bar as a search field when the user clicks the icon.

Note: Later in this class, you will learn how to make your app compatible down to Android 2.1 (API level 7) for devices that do not support [SearchView](#) ([/reference/android/widget/SearchView.html](#)).

## THIS LESSON TEACHES YOU TO

- [Add the Search View to the Action Bar](#)
- [Create a Searchable Configuration](#)
- [Create a Searchable Activity](#)

## YOU SHOULD ALSO READ:

- [Action Bar](#)

## Add the Search View to the Action Bar

To add a [SearchView](#) ([/reference/android/widget/SearchView.html](#)) widget to the action bar, create a file named `res/menu/options_menu.xml` in your project and add the following code to the file. This code defines how to create the search item, such as the icon to use and the title of the item. The `collapseActionView` attribute allows your [SearchView](#) ([/reference/android/widget/SearchView.html](#)) to expand to take up the whole action bar and collapse back down into a normal action bar item when not in use. Because of the limited action bar space on handset devices, using the `collapsibleActionView` attribute is recommended to provide a better user experience.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/search"
          android:title="@string/search_title"
          android:icon="@drawable/ic_search"
          android:showAsAction="collapseActionView|ifRoom"
          android:actionViewClass="android.widget.SearchView" />
</menu>
```

Note: If you already have an existing XML file for your menu items, you can add the `<item>` element to that file instead.

To display the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) in the action bar, inflate the XML menu resource (`res/menu/options_menu.xml`) in the [onCreateOptionsMenu\(\)](#) ([/reference/android/app/Activity.html#onCreateOptionsMenu\(android.view.Menu\)](#)) method of your activity:

```
@Override
```

```
public boolean onCreateOptionsMenu(Menu menu) {  
    MenuInflater inflater = getMenuInflater();  
    inflater.inflate(R.menu.options_menu, menu);  
  
    return true;  
}
```

If you run your app now, the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) appears in your app's action bar, but it isn't functional. You now need to define *how* the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) behaves.

## Create a Searchable Configuration

A [searchable configuration](#) (<http://developer.android.com/guide/topics/search/searchable-config.html>) defines how the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) behaves and is defined in a res/xml/searchable.xml file. At a minimum, a searchable configuration must contain an android:label attribute that has the same value as the android:label attribute of the [<application>](#) ([/guide/topics/manifest/application-element.html](#)) or [<activity>](#) ([/guide/topics/manifest/activity-element.html](#)) element in your Android manifest. However, we also recommend adding an android:hint attribute to give the user an idea of what to enter into the search box:

```
<?xml version="1.0" encoding="utf-8"?>  
  
<searchable xmlns:android="http://schemas.android.com/apk/res/android"  
    android:label="@string/app_name"  
    android:hint="@string/search_hint" />
```

In your application's manifest file, declare a [<meta-data>](#) ([/guide/topics/manifest/meta-data-element.html](#)) element that points to the res/xml/searchable.xml file, so that your application knows where to find it. Declare the element in an [<activity>](#) that you want to display the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) in:

```
<activity ... >  
    ...  
    <meta-data android:name="android.app.searchable"  
        android:resource="@xml/searchable" />  
  
</activity>
```

In the [onCreateOptionsMenu\(\)](#) ([/reference/android/app/Activity.html#onCreateOptionsMenu\(android.view.Menu\)](#)) method that you created before, associate the searchable configuration with the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) by calling [setSearchableInfo\(SearchableInfo\)](#) ([/reference/android/widget/SearchView.html#setSearchableInfo\(android.app.SearchableInfo\)](#)):

```
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    MenuInflater inflater = getMenuInflater();  
    inflater.inflate(R.menu.options_menu, menu);  
  
    // Associate searchable configuration with the SearchView  
    SearchManager searchManager =  
        (SearchManager) getSystemService(Context.SEARCH_SERVICE);  
    SearchView searchView =  
        (SearchView) menu.findItem(R.id.search).getActionView();  
    searchView.setSearchableInfo(  
        searchManager.getSearchableInfo(getApplicationContext()));  
  
    return true;  
}
```

The call to [getSearchableInfo\(\)](#) ([/reference/android/app/SearchManager.html#getSearchableInfo\(android.content.ComponentName\)](#)) obtains a [SearchableInfo](#) ([/reference/android/app/SearchableInfo.html](#)) object that is created from the searchable configuration XML file. When the searchable configuration is correctly associated with your [SearchView](#) ([/reference/android/widget/SearchView.html](#)), the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) starts an activity with the [ACTION\\_SEARCH](#) ([/reference/android/content/Intent.html#ACTION\\_SEARCH](#)) intent when a user submits a query. You now need an activity that can filter for this intent and handle the search query.

## Create a Searchable Activity

A [SearchView](#) ([/reference/android/widget/SearchView.html](#)) tries to start an activity with the [ACTION\\_SEARCH](#) ([/reference/android/content/Intent.html#ACTION\\_SEARCH](#)) when a user submits a search query. A searchable activity filters for the [ACTION\\_SEARCH](#) ([/reference/android/content/Intent.html#ACTION\\_SEARCH](#)) intent and searches for the query in some sort of data set. To create a searchable activity, declare an activity of your choice to filter for the [ACTION\\_SEARCH](#) ([/reference/android/content/Intent.html#ACTION\\_SEARCH](#)) intent:

```
<activity android:name=".SearchResultsActivity" ... >  
    ...  
    <intent-filter>  
        <action android:name="android.intent.action.SEARCH" />  
    </intent-filter>  
    ...  
</activity>
```

In your searchable activity, handle the [ACTION\\_SEARCH](#) ([/reference/android/content/Intent.html#ACTION\\_SEARCH](#)) intent by checking for it in your [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method.

Note: If your searchable activity launches in single top mode (`android:launchMode="singleTop"`), also handle the [ACTION\\_SEARCH](#) (`/reference/android/content/Intent.html#ACTION_SEARCH`) intent in the [onNewIntent\(\)](#) (`/reference/android/app/Activity.html#onNewIntent(android.content.Intent)`) method. In single top mode, only one instance of your activity is created and subsequent calls to start your activity do not create a new activity on the stack. This launch mode is useful so users can perform searches from the same activity without creating a new activity instance every time.

```
public class SearchResultsActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        handleIntent(getIntent());
    }

    @Override
    protected void onNewIntent(Intent intent) {
        ...
        handleIntent(intent);
    }

    private void handleIntent(Intent intent) {

        if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
            String query = intent.getStringExtra(SearchManager.QUERY);
            //use the query to search your data somehow
        }
    }
}
```

If you run your app now, the [SearchView](#) (`/reference/android/widget/SearchView.html`) can accept the user's query and start your searchable activity with the [ACTION\\_SEARCH](#) (`/reference/android/content/Intent.html#ACTION_SEARCH`) intent. It is now up to you to figure out how to store and search your data given a query.

# Storing and Searching for Data

There are many ways to store your data, such as in an online database, in a local SQLite database, or even in a text file. It is up to you to decide what is the best solution for your application. This lesson shows you how to create a SQLite virtual table that can provide robust full-text searching. The table is populated with data from a text file that contains a word and definition pair on each line in the file.

## THIS LESSON TEACHES YOU TO

- [Create the Virtual Table](#)
- [Populate the Virtual Table](#)
- [Search for the Query](#)

## Create the Virtual Table

A virtual table behaves similarly to a SQLite table, but reads and writes to an object in memory via callbacks, instead of to a database file. To create a virtual table, create a class for the table:

```
public class DatabaseTable {  
    private final DatabaseOpenHelper mDatabaseOpenHelper;  
  
    public DatabaseTable(Context context) {  
        mDatabaseOpenHelper = new DatabaseOpenHelper(context);  
    }  
}
```

Create an inner class in `DatabaseTable` that extends [SQLiteOpenHelper](#) ([/reference/android/database/sqlite/SQLiteOpenHelper.html](#)). The [SQLiteOpenHelper](#) ([/reference/android/database/sqlite/SQLiteOpenHelper.html](#)) class defines abstract methods that you must override so that your database table can be created and upgraded when necessary. For example, here is some code that declares a database table that will contain words for a dictionary app:

```
public class DatabaseTable {  
  
    private static final String TAG = "DictionaryDatabase";  
  
    //The columns we'll include in the dictionary table  
    public static final String COL_WORD = "WORD";  
    public static final String COL_DEFINITION = "DEFINITION";  
  
    private static final String DATABASE_NAME = "DICTIONARY";  
    private static final String FTS_VIRTUAL_TABLE = "FTS";  
    private static final int DATABASE_VERSION = 1;  
  
    private final DatabaseOpenHelper mDatabaseOpenHelper;
```

```
public DatabaseTable(Context context) {
    mDatabaseOpenHelper = new DatabaseOpenHelper(context);
}

private static class DatabaseOpenHelper extends SQLiteOpenHelper {

    private final Context mHelperContext;
    private SQLiteDatabase mDatabase;

    private static final String FTS_TABLE_CREATE =
        "CREATE VIRTUAL TABLE " + FTS_VIRTUAL_TABLE +
        " USING fts3 (" +
        COL_WORD + ", " +
        COL_DEFINITION + ")";

    DatabaseOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
        mHelperContext = context;
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        mDatabase = db;
        mDatabase.execSQL(FTS_TABLE_CREATE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
        Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
            + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS " + FTS_VIRTUAL_TABLE);
        onCreate(db);
    }
}
}
```

## Populate the Virtual Table

The table now needs data to store. The following code shows you how to read a text file (located in res/raw/definitions.txt) that contains words and their definitions, how to parse that file, and how to insert each line of that file as a row in the virtual table. This is all done in another thread to prevent the UI from locking. Add the following code to your DatabaseOpenHelper inner class.

**Tip:** You also might want to set up a callback to notify your UI activity of this thread's completion.

```
private void loadDictionary() {
    new Thread(new Runnable() {
```

```
public void run() {
    try {
        loadWords();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
}).start();
}

private void loadWords() throws IOException {
final Resources resources = mHelperContext.getResources();
InputStream inputStream = resources.openRawResource(R.rawdefinitions);
BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream))

try {
    String line;
    while ((line = reader.readLine()) != null) {
        String[] strings = TextUtils.split(line, "-");
        if (strings.length < 2) continue;
        long id = addWord(strings[0].trim(), strings[1].trim());
        if (id < 0) {
            Log.e(TAG, "unable to add word: " + strings[0].trim());
        }
    }
} finally {
    reader.close();
}
}

public long addWord(String word, String definition) {
ContentValues initialValues = new ContentValues();
initialValues.put(COL_WORD, word);
initialValues.put(COL_DEFINITION, definition);

return mDatabase.insert(FTS_VIRTUAL_TABLE, null, initialValues);
}
```

Call the `loadDictionary()` method wherever appropriate to populate the table. A good place would be in the [onCreate\(\)](#) ([/reference/android/database/sqlite/SQLiteDatabase.html#onCreate\(android.database.sqlite.SQLiteDatabase\)](#)) method of the `DatabaseOpenHelper` class, right after you create the table:

```
@Override
public void onCreate(SQLiteDatabase db) {
    mDatabase = db;
    mDatabase.execSQL(FTS_TABLE_CREATE);
    loadDictionary();
```

}

## Search for the Query

When you have the virtual table created and populated, use the query supplied by your [SearchView](#) ([/reference/android/widget/SearchView.html](#)) to search the data. Add the following methods to the DatabaseTable class to build a SQL statement that searches for the query:

```
public Cursor getWordMatches(String query, String[] columns) {
    String selection = COL_WORD + " MATCH ?";
    String[] selectionArgs = new String[] {query+"*"};
    return query(selection, selectionArgs, columns);
}

private Cursor query(String selection, String[] selectionArgs, String[] columns)
    SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
    builder.setTables(FTS_VIRTUAL_TABLE);

    Cursor cursor = builder.query(mDatabaseOpenHelper.getReadableDatabase(),
        columns, selection, selectionArgs, null, null, null);

    if (cursor == null) {
        return null;
    } else if (!cursor.moveToFirst()) {
        cursor.close();
        return null;
    }
    return cursor;
}
```

Search for a query by calling `getWordMatches()`. Any matching results are returned in a [Cursor](#) ([/reference/android/database/Cursor.html](#)) that you can iterate through or use to build a [ListView](#) ([/reference/android/widget/ListView.html](#)). This example calls `getWordMatches()` in the `handleIntent()` method of the searchable activity. Remember that the searchable activity receives the query inside of the [ACTION\\_SEARCH](#) ([/reference/android/content/Intent.html#ACTION\\_SEARCH](#)) intent as an extra, because of the intent filter that you previously created:

```
DatabaseTable db = new DatabaseTable(this);

...
private void handleIntent(Intent intent) {
```

```
if (Intent.ACTION_SEARCH.equals(intent.getAction())) {  
    String query = intent.getStringExtra(SearchManager.QUERY);  
    Cursor c = db.getWordMatches(query, null);  
    //process Cursor and display results  
}  
}
```

# Remaining Backward Compatible

The [SearchView](#) ([/reference/android/widget/SearchView.html](#)) and action bar are only available on Android 3.0 and later. To support older platforms, you can fall back to the search dialog. The search dialog is a system provided UI that overlays on top of your application when invoked.

## THIS LESSON TEACHES YOU TO

- [Set Minimum and Target API levels](#)
- [Provide the Search Dialog for Older Devices](#)
- [Check the Android Build Version at Runtime](#)

## Set Minimum and Target API levels

To setup the search dialog, first declare in your manifest that you want to support older devices, but want to target Android 3.0 or later versions. When you do this, your application automatically uses the action bar on Android 3.0 or later and uses the traditional menu system on older devices:

```
<uses-sdk android:minSdkVersion="7" android:targetSdkVersion="15" />

<application>
  ...

```

## Provide the Search Dialog for Older Devices

To invoke the search dialog on older devices, call [onSearchRequested\(\)](#) ([/reference/android/app/Activity.html#onSearchRequested\(\)](#)) whenever a user selects the search menu item from the options menu. Because Android 3.0 and higher devices show the [SearchView](#) ([/reference/android/widget/SearchView.html](#)) in the action bar (as demonstrated in the first lesson), only versions older than 3.0 call [onOptionsItemSelected\(\)](#) ([/reference/android/app/Activity.html#onOptionsItemSelected\(android.view.MenuItem\)](#)) when the user selects the search menu item.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.search:
            onSearchRequested();
            return true;
        default:
            return false;
    }
}
```

## Check the Android Build Version at Runtime

At runtime, check the device version to make sure an unsupported use of [SearchView](#) ([/reference/android/widget/SearchView.html](#)) does not occur on older devices. In our example code, this happens in the [onCreateOptionsMenu\(\)](#) ([/reference/android/app/Activity.html#onCreateOptionsMenu\(android.view.Menu\)](#)) method:

```
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
  
    MenuInflater inflater = getMenuInflater();  
    inflater.inflate(R.menu.options_menu, menu);  
  
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {  
        SearchManager searchManager =  
            (SearchManager) getSystemService(Context.SEARCH_SERVICE);  
        SearchView searchView =  
            (SearchView) menu.findItem(R.id.search).getActionView();  
        searchView.setSearchableInfo(  
            searchManager.getSearchableInfo(getComponentName()));  
        searchView.setIconifiedByDefault(false);  
    }  
    return true;  
}
```

# Remembering Users

Android users get attached to their devices and to applications that they love. One way to make your application lovable is to make it personal. Android devices know who your user is, what services they have access to, and where they store your data. With your user's permission, you can use that information to make your application a richer, more personal experience.

In this class, you will learn multiple techniques for interacting with your user's identity, enabling you to:

- Personalize your app by remembering users by their account name(s)
- Authenticate the user to make sure they are who they say they are
- Gain permission to access the user's online data via services like the Google APIs
- Add a custom account to the user's device to authenticate your own back-end services

## Lessons

---

### **Remembering Your User**

Use [AccountManager](#) to learn the user's account name(s).

### **Authenticating to OAuth2 Services**

Use OAuth2 to help users get permission to access web services without needing to type in a login name or password.

### **Creating a Custom Account Type**

Add your own account type to the Android Account Manager.

### **REQUIREMENTS AND PREREQUISITES**

---

- Android 2.0 (API level 5) or higher
- Experience with [Services](#)
- Experience with [OAuth 2.0](#)

### **YOU SHOULD ALSO READ**

---

- [SampleSyncAdapter app](#)

# Remembering Your User

Everyone likes it when you remember their name. One of the simplest, most effective things you can do to make your app more lovable is to remember who your user is—especially when the user upgrades to a new device or starts carrying a tablet as well as a phone. But how do you know who your user is? And how do you recognize them on a new device?

For many applications, the answer is the [AccountManager \(/reference/android/accounts/AccountManager.html\)](#) APIs. With the user's permission, you can use Account Manager to fetch the account names that the user has stored on their device.

Integration with the user's accounts allows you to do a variety of things such as:

- Auto-fill forms with the user's email address.
- Retrieve an ID that is tied to a user, not the device.

## THIS LESSON TEACHES YOU TO

1. [Determine if AccountManager for You](#)
2. [Decide What Type of Account to Use](#)
3. [Request GET\\_ACCOUNT permission](#)
4. [Query AccountManager for a List of Accounts](#)
5. [Use the Account Object to Personalize Your App](#)
6. [Decide Whether an Account Name is Enough](#)

## Determine if AccountManager for You

Applications typically try to remember the user using one of three techniques:

- a. Ask the user to type in a username
- b. Retrieve a unique device ID to remember the device
- c. Retrieve a built-in account from [AccountManager](#)

Option (a) is problematic. First, asking the user to type something before entering your app will automatically make your app less appealing. Second, there's no guarantee that the username chosen will be unique.

Option (b) is less onerous for the user, but it's [tricky to get right \(http://android-developers.blogspot.com/2011/03/identifying-app-installations.html\)](#). More importantly, it only allows you to remember the user on one device. Imagine the frustration of someone who upgrades to a shiny new device, only to find that your app no longer remembers them.

Option (c) is the preferred technique. Account Manager allows you to get information about the accounts that are stored on the user's device. As we'll see in this lesson, using Account Manager lets you remember your user, no matter how many devices the user may own, by adding just a couple of extra taps to your UI.

## Decide What Type of Account to Use

Android devices can store multiple accounts from many different providers. When you query

[AccountManager](#) ([/reference/android/accounts/AccountManager.html](#)) for account names, you can choose to filter by account type. The account type is a string that uniquely identifies the entity that issued the account. For instance, Google accounts have type "com.google," while Twitter uses "com.twitter.android.auth.login."

## Request GET\_ACCOUNT permission

In order to get a list of accounts on the device, your app needs the [GET\\_ACCOUNTS](#) ([/reference/android/Manifest.permission.html#GET\\_ACCOUNTS](#)) permission. Add a [<uses-permission>](#) ([/guide/topics/manifest/uses-permission-element.html](#)) tag in your manifest file to request this permission:

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

## Query AccountManager for a List of Accounts

Once you decide what account type you're interested in, you need to query for accounts of that type. Get an instance of [AccountManager](#) ([/reference/android/accounts/AccountManager.html](#)) by calling [AccountManager.get\(\)](#) ([/reference/android/accounts/AccountManager.html#get\(android.content.Context\)](#)). Then use that instance to call [getAccountsByType\(\)](#) ([/reference/android/accounts/AccountManager.html#getAccountsByType\(java.lang.String\)](#)).

```
AccountManager am = AccountManager.get(this); // "this" references the current Context
Account[] accounts = am.getAccountsByType("com.google");
```

This returns an array of [Account](#) ([/reference/android/accounts/Account.html](#)) objects. If there's more than one [Account](#) ([/reference/android/accounts/Account.html](#)) in the array, you should present a dialog asking the user to select one.

## Use the Account Object to Personalize Your App

The [Account](#) ([/reference/android/accounts/Account.html](#)) object contains an account name, which for Google accounts is an email address. You can use this information in several different ways, such as:

- As suggestions in forms, so the user doesn't need to input account information by hand.
- As a key into your own online database of usage and personalization information.

## Decide Whether an Account Name is Enough

---

An account name is a good way to remember the user, but the [Account \(/reference/android/accounts/Account.html\)](#) object by itself doesn't protect your data or give you access to anything besides the user's account name. If your app needs to allow the user to go online to access private data, you'll need something stronger: authentication. The next lesson explains how to authenticate to existing online services. The lesson after that deals with writing a custom authenticator so that you can install your own account types.

# Authenticating to OAuth2 Services

In order to securely access an online service, users need to authenticate to the service—they need to provide proof of their identity. For an application that accesses a third-party service, the security problem is even more complicated. Not only does the user need to be authenticated to access the service, but the application also needs to be authorized to act on the user's behalf.

The industry standard way to deal with authentication to third-party services is the OAuth2 protocol. OAuth2 provides a single value, called an auth token, that represents both the user's identity and the application's authorization to act on the user's behalf. This lesson demonstrates connecting to a Google server that supports OAuth2. Although Google services are used as an example, the techniques demonstrated will work on any service that correctly supports the OAuth2 protocol.

Using OAuth2 is good for:

- Getting permission from the user to access an online service using his or her account.
- Authenticating to an online service on behalf of the user.
- Handling authentication errors.

## Gather Information

To begin using OAuth2, you need to know a few things about the API you're trying to access:

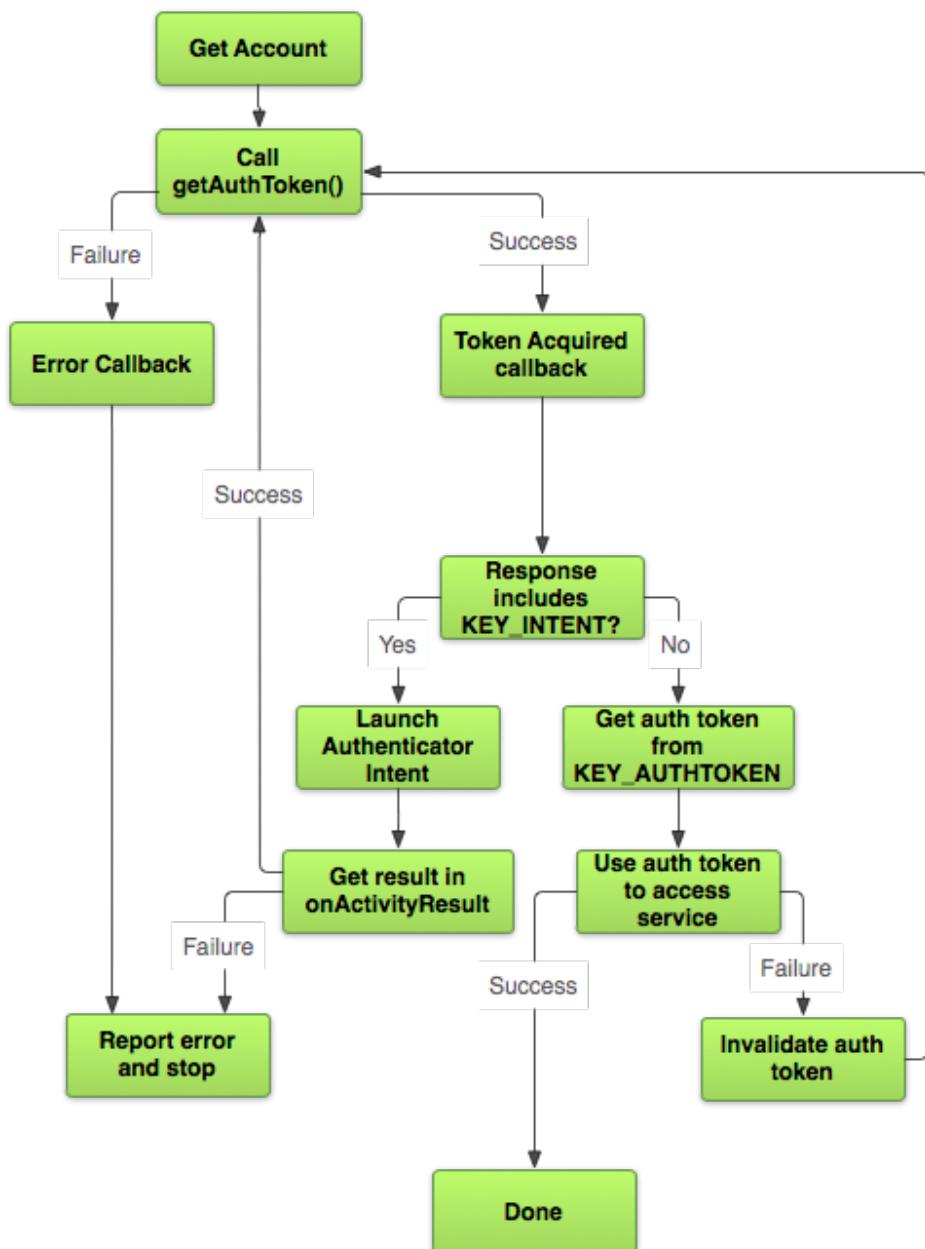
- The url of the service you want to access.
- The auth scope, which is a string that defines the specific type of access your app is asking for. For instance, the auth scope for read-only access to Google Tasks is View your tasks, while the auth scope for read-write access to Google Tasks is Manage Your Tasks.
- A client id and client secret, which are strings that identify your app to the service. You need to obtain these strings directly from the service owner. Google has a self-service system for obtaining client ids and secrets. The article [Getting Started with the Tasks API and OAuth 2.0 on Android](#) explains how to use this system to obtain these values for use with the Google Tasks API.

## Request an Auth Token

Now you're ready to request an auth token. This is a multi-step process.

### THIS LESSON TEACHES YOU TO

1. [Gather Information](#)
2. [Request an Auth Token](#)
3. [Request an Auth Token... Again](#)
4. [Connect to the Online Service](#)



To get an auth token you first need to request the [ACCOUNT\\_MANAGER](#) (`/reference/android/Manifest.permission.html#ACCOUNT_MANAGER`) to your manifest file. To actually do anything useful with the token, you'll also need to add the [INTERNET](#) (`/reference/android/Manifest.permission.html#INTERNET`) permission.

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

Once your app has these permissions set, you can call [AccountManager.getAuthToken\(\)](#) (`/reference/android/accounts/AccountManager.html#getAuthToken(android.accounts.Account,`

`java.lang.String, android.os.Bundle, android.app.Activity,`  
`android.accounts.AccountManagerCallback<android.os.Bundle>, android.os.Handler)`) to get the token.

Watch out! Calling methods on `AccountManager` (</reference/android/accounts/AccountManager.html>) can be tricky! Since account operations may involve network communication, most of the `AccountManager` (</reference/android/accounts/AccountManager.html>) methods are asynchronous. This means that instead of doing all of your auth work in one function, you need to implement it as a series of callbacks. For example:

```
AccountManager am = AccountManager.get(this);
Bundle options = new Bundle();

am.getAuthToken(
    myAccount_,                                // Account retrieved using getAccountsByType()
    "Manage your tasks",                        // Auth scope
    options,                                    // Authenticator-specific options
    this,                                       // Your activity
    new OnTokenAcquired(),                      // Callback called when a token is successful
    new Handler(new OnError())));                // Callback called if an error occurs
```

In this example, `OnTokenAcquired` is a class that extends `AccountManagerCallback` (</reference/android/accounts/AccountManagerCallback.html>). `AccountManager` (</reference/android/accounts/AccountManager.html>) calls `run()` ([/reference/android/accounts/AccountManagerCallback.html#run\(android.accounts.AccountManagerFuture<V>\)](/reference/android/accounts/AccountManagerCallback.html#run(android.accounts.AccountManagerFuture<V>))) on `OnTokenAcquired` with an `AccountManagerFuture` (</reference/android/accounts/AccountManagerFuture.html>) that contains a `Bundle` (</reference/android/os/Bundle.html>). If the call succeeded, the token is inside the `Bundle` (</reference/android/os/Bundle.html>).

Here's how you can get the token from the `Bundle` (</reference/android/os/Bundle.html>):

```
private class OnTokenAcquired implements AccountManagerCallback<Bundle> {
    @Override
    public void run(AccountManagerFuture<Bundle> result) {
        // Get the result of the operation from the AccountManagerFuture.
        Bundle bundle = result.getResult();

        // The token is a named value in the bundle. The name of the value
        // is stored in the constant AccountManager.KEY_AUTHTOKEN.
        token = bundle.getString(AccountManager.KEY_AUTHTOKEN);
        ...
    }
}
```

If all goes well, the `Bundle` (</reference/android/os/Bundle.html>) contains a valid token in the `KEY_AUTHTOKEN` ([/reference/android/accounts/AccountManager.html#KEY\\_AUTHTOKEN](/reference/android/accounts/AccountManager.html#KEY_AUTHTOKEN)) key and you're off to the races. Things don't always go that smoothly, though...

## Request an Auth Token... Again

Your first request for an auth token might fail for several reasons:

- An error in the device or network caused [AccountManager](#) to fail.
- The user decided not to grant your app access to the account.
- The stored account credentials aren't sufficient to gain access to the account.
- The cached auth token has expired.

Applications can handle the first two cases trivially, usually by simply showing an error message to the user. If the network is down or the user decided not to grant access, there's not much that your application can do about it. The last two cases are a little more complicated, because well-behaved applications are expected to handle these failures automatically.

The third failure case, having insufficient credentials, is communicated via the [Bundle](#) ([/reference/android/os/Bundle.html](#)) you receive in your [AccountManagerCallback](#) ([/reference/android/accounts/AccountManagerCallback.html](#)) ([OnTokenAcquired](#) from the previous example). If the [Bundle](#) ([/reference/android/os/Bundle.html](#)) includes an [Intent](#) ([/reference/android/content/Intent.html](#)) in the [KEY\\_INTENT](#) ([/reference/android/accounts/AccountManager.html#KEY\\_INTENT](#)) key, then the authenticator is telling you that it needs to interact directly with the user before it can give you a valid token.

There may be many reasons for the authenticator to return an [Intent](#) ([/reference/android/content/Intent.html](#)). It may be the first time the user has logged in to this account. Perhaps the user's account has expired and they need to log in again, or perhaps their stored credentials are incorrect. Maybe the account requires two-factor authentication or it needs to activate the camera to do a retina scan. It doesn't really matter what the reason is. If you want a valid token, you're going to have to fire off the [Intent](#) ([/reference/android/content/Intent.html](#)) to get it.

```
private class OnTokenAcquired implements AccountManagerCallback<Bundle> {
    @Override
    public void run(AccountManagerFuture<Bundle> result) {
        ...
        Intent launch = (Intent) result.get(AccountManager.KEY_INTENT);
        if (launch != null) {
            startActivityForResult(launch, 0);
            return;
        }
    }
}
```

Note that the example uses [startActivityForResult\(\)](#) ([/reference/android/app/Activity.html#startActivityForResult\(android.content.Intent, int\)](#)), so that you can capture the result of the [Intent](#) ([/reference/android/content/Intent.html](#)) by implementing [onActivityResult\(\)](#) ([/reference/android/app/Activity.html#onActivityResult\(int, int, android.content.Intent\)](#)) in your own activity. This is important! If you don't capture the result from the authenticator's response [Intent](#) ([/reference/android/content/Intent.html](#)), it's impossible to tell whether the user has successfully authenticated or not. If the result is [RESULT\\_OK](#) ([/reference/android/app/Activity.html#RESULT\\_OK](#)), then the authenticator has updated the stored credentials so that they are sufficient for the level of access you requested, and you should call [AccountManager.getAuthToken\(\)](#) ([/reference/android/accounts/AccountManager.html#getAuthToken\(android.accounts.Account, java.lang.String, android.os.Bundle, android.accounts.AccountManager.OnTokenAcquired, android.os.Handler\)](#).

```
/accounts/AccountManager.html#getAuthToken(android.accounts.Account, java.lang.String,  
android.os.Bundle, android.app.Activity, android.accounts.AccountManagerCallback<android.os.Bundle>,  
android.os.Handler)) again to request the new auth token.
```

The last case, where the token has expired, it is not actually an [AccountManager \(/reference/android/accounts/AccountManager.html\)](#) failure. The only way to discover whether a token is expired or not is to contact the server, and it would be wasteful and expensive for [AccountManager \(/reference/android/accounts/AccountManager.html\)](#) to continually go online to check the state of all of its tokens. So this is a failure that can only be detected when an application like yours tries to use the auth token to access an online service.

## Connect to the Online Service

The example below shows how to connect to a Google server. Since Google uses the industry standard OAuth2 protocol to authenticate requests, the techniques discussed here are broadly applicable. Keep in mind, though, that every server is different. You may find yourself needing to make minor adjustments to these instructions to account for your specific situation.

The Google APIs require you to supply four values with each request: the API key, the client ID, the client secret, and the auth key. The first three come from the Google API Console website. The last is the string value you obtained by calling [AccountManager.getAuthToken\(\) \(/reference/android/accounts/AccountManager.html#getAuthToken\(android.accounts.Account, java.lang.String, android.os.Bundle, android.app.Activity, android.accounts.AccountManagerCallback<android.os.Bundle>, android.os.Handler\)\)](#). You pass these to the Google Server as part of an HTTP request.

```
URL url = new URL("https://www.googleapis.com/tasks/v1/users/@me/lists?key=" + yo  
URLConnection conn = (HttpURLConnection) url.openConnection();  
conn.addRequestProperty("client_id", your client id);  
conn.addRequestProperty("client_secret", your client secret);  
conn.setRequestProperty("Authorization", "OAuth " + token);
```

If the request returns an HTTP error code of 401, then your token has been denied. As mentioned in the last section, the most common reason for this is that the token has expired. The fix is simple: call [AccountManager.invalidateAuthToken\(\) \(/reference/android/accounts/AccountManager.html#invalidateAuthToken\(java.lang.String, java.lang.String\)\)](#) and repeat the token acquisition dance one more time.

Because expired tokens are such a common occurrence, and fixing them is so easy, many applications just assume the token has expired before even asking for it. If renewing a token is a cheap operation for your server, you might prefer to call [AccountManager.invalidateAuthToken\(\) \(/reference/android/accounts/AccountManager.html#invalidateAuthToken\(java.lang.String, java.lang.String\)\)](#) before the first call to [AccountManager.getAuthToken\(\) \(/reference/android/accounts/AccountManager.html#getAuthToken\(android.accounts.Account, java.lang.String, android.os.Bundle, android.app.Activity, android.accounts.AccountManagerCallback<android.os.Bundle>, android.os.Handler\)\)](#), and spare yourself the need to request an auth token twice.

# Creating a Custom Account Type

So far we've talked about accessing Google APIs, which use accounts and users defined by Google. If you have your own online service, though, it won't have Google accounts or users, so what do you do? It turns out to be relatively straightforward to install new account types on a user's device. This lesson explains how to create a custom account type that works the same way as the built-in accounts do.

## Implement Your Custom Account Code

The first thing you'll need is a way to get credentials from the user. This may be as simple as a dialog box that asks for a name and a password. Or it may be a more exotic procedure like a one-time password or a biometric scan. Either way, it's your responsibility to implement the code that:

1. Collects credentials from the user
2. Authenticates the credentials with the server
3. Stores the credentials on the device

Typically all three of these requirements can be handled by one activity. We'll call this the authenticator activity.

Because they need to interact with the [AccountManager \(/reference/android/accounts/AccountManager.html\)](#) system, authenticator activities have certain requirements that normal activities don't. To make it easy to get things right, the Android framework supplies a base class, [AccountAuthenticatorActivity \(/reference/android/accounts/AccountAuthenticatorActivity.html\)](#), which you can extend to create your own custom authenticator.

How you address the first two requirements of an authenticator activity, credential collection and authentication, is completely up to you. (If there were only one way to do it, there'd be no need for "custom" account types, after all.) The third requirement has a canonical, and rather simple, implementation:

```
final Account account = new Account(mUsername, your_account_type);
mAccountManager.addAccountExplicitly(account, mPassword, null);
```

## Be Smart About Security!

It's important to understand that [AccountManager \(/reference/android/accounts/AccountManager.html\)](#) is not an encryption service or a keychain. It stores account credentials just as you pass them, in

### THIS LESSON TEACHES YOU TO

1. [Implement Your Custom Account Code](#)
2. [Be Smart About Security!](#)
3. [Extend AbstractAccountAuthenticator](#)
4. [Create an Authenticator Service](#)
5. [Distribute Your Service](#)

### YOU SHOULD ALSO READ

- [SampleSyncAdapter app](#)

plain text. On most devices, this isn't a particular concern, because it stores them in a database that is only accessible to root. But on a rooted device, the credentials would be readable by anyone with adb access to the device.

With this in mind, you shouldn't pass the user's actual password to [AccountManager.addAccountExplicitly\(\)](#). Instead, you should store a cryptographically secure token that would be of limited use to an attacker. If your user credentials are protecting something valuable, you should carefully consider doing something similar.

Remember: When it comes to security code, follow the "Mythbusters" rule: don't try this at home! Consult a security professional before implementing any custom account code.

Now that the security disclaimers are out of the way, it's time to get back to work. You've already implemented the meat of your custom account code; what's left is plumbing.

## Extend AbstractAccountAuthenticator

---

In order for the [AccountManager](#) to work with your custom account code, you need a class that implements the interfaces that [AccountManager](#) expects. This class is the *authenticator class*.

The easiest way to create an authenticator class is to extend [AbstractAccountAuthenticator](#) and implement its abstract methods. If you've worked through the previous lessons, the abstract methods of [AbstractAccountAuthenticator](#) should look familiar: they're the opposite side of the methods you called in the previous lesson to get account information and authorization tokens.

Implementing an authenticator class properly requires a number of separate pieces of code. First, [AbstractAccountAuthenticator](#) has seven abstract methods that you must override. Second, you need to add an [intent filter](#) for "android.accounts.AccountAuthenticator" to your application manifest (shown in the next section). Finally, you must supply two XML resources that define, among other things, the name of your custom account type and the icon that the system will display next to accounts of this type.

You can find a step-by-step guide to implementing a successful authenticator class and the XML files in the [AbstractAccountAuthenticator](#) documentation. There's also a sample implementation in the [SampleSyncAdapter](#) sample app (<http://developer.android.com/resources/samples/SampleSyncAdapter/index.html>).

As you read through the [SampleSyncAdapter](#) code, you'll notice that several of the methods return an intent in a bundle. This is the same intent that will be used to launch your custom authenticator activity. If your authenticator activity needs any special initialization parameters, you can attach them to the intent using [Intent.putExtra\(\)](#).

## Create an Authenticator Service

---

Now that you have an authenticator class, you need a place for it to live. Account authenticators need to be available to multiple applications and work in the background, so naturally they're required to run inside a [Service](#) ([/reference/android/app/Service.html](#)). We'll call this the authenticator service.

Your authenticator service can be very simple. All it needs to do is create an instance of your authenticator class in [onCreate\(\)](#) ([/reference/android/app/Service.html#onCreate\(\)](#)) and call [getIBinder\(\)](#) ([/reference/android/accounts/AbstractAccountAuthenticator.html#getIBinder\(\)](#)) in [onBind\(\)](#) ([/reference/android/app/Service.html#onBind\(android.content.Intent\)](#)). The [SampleSyncAdapter](#) (<http://developer.android.com/resources/samples/SampleSyncAdapter/index.html>) contains a good example of an authenticator service.

Don't forget to add a `<service>` tag to your manifest file and add an intent filter for the `AccountAuthenticator` intent and declare the account authenticator:

```
<service ...>
    <intent-filter>
        <action android:name="android.accounts.AccountAuthenticator" />
    </intent-filter>
    <meta-data android:name="android.accounts.AccountAuthenticator"
              android:resource="@xml/authenticator" />
</service>
```

## Distribute Your Service

---

You're done! The system now recognizes your account type, right alongside all the big name account types like "Google" and "Corporate." You can use the Accounts & Sync Settings page to add an account, and apps that ask for accounts of your custom type will be able to enumerate and authenticate just as they would with any other account type.

Of course, all of this assumes that your account service is actually installed on the device. If only one app will ever access the service, then this isn't a big deal—just bundle the service in the app. But if you want your account service to be used by more than one app, things get trickier. You don't want to bundle the service with all of your apps and have multiple copies of it taking up space on your user's device.

One solution is to place the service in one small, special-purpose APK. When an app wishes to use your custom account type, it can check the device to see if your custom account service is available. If not, it can direct the user to Google Play to download the service. This may seem like a great deal of trouble at first, but compared with the alternative of re-entering credentials for every app that uses your custom account, it's refreshingly easy.

# Sharing Content

One of the great things about Android applications is their ability to communicate and integrate with each other. Why reinvent functionality that isn't core to your application when it already exists in another application?

This class covers some common ways you can send and receive content between applications using [Intent](#) ([/reference/android/content/Intent.html](#)) APIs and the [ActionProvider](#) ([/reference/android/view/ActionProvider.html](#)) object.

## DEPENDENCIES AND PREREQUISITES

- Android 1.0 or higher (greater requirements where noted)
- Experience with [Intents and Intent Filters](#)

## Lessons

---

### **Sending Content to Other Apps**

Learn how to set up your application to be able to send text and binary data to other applications with intents.

### **Receiving Content from Other Apps**

Learn how to set up your application to receive text and binary data from intents.

### **Adding an Easy Share Action**

Learn how to add a "share" action item to your action bar.

# Sending Content to Other Apps

When you construct an intent, you must specify the action you want the intent to "trigger." Android defines several actions, including [ACTION\\_SEND](#) ([/reference/android/content/Intent.html#ACTION\\_SEND](#)) which, as you can probably guess, indicates that the intent is sending data from one activity to another, even across process boundaries. To send data to another activity, all you need to do is specify the data and its type, the system will identify compatible receiving activities and display them to the user (if there are multiple options) or immediately start the activity (if there is only one option). Similarly, you can advertise the data types that your activities support receiving from other applications by specifying them in your manifest.

Sending and receiving data between applications with intents is most commonly used for social sharing of content. Intents allow users to share information quickly and easily, using their favorite applications.

Note: The best way to add a share action item to an [ActionBar](#) ([/reference/android/app/ActionBar.html](#)) is to use [ShareActionProvider](#) ([/reference/android/widget/ShareActionProvider.html](#)), which became available in API level 14. [ShareActionProvider](#) ([/reference/android/widget/ShareActionProvider.html](#)) is discussed in the lesson about [Adding an Easy Share Action](#) ([shareaction.html](#)).

## Send Text Content

The most straightforward and common use of the [ACTION\\_SEND](#) ([/reference/android/content/Intent.html#ACTION\\_SEND](#)) action is sending text content from one activity to another. For example, the built-in Browser app can share the URL of the currently-displayed page as text with any application. This is useful for sharing an article or website with friends via email or social networking. Here is the code to implement this type of sharing:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(sendIntent);
```

If there's an installed application with a filter that matches [ACTION\\_SEND](#) ([/reference/android/content/Intent.html#ACTION\\_SEND](#)) and MIME type text/plain, the Android system will run it; if more than one application matches, the system displays a disambiguation dialog (a "chooser") that allows the user to choose an app. If you call [Intent.createChooser\(\)](#) ([/reference/android/content](#)

### THIS LESSON TEACHES YOU TO

1. [Send Text Content](#)
2. [Send Binary Content](#)
3. [Send Multiple Pieces of Content](#)

### YOU SHOULD ALSO READ

- [Intents and Intent Filters](#)

`/Intent.html#createChooser(android.content.Intent, java.lang.CharSequence)` for the intent, Android will always display the chooser. This has some advantages:

- Even if the user has previously selected a default action for this intent, the chooser will still be displayed.
- If no applications match, Android displays a system message.
- You can specify a title for the chooser dialog.

Here's the updated code:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text");
sendIntent.setType("text/plain");
startActivity(Intent.createChooser(sendIntent, getResources().getString(R.string.share_to_title))
```

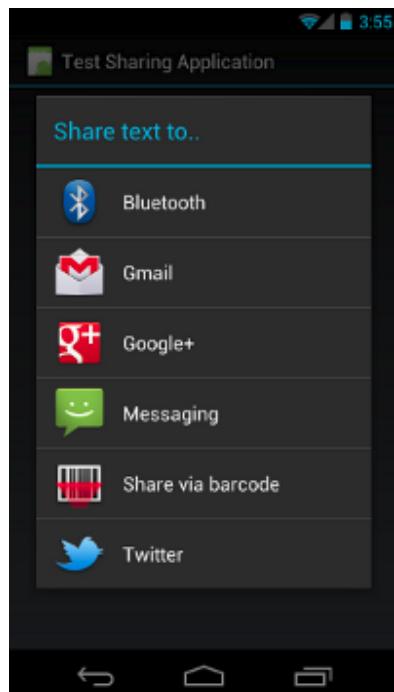


Figure 1. Screenshot of `ACTION_SEND` ([/reference/android/content/Intent.html#ACTION\\_SEND](#)) intent chooser on a handset.

The resulting dialog is shown in figure 1.

Optionally, you can set some standard extras for the intent:  
`EXTRA_EMAIL` ([/reference/android/content/Intent.html#EXTRA\\_EMAIL](#)),  
`EXTRA_CC` ([/reference/android/content/Intent.html#EXTRA\\_CC](#)),  
`EXTRA_BCC` ([/reference/android/content/Intent.html#EXTRA\\_BCC](#)),  
`EXTRA_SUBJECT` ([/reference/android/content/Intent.html#EXTRA SUBJECT](#)). However, if the receiving application is not designed to use them, nothing will happen. You can use custom extras as well, but there's no effect unless the receiving application understands them. Typically, you'd use custom extras defined by the receiving application itself.

Note: Some e-mail applications, such as Gmail, expect a `String[]` ([/reference/java/lang/String.html](#)) for extras like `EXTRA_EMAIL` ([/reference/android/content/Intent.html#EXTRA\\_EMAIL](#)) and `EXTRA_CC` ([/reference/android/content/Intent.html#EXTRA\\_CC](#)), use `putExtra(String, String[])` ([/reference/android/content/Intent.html#putExtra\(java.lang.String, java.lang.String\[\]\)](#)) to add these to your intent.

## Send Binary Content

Binary data is shared using the `ACTION_SEND` ([/reference/android/content/Intent.html#ACTION\\_SEND](#)) action combined with setting the appropriate MIME type and placing the URI to the data in an extra named `EXTRA_STREAM` ([/reference/android/content/Intent.html#EXTRA\\_STREAM](#)). This is commonly used to share an image but can be used to share any type of binary content:

```
Intent shareIntent = new Intent();
shareIntent.setAction(Intent.ACTION_SEND);
shareIntent.putExtra(Intent.EXTRA_STREAM, uriToImage);
shareIntent.setType("image/jpeg");
startActivity(Intent.createChooser(shareIntent, getResources().getString(R.string.share_to_title)))
```

Note the following:

- You can use a MIME type of "`*/*`", but this will only match activities that are able to handle generic data streams.
- The receiving application needs permission to access the data the `Uri` points to. There are a number of ways to handle this:
  - Write the data to a file on external/shared storage (such as the SD card), which all apps can read. Use `Uri.fromFile()` to create the `Uri` that can be passed to the share intent. However, keep in mind that not all applications process a `file://` style `Uri`.
  - Write the data to a file in your own application directory using `openFileOutput()` with mode `MODE_WORLD_READABLE` after which `getFileStreamPath()` can be used to return a `File`. As with the previous option, `Uri.fromFile()` will create a `file://` style `Uri` for your share intent.
  - Media files like images, videos and audio can be scanned and added to the system `MediaStore` using `scanFile()`. The `onScanCompleted()` callback returns a `content://` style `Uri` suitable for including in your share intent.
  - Images can be inserted into the system `MediaStore` using `insertImage()` which will return a `content://` style `Uri` suitable for including in a share intent.
- Store the data in your own `ContentProvider`, make sure that other apps have the correct permission to access your provider (or use `per-URI permissions`).

## Send Multiple Pieces of Content

To share multiple pieces of content, use the `ACTION_SEND_MULTIPLE` ([/reference/android/content/Intent.html#ACTION\\_SEND\\_MULTIPLE](/reference/android/content/Intent.html#ACTION_SEND_MULTIPLE)) action together with a list of URIs pointing to the content. The MIME type varies according to the mix of content you're sharing. For example, if you share 3 JPEG images, the type is still "`image/jpeg`". For a mixture of image types, it should be "`image/*`" to match an activity that handles any type of image. You should only use "`*/*`" if you're sharing out a wide variety of types. As previously stated, it's up to the receiving application to parse and process your data. Here's an example:

```
ArrayList<Uri> imageUrises = new ArrayList<Uri>();
imageUrises.add(imageUri1); // Add your image URIs here
imageUrises.add(imageUri2);

Intent shareIntent = new Intent();
shareIntent.setAction(Intent.ACTION_SEND_MULTIPLE);
shareIntent.putParcelableArrayListExtra(Intent.EXTRA_STREAM, imageUrises);
shareIntent.setType("image/*");
startActivity(Intent.createChooser(shareIntent, "Share images to..."));
```

As before, make sure the provided `URIs` (</reference/android/net/Uri.html>) point to data that a receiving application can access.

# Receiving Content from Other Apps

Just as your application can send data to other applications, so too can it easily receive data from applications. Think about how users interact with your application, and what data types you want to receive from other applications. For example, a social networking application would likely be interested in receiving text content, like an interesting web URL, from another app. The [Google+ Android application](#) (<https://play.google.com/store/apps/details?id=com.google.android.apps.plus>) accepts both text and single or multiple images. With this app, a user can easily start a new Google+ post with photos from the Android Gallery app.

## THIS LESSON TEACHES YOU TO

1. [Update Your Manifest](#)
2. [Handle the Incoming Content](#)

## YOU SHOULD ALSO READ

- [Intents and Intent Filters](#)

## Update Your Manifest

Intent filters inform the system what intents an application component is willing to accept. Similar to how you constructed an intent with action [ACTION\\_SEND](#) ([/reference/android/content/Intent.html#ACTION\\_SEND](#)) in the [Send Content to Other Apps Using Intents](#) ([/training/sharing/send.html](#)) lesson, you create intent filters in order to be able to receive intents with this action. You define an intent filter in your manifest, using the [<intent-filter>](#) ([/guide/components/intents-filters.html#ifs](#)) element. For example, if your application handles receiving text content, a single image of any type, or multiple images of any type, your manifest would look like:

```
<activity android:name=".ui.MyActivity" >
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SEND_MULTIPLE" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
    </intent-filter>
</activity>
```

Note: For more information on intent filters and intent resolution please read [Intents and Intent Resolution](#)

### Filters (/guide/components/intents-filters.html#ifs)

When another application tries to share any of these things by constructing an intent and passing it to `startActivity()` ([/reference/android/content/Context.html#startActivity\(android.content.Intent\)](#)), your application will be listed as an option in the intent chooser. If the user selects your application, the corresponding activity (`.ui.MyActivity` in the example above) will be started. It is then up to you to handle the content appropriately within your code and UI.

## Handle the Incoming Content

To handle the content delivered by an `Intent` ([/reference/android/content/Intent.html](#)), start by calling `getIntent()` ([/reference/android/content/Intent.html#getIntent\(java.lang.String\)](#)) to get `Intent` ([/reference/android/content/Intent.html](#)) object. Once you have the object, you can examine its contents to determine what to do next. Keep in mind that if this activity can be started from other parts of the system, such as the launcher, then you will need to take this into consideration when examining the intent.

```
void onCreate (Bundle savedInstanceState) {  
    ...  
    // Get intent, action and MIME type  
    Intent intent = getIntent();  
    String action = intent.getAction();  
    String type = intent.getType();  
  
    if (Intent.ACTION_SEND.equals(action) && type != null) {  
        if ("text/plain".equals(type)) {  
            handleSendText(intent); // Handle text being sent  
        } else if (type.startsWith("image/")) {  
            handleSendImage(intent); // Handle single image being sent  
        }  
    } else if (Intent.ACTION_SEND_MULTIPLE.equals(action) && type != null) {  
        if (type.startsWith("image/")) {  
            handleSendMultipleImages(intent); // Handle multiple images being sent  
        }  
    } else {  
        // Handle other intents, such as being started from the home screen  
    }  
    ...  
}  
  
void handleSendText(Intent intent) {  
    String sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);  
    if (sharedText != null) {  
        // Update UI to reflect text being shared  
    }  
}
```

```
void handleSendImage(Intent intent) {  
    Uri imageUri = (Uri) intent.getParcelableExtra(Intent.EXTRA_STREAM);  
    if (imageUri != null) {  
        // Update UI to reflect image being shared  
    }  
}  
  
void handleSendMultipleImages(Intent intent) {  
    ArrayList<Uri> imageUrils = intent.getParcelableArrayListExtra(Intent.EXTRA_ST  
    if (imageUrils != null) {  
        // Update UI to reflect multiple images being shared  
    }  
}
```

Caution: Take extra care to check the incoming data, you never know what some other application may send you. For example, the wrong MIME type might be set, or the image being sent might be extremely large. Also, remember to process binary data in a separate thread rather than the main ("UI") thread.

Updating the UI can be as simple as populating an [EditText](#) ([/reference/android/widget/EditText.html](#)), or it can be more complicated like applying an interesting photo filter to an image. It's really specific to your application what happens next.

# Adding an Easy Share Action

Implementing an effective and user friendly share action in your [ActionBar](#) ([/reference/android/app/ActionBar.html](#)) is made even easier with the introduction of [ActionProvider](#) ([/reference/android/view/ActionProvider.html](#)) in Android 4.0 (API Level 14). An [ActionProvider](#) ([/reference/android/view/ActionProvider.html](#)), once attached to a menu item in the action bar, handles both the appearance and behavior of that item. In the case of [ShareActionProvider](#) ([/reference/android/widget/ShareActionProvider.html](#)), you provide a share intent and it does the rest.

Note: [ShareActionProvider](#) ([/reference/android/widget/ShareActionProvider.html](#)) is available starting with API Level 14 and higher.

## Update Menu Declarations

To get started with [ShareActionProviders](#) ([/reference/android/widget/ShareActionProvider.html](#)), define the `android:actionProviderClass` attribute for the corresponding `<item>` in your [menu resource](#) ([/guide/topics/resources/menu-resource.html](#)) file:

```
<menu xmlns:android="http://schemas.android.com/apk/res
    <item android:id="@+id/menu_item_share"
        android:showAsAction="ifRoom"
        android:title="Share"
        android:actionProviderClass="android.widget.ShareActionProvider"/>
    ...
</menu>
```

This delegates responsibility for the item's appearance and function to [ShareActionProvider](#) ([/reference/android/widget/ShareActionProvider.html](#)). However, you will need to tell the provider what you would like to share.

### THIS LESSON TEACHES YOU TO

1. [Update Menu Declarations](#)
2. [Set the Share Intent](#)

### YOU SHOULD ALSO READ

- [Action Bar](#)

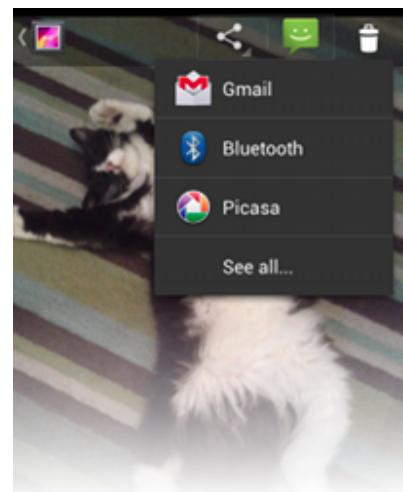


Figure 1. The [ShareActionProvider](#) ([/reference/android/widget/ShareActionProvider.html](#)) in the Gallery app.

## Set the Share Intent

In order for [ShareActionProvider](#) ([/reference/android/widget/ShareActionProvider.html](#)) to function, you must provide it a share intent. This share intent should be the same as described in the [Sending Content to Other Apps](#) ([/training/sharing/send.html](#)) lesson, with action [ACTION\\_SEND](#) ([/reference/android/content/Intent.html#ACTION\\_SEND](#)) and additional data set via extras like [EXTRA\\_TEXT](#).

([/reference/android/content/Intent.html#EXTRA\\_TEXT](#)) and [EXTRA\\_STREAM](#) ([/reference/android/content/Intent.html#EXTRA\\_STREAM](#)). To assign a share intent, first find the corresponding [MenuItem](#) ([/reference/android/view/MenuItem.html](#)) while inflating your menu resource in your [Activity](#) ([/reference/android/app/Activity.html](#)) or [Fragment](#) ([/reference/android/app/Fragment.html](#)). Next, call [MenuItem.getActionProvider\(\)](#) ([/reference/android/view/MenuItem.html#getActionProvider\(\)](#)) to retrieve an instance of [ShareActionProvider](#) ([/reference/android/widget/ShareActionProvider.html](#)). Use [setShareIntent\(\)](#) ([/reference/android/widget/ShareActionProvider.html#setShareIntent\(android.content.Intent\)](#)) to update the share intent associated with that action item. Here's an example:

```
private ShareActionProvider mShareActionProvider;  
...  
  
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    // Inflate menu resource file.  
    getMenuInflater().inflate(R.menu.share_menu, menu);  
  
    // Locate MenuItem with ShareActionProvider  
    MenuItem item = menu.findItem(R.id.menu_item_share);  
  
    // Fetch and store ShareActionProvider  
    mShareActionProvider = (ShareActionProvider) item.getActionProvider();  
  
    // Return true to display menu  
    return true;  
}  
  
// Call to update the share intent  
private void setShareIntent(Intent shareIntent) {  
    if (mShareActionProvider != null) {  
        mShareActionProvider.setShareIntent(shareIntent);  
    }  
}
```

You may only need to set the share intent once during the creation of your menus, or you may want to set it and then update it as the UI changes. For example, when you view photos full screen in the Gallery app, the sharing intent changes as you flip between photos.

For further discussion about the [ShareActionProvider](#) ([/reference/android/widget/ShareActionProvider.html](#)) object, see the [Action Bar](#) ([/guide/topics/ui/actionbar.html#ActionProvider](#)) guide.

# Capturing Photos

The world was a dismal and featureless place before rich media became prevalent. Remember Gopher? We don't, either. For your app to become part of your users' lives, give them a way to put their lives into it. Using the on-board cameras, your application can enable users to augment what they see around them, make unique avatars, look for zombies around the corner, or simply share their experiences.

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.

## Lessons

---

### Taking Photos Simply

Leverage other applications and capture photos with just a few lines of code.

### Recording Videos Simply

Leverage other applications and record videos with just a few lines of code.

### Controlling the Camera

Control the camera hardware directly and implement your own camera application.

### DEPENDENCIES AND PREREQUISITES

---

- Android 1.5 (API level 3) or higher
- A device with a camera

### YOU SHOULD ALSO READ

---

- [Camera](#)
- [Activities](#)

### TRY IT OUT

---

[Download the sample](#)

PhotoIntentActivity.zip

# Taking Photos Simply

This lesson explains how to capture photos using an existing camera application.

Suppose you are implementing a crowd-sourced weather service that makes a global weather map by blending together pictures of the sky taken by devices running your client app. Integrating photos is only a small part of your application. You want to take photos with minimal fuss, not reinvent the camera. Happily, most Android-powered devices already have at least one camera application installed. In this lesson, you learn how to make it take a picture for you.

## Request Camera Permission

If an essential function of your application is taking pictures, then restrict its visibility on Google Play to devices that have a camera. To advertise that your application depends on having a camera, put a `<uses-feature>` (</guide/topics/manifest/uses-feature-element.html>) tag in your manifest file:

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera" />
    ...
</manifest ... >
```

If your application uses, but does not require a camera in order to function, add `android:required="false"` to the tag. In doing so, Google Play will allow devices without a camera to download your application. It's then your responsibility to check for the availability of the camera at runtime by calling `hasSystemFeature(PackageManager.FEATURE_CAMERA)` ([/reference/android/content/pm/PackageManager.html#hasSystemFeature\(java.lang.String\)](/reference/android/content/pm/PackageManager.html#hasSystemFeature(java.lang.String))). If a camera is not available, you should then disable your camera features.

## Take a Photo with the Camera App

The Android way of delegating actions to other applications is to invoke an `Intent` (</reference/android/content/Intent.html>) that describes what you want done. This process involves three pieces: The `Intent` (</reference/android/content/Intent.html>) itself, a call to start the external `Activity` (</reference/android/app/Activity.html>), and some code to handle the image data when focus returns to your activity.

### THIS LESSON TEACHES YOU TO

1. [Request Camera Permission](#)
2. [Take a Photo with the Camera App](#)
3. [View the Photo](#)
4. [Save the Photo](#)
5. [Add the Photo to a Gallery](#)
6. [Decode a Scaled Image](#)

### YOU SHOULD ALSO READ

- [Camera](#)
- [Intents and Intent Filters](#)

### TRY IT OUT

[Download the sample](#)

PhotoIntentActivity.zip

Here's a function that invokes an intent to capture a photo.

```
private void dispatchTakePictureIntent(int requestCode) {  
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);  
    startActivityForResult(takePictureIntent, requestCode);  
}
```

Congratulations: with this code, your application has gained the ability to make another camera application do its bidding! Of course, if no compatible application is ready to catch the intent, then your app will fall down like a botched stage dive. Here is a function to check whether an app can handle your intent:

```
public static boolean isIntentAvailable(Context context, String action) {  
    final PackageManager packageManager = context.getPackageManager();  
    final Intent intent = new Intent(action);  
    List<ResolveInfo> list =  
        packageManager.queryIntentActivities(intent, PackageManager.MATCH_DEFAULT_ONLY);  
    return list.size() > 0;  
}
```

## View the Photo

If the simple feat of taking a photo is not the culmination of your app's ambition, then you probably want to get the image back from the camera application and do something with it.

The Android Camera application encodes the photo in the return [Intent](#) ([/reference/android/content/Intent.html](#)) delivered to [onActivityResult\(\)](#) ([/reference/android/app/Activity.html#onActivityResult\(int, int, android.content.Intent\)](#)) as a small [Bitmap](#) ([/reference/android/graphics/Bitmap.html](#)) in the extras, under the key "data". The following code retrieves this image and displays it in an [ImageView](#) ([/reference/android/widget/ImageView.html](#)).

```
private void handleSmallCameraPhoto(Intent intent) {  
    Bundle extras = intent.getExtras();  
    mImageBitmap = (Bitmap) extras.get("data");  
    mImageView.setImageBitmap(mImageBitmap);  
}
```

Note: This thumbnail image from "data" might be good for an icon, but not a lot more. Dealing with a full-sized image takes a bit more work.

## Save the Photo

The Android Camera application saves a full-size photo if you give it a file to save into. You must provide a path that includes the storage volume, folder, and file name.

There is an easy way to get the path for photos, but it works only on Android 2.2 (API level 8) and later:

```
storageDir = new File(  
    Environment.getExternalStoragePublicDirectory(  
        Environment.DIRECTORY_PICTURES  
)  
,  
    getAlbumName()  
);
```

For earlier API levels, you have to provide the name of the photo directory yourself.

```
storageDir = new File (  
    Environment.getExternalStorageDirectory()  
    + PICTURES_DIR  
    + getAlbumName()  
);
```

Note: The path component PICTURES\_DIR is just Pictures/, the standard location for shared photos on the external/shared storage.

## Set the file name

As shown in the previous section, the file location for an image should be driven by the device environment. What you need to do yourself is choose a collision-resistant file-naming scheme. You may wish also to save the path in a member variable for later use. Here's an example solution:

```
private File createImageFile() throws IOException {  
    // Create an image file name  
    String timeStamp =  
        new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());  
    String imageFileName = JPEG_FILE_PREFIX + timeStamp + "_";  
    File image = File.createTempFile(  
        imageFileName,  
        JPEG_FILE_SUFFIX,  
        getAlbumDir()  
    );  
    mCurrentPhotoPath = image.getAbsolutePath();  
    return image;  
}
```

## Append the file name onto the Intent

Once you have a place to save your image, pass that location to the camera application via the

[Intent \(/reference/android/content/Intent.html\)](#).

```
File f = createImageFile();
takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(f));
```

## Add the Photo to a Gallery

When you create a photo through an intent, you should know where your image is located, because you said where to save it in the first place. For everyone else, perhaps the easiest way to make your photo accessible is to make it accessible from the system's Media Provider.

The following example method demonstrates how to invoke the system's media scanner to add your photo to the Media Provider's database, making it available in the Android Gallery application and to other apps.

```
private void galleryAddPic() {
    Intent mediaScanIntent = new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
    File f = new File(mCurrentPhotoPath);
    Uri contentUri = Uri.fromFile(f);
    mediaScanIntent.setData(contentUri);
    this.sendBroadcast(mediaScanIntent);
}
```

## Decode a Scaled Image

Managing multiple full-sized images can be tricky with limited memory. If you find your application running out of memory after displaying just a few images, you can dramatically reduce the amount of dynamic heap used by expanding the JPEG into a memory array that's already scaled to match the size of the destination view. The following example method demonstrates this technique.

```
private void setPic() {
    // Get the dimensions of the View
    int targetW = mImageView.getWidth();
    int targetH = mImageView.getHeight();

    // Get the dimensions of the bitmap
    BitmapFactory.Options bmOptions = new BitmapFactory.Options();
    bmOptions.inJustDecodeBounds = true;
    BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
    int photoW = bmOptions.outWidth;
    int photoH = bmOptions.outHeight;

    // Determine how much to scale down the image
```

```
int scaleFactor = Math.min(photoW/targetW, photoH/targetH);

// Decode the image file into a Bitmap sized to fill the View
bmOptions.inJustDecodeBounds = false;
bmOptions.inSampleSize = scaleFactor;
bmOptions.inPurgeable = true;

Bitmap bitmap = BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
mImageView.setImageBitmap(bitmap);
}
```

# Recording Videos Simply

This lesson explains how to capture video using existing camera applications.

Your application has a job to do, and integrating videos is only a small part of it. You want to take videos with minimal fuss, and not reinvent the camcorder. Happily, most Android-powered devices already have a camera application that records video. In this lesson, you make it do this for you.

## Request Camera Permission

To advertise that your application depends on having a camera, put a `<uses-feature>` tag in the manifest file:

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera" ... />
</manifest ... >
```

If your application uses, but does not require a camera in order to function, add `android:required="false"` to the tag. In doing so, Google Play will allow devices without a camera to download your application. It's then your responsibility to check for the availability of the camera at runtime by calling `hasSystemFeature(PackageManager.FEATURE_CAMERA)` ([/reference/android/content/pm/PackageManager.html#hasSystemFeature\(java.lang.String\)](#)). If a camera is not available, you should then disable your camera features.

## Record a Video with a Camera App

The Android way of delegating actions to other applications is to invoke an [Intent](#) ([/reference/android/content/Intent.html](#)) that describes what you want done. This involves three pieces: the [Intent](#) ([/reference/android/content/Intent.html](#)) itself, a call to start the external [Activity](#) ([/reference/android/app/Activity.html](#)), and some code to handle the video when focus returns to your activity.

Here's a function that invokes an intent to capture video.

```
private void dispatchTakeVideoIntent() {
    Intent takeVideoIntent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
    startActivityForResult(takeVideoIntent, ACTION_TAKE_VIDEO);
}
```

### THIS LESSON TEACHES YOU TO

1. [Request Camera Permission](#)
2. [Record a Video with a Camera App](#)
3. [View the Video](#)

### YOU SHOULD ALSO READ

- [Camera](#)
- [Intents and Intent Filters](#)

### TRY IT OUT

[Download the sample](#)

PhotoIntentActivity.zip

It's a good idea to make sure an app exists to handle your intent before invoking it. Here's a function that checks for apps that can handle your intent:

```
public static boolean isIntentAvailable(Context context, String action) {  
    final PackageManager packageManager = context.getPackageManager();  
    final Intent intent = new Intent(action);  
    List<ResolveInfo> list =  
        packageManager.queryIntentActivities(intent,  
            PackageManager.MATCH_DEFAULT_ONLY);  
    return list.size() > 0;  
}
```

## View the Video

---

The Android Camera application returns the video in the [Intent](#) ([/reference/android/content/Intent.html](#)) delivered to [onActivityResult\(\)](#) ([/reference/android/app/Activity.html#onActivityResult\(int, int, android.content.Intent\)](#)) as a [Uri](#) ([/reference/android/net/Uri.html](#)) pointing to the video location in storage. The following code retrieves this video and displays it in a [VideoView](#) ([/reference/android/widget/VideoView.html](#)).

```
private void handleCameraVideo(Intent intent) {  
    mVideoUri = intent.getData();  
    mVideoView.setVideoURI(mVideoUri);  
}
```

# Controlling the Camera

In this lesson, we discuss how to control the camera hardware directly using the framework APIs.

Directly controlling a device camera requires a lot more code than requesting pictures or videos from existing camera applications. However, if you want to build a specialized camera application or something fully integrated in your app UI, this lesson shows you how.

## Open the Camera Object

Getting an instance of the [Camera](#) ([/reference/android/hardware/Camera.html](#)) object is the first step in the process of directly controlling the camera. As Android's own Camera application does, the recommended way to access the camera is to open [Camera](#) ([/reference/android/hardware/Camera.html](#)) on a separate thread that's launched from [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)). This approach is a good idea since it can take a while and might bog down the UI thread. In a more basic implementation, opening the camera can be deferred to the [onResume\(\)](#) ([/reference/android/app/Activity.html#onResume\(\)](#)) method to facilitate code reuse and keep the flow of control simple.

Calling [Camera.open\(\)](#) ([/reference/android/hardware/Camera.html#open\(\)](#)) throws an exception if the camera is already in use by another application, so we wrap it in a try block.

```
private boolean safeCameraOpen(int id) {
    boolean qOpened = false;

    try {
        releaseCameraAndPreview();
        mCamera = Camera.open(id);
        qOpened = (mCamera != null);
    } catch (Exception e) {
        Log.e(getString(R.string.app_name), "failed to open Camera");
        e.printStackTrace();
    }

    return qOpened;
}

private void releaseCameraAndPreview() {
    mPreview.setCamera(null);
    if (mCamera != null) {
        mCamera.release();
    }
}
```

### THIS LESSON TEACHES YOU TO

1. [Open the Camera Object](#)
2. [Create the Camera Preview](#)
3. [Modify Camera Settings](#)
4. [Set the Preview Orientation](#)
5. [Take a Picture](#)
6. [Restart the Preview](#)
7. [Stop the Preview and Release the Camera](#)

### YOU SHOULD ALSO READ

- [Building a Camera App](#)

```
mCamera = null;  
}  
}
```

Since API level 9, the camera framework supports multiple cameras. If you use the legacy API and call [open\(\)](#) ([/reference/android/hardware/Camera.html#open\(\)](#)) without an argument, you get the first rear-facing camera.

## Create the Camera Preview

Taking a picture usually requires that your users see a preview of their subject before clicking the shutter. To do so, you can use a [SurfaceView](#) ([/reference/android/view/SurfaceView.html](#)) to draw previews of what the camera sensor is picking up.

### Preview Class

To get started with displaying a preview, you need preview class. The preview requires an implementation of the `android.view.SurfaceHolder.Callback` interface, which is used to pass image data from the camera hardware to the application.

```
class Preview extends ViewGroup implements SurfaceHolder.Callback {  
  
    SurfaceView mSurfaceView;  
    SurfaceHolder mHolder;  
  
    Preview(Context context) {  
        super(context);  
  
        mSurfaceView = new SurfaceView(context);  
        addView(mSurfaceView);  
  
        // Install a SurfaceHolder.Callback so we get notified when the  
        // underlying surface is created and destroyed.  
        mHolder = mSurfaceView.getHolder();  
        mHolder.addCallback(this);  
        mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);  
    }  
    ...  
}
```

The preview class must be passed to the [Camera](#) ([/reference/android/hardware/Camera.html](#)) object before the live image preview can be started, as shown in the next section.

### Set and Start the Preview

A camera instance and its related preview must be created in a specific order, with the camera object being first. In the snippet below, the process of initializing the camera is encapsulated so

that `Camera.startPreview()` ([/reference/android/hardware/Camera.html#startPreview\(\)](#)) is called by the `setCamera()` method, whenever the user does something to change the camera. The preview must also be restarted in the preview class `surfaceChanged()` callback method.

```
public void setCamera(Camera camera) {
    if (mCamera == camera) { return; }

    stopPreviewAndFreeCamera();

    mCamera = camera;

    if (mCamera != null) {
        List<Size> localSizes = mCamera.getParameters().getSupportedPreviewSizes();
        mSupportedPreviewSizes = localSizes;
        requestLayout();

        try {
            mCamera.setPreviewDisplay(mHolder);
        } catch (IOException e) {
            e.printStackTrace();
        }

        /*
         Important: Call startPreview() to start updating the preview surface. Preview
         must be started before you can take a picture.
        */
        mCamera.startPreview();
    }
}
```

## Modify Camera Settings

Camera settings change the way that the camera takes pictures, from the zoom level to exposure compensation. This example changes only the preview size; see the source code of the Camera application for many more.

```
public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
    // Now that the size is known, set up the camera parameters and begin
    // the preview.
    Camera.Parameters parameters = mCamera.getParameters();
    parameters.setPreviewSize(mPreviewSize.width, mPreviewSize.height);
    requestLayout();
    mCamera.setParameters(parameters);

    /*
     Important: Call startPreview() to start updating the preview surface. Preview
     must be started before you can take a picture.
    */
}
```

```
    started before you can take a picture.  
*/  
mCamera.startPreview();  
}
```

## Set the Preview Orientation

Most camera applications lock the display into landscape mode because that is the natural orientation of the camera sensor. This setting does not prevent you from taking portrait-mode photos, because the orientation of the device is recorded in the EXIF header. The [setCameraDisplayOrientation\(\) \(/reference/android/hardware/Camera.html#setDisplayOrientation\(int\)\)](#) method lets you change how the preview is displayed without affecting how the image is recorded. However, in Android prior to API level 14, you must stop your preview before changing the orientation and then restart it.

## Take a Picture

Use the [Camera.takePicture\(\) \(/reference/android/hardware/Camera.html#takePicture\(android.hardware.Camera.ShutterCallback, android.hardware.Camera.PictureCallback, android.hardware.Camera.PictureCallback\)\)](#) method to take a picture once the preview is started. You can create [Camera.PictureCallback \(/reference/android/hardware/Camera.PictureCallback.html\)](#) and [Camera.ShutterCallback \(/reference/android/hardware/Camera.ShutterCallback.html\)](#) objects and pass them into [Camera.takePicture\(\) \(/reference/android/hardware/Camera.html#takePicture\(android.hardware.Camera.ShutterCallback, android.hardware.Camera.PictureCallback, android.hardware.Camera.PictureCallback\)\)](#).

If you want to grab images continuously, you can create a [Camera.PreviewCallback \(/reference/android/hardware/Camera.PreviewCallback.html\)](#) that implements [onPreviewFrame\(\) \(/reference/android/hardware/Camera.PreviewCallback.html#onPreviewFrame\(byte\[\], android.hardware.Camera\)\)](#). For something in between, you can capture only selected preview frames, or set up a delayed action to call [takePicture\(\) \(/reference/android/hardware/Camera.html#takePicture\(android.hardware.Camera.ShutterCallback, android.hardware.Camera.PictureCallback, android.hardware.Camera.PictureCallback\)\)](#).

## Restart the Preview

After a picture is taken, you must restart the preview before the user can take another picture. In this example, the restart is done by overloading the shutter button.

```
@Override  
public void onClick(View v) {  
    switch(mPreviewState) {  
        case K_STATE_FROZEN:  
            mCamera.startPreview();
```

```
mPreviewState = K_STATE_PREVIEW;
break;

default:
    mCamera.takePicture( null, rawCallback, null);
    mPreviewState = K_STATE_BUSY;
} // switch
shutterBtnConfig();
}
```

## Stop the Preview and Release the Camera

Once your application is done using the camera, it's time to clean up. In particular, you must release the [Camera \(/reference/android/hardware/Camera.html\)](#) object, or you risk crashing other applications, including new instances of your own application.

When should you stop the preview and release the camera? Well, having your preview surface destroyed is a pretty good hint that it's time to stop the preview and release the camera, as shown in these methods from the Preview class.

```
public void surfaceDestroyed(SurfaceHolder holder) {
    // Surface will be destroyed when we return, so stop the preview.
    if (mCamera != null) {
        /*
         * Call stopPreview() to stop updating the preview surface.
         */
        mCamera.stopPreview();
    }
}

/**
 * When this function returns, mCamera will be null.
 */
private void stopPreviewAndFreeCamera() {

    if (mCamera != null) {
        /*
         * Call stopPreview() to stop updating the preview surface.
         */
        mCamera.stopPreview();

        /*
         * Important: Call release() to release the camera for use by other applications.
         * Applications should release the camera immediately in onPause() (and resume()).
         */
        mCamera.release();
    }
}
```

```
    mCamera = null;  
}  
}
```

Earlier in the lesson, this procedure was also part of the `setCamera()` method, so initializing a camera always begins with stopping the preview.

# Maintaining Multiple APKs

Multiple APK support is a feature of Google Play that allows you to publish multiple APKs under the same application listing. Each APK is a complete instance of your application, optimized to target specific device configurations. Each APK can target a specific set of GL textures, API levels, screen sizes, or some combination thereof.

This class shows you how to write your multiple APK application using any one of these configuration variables. Each lesson covers basics about how to organize your codebase and target the right devices, as well as the smart way to avoid pitfalls such as unnecessary redundancy across your codebase, and making mistakes in your manifest that could render an APK invisible to all devices on Google Play. By going through any of these lessons, you'll know how to develop multiple APKs the smart way, make sure they're targeting the devices you want them to, and know how to catch mistakes *before* your app goes live.

## Lessons

---

### **Creating Multiple APKs for Different API Levels**

Learn how to target different versions of the Android platform using multiple APKs. Also learn how to organize your codebase, what to do with your manifest, and how to investigate your APK configuration using the aapt tool before pushing live.

### **Creating Multiple APKs for Different Screen Sizes**

Learn how to target Android devices by screen size using multiple APKs. Also learn how to organize your codebase, what to do with your manifest, and how to investigate your APK configuration using the aapt tool before pushing live.

### **Creating Multiple APKs for Different GL Textures**

Learn how to target Android devices based on their support for GL texture. Also learn how to organize your codebase, what to do with your manifest, and how to investigate your APK configuration using the aapt tool before pushing live.

### **Creating Multiple APKs with 2+ Dimensions**

Learn how to target different Android devices based on more than one configuration variable (screen size, API version, GL texture). Examples in the lesson target using a combination of API level and screen size. Also learn how to organize your codebase, what to do with your manifest, and how to investigate your APK configuration using the aapt tool before pushing live.

### **DEPENDENCIES AND PREREQUISITES**

---

- Android 1.0 and higher
- You must have an [Google Play](#) publisher account

### **YOU SHOULD ALSO READ**

---

- [Multiple APK Support](#)

# Creating Multiple APKs for Different API Levels

When developing your Android application to take advantage of multiple APKs on Google Play, it's important to adopt some good practices from the get-go, and prevent unnecessary headaches further into the development process. This lesson shows you how to create multiple APKs of your app, each covering a slightly different range of API levels. You will also gain some tools necessary to make maintaining a multiple APK codebase as painless as possible.

## Confirm You Need Multiple APKs

When trying to create an application that works across multiple generations of the Android platform, naturally you want your application to take advantage of new features on new devices, without sacrificing backwards compatibility. It may seem at the outset as though multiple APK support is the best solution, but this often isn't the case. The [Using Single APK Instead \(/guide/google/play/publishing/multiple-apks.html#ApiLevelOptions\)](#) section of the multiple APK developer guide includes some useful information on how to accomplish this with a single APK, including use of our support library. You can also learn how to write code that runs only at certain API levels in a single APK, without resorting to computationally expensive techniques like reflection from [this article \(http://android-developers.blogspot.com/2010/07/how-to-have-your-cupcake-and-eat-it-too.html\)](#).

If you can manage it, confining your application to a single APK has several advantages, including:

- Publishing and testing are easier
- There's only one codebase to maintain
- Your application can adapt to device configuration changes
- App restore across devices just works
- You don't have to worry about market preference, behavior from "upgrades" from one APK to the next, or which APK goes with which class of devices

The rest of this lesson assumes that you've researched the topic, studiously absorbed the material in the resources linked, and determined that multiple APKs are the right path for your application.

## Chart Your Requirements

Start off by creating a simple chart to quickly determine how many APKs you need, and what API range each APK covers. For handy reference, the [Platform Versions \(/about/dashboards/index.html\)](#) page of the Android Developer website provides data about the relative number of active devices running a given version of the Android platform. Also, although it sounds easy at first, keeping track of which set of API levels each APK is going to target gets difficult rather quickly, especially if

### THIS LESSON TEACHES YOU TO

1. [Confirm You Need Multiple APKs](#)
2. [Chart Your Requirements](#)
3. [Put All Common Code and Resources in a Library Project](#)
4. [Create New APK Projects](#)
5. [Adjust the Manifests](#)
6. [Go Over Pre-launch Checklist](#)

### YOU SHOULD ALSO READ

- [Multiple APK Support](#)
- [How to have your \(Cup\)cake and eat it too](#)

there's going to be some overlap (there often is). Fortunately, it's easy to chart out your requirements quickly, easily, and have an easy reference for later.

In order to create your multiple APK chart, start out with a row of cells representing the various API levels of the Android platform. Throw an extra cell at the end to represent future versions of Android.

3	4	5	6	7	8	9	10	11	12	13	+
---	---	---	---	---	---	---	----	----	----	----	---

Now just color in the chart such that each color represents an APK. Here's one example of how you might apply each APK to a certain range of API levels.

3	4	5	6	7	8	9	10	11	12	13	+
---	---	---	---	---	---	---	----	----	----	----	---

Once you've created this chart, distribute it to your team. Team communication on your project just got immediately simpler, since instead of asking "How's the APK for API levels 3 to 6, er, you know, the Android 1.x one. How's that coming along?" You can simply say "How's the Blue APK coming along?"

## Put All Common Code and Resources in a Library Project

---

Whether you're modifying an existing Android application or starting one from scratch, this is the first thing that you should do to the codebase, and by far the most important. Everything that goes into the library project only needs to be updated once (think language-localized strings, color themes, bugs fixed in shared code), which improves your development time and reduces the likelihood of mistakes that could have been easily avoided.

Note: While the implementation details of how to create and include library projects are beyond the scope of this lesson, you can get up to speed quickly on their creation at the following links:

- [Setting up a library project \(Eclipse\)](#)
- [Setting up a library project \(Command line\)](#)

If you're converting an existing application to use multiple APK support, scour your codebase for every localized string file, list of values, theme colors, menu icons and layout that isn't going to change across APKs, and put it all in the library project. Code that isn't going to change much should also go in the library project. You'll likely find yourself extending these classes to add a method or two from APK to APK.

If, on the other hand, you're creating the application from scratch, try as much as possible to write code in the library project *first*, then only move it down to an individual APK if necessary. This is much easier to manage in the long run than adding it to one, then another, then another, then months later trying to figure out whether this blob can be moved up to the library section without screwing anything up.

## Create New APK Projects

---

There should be a separate Android project for each APK you're going to release. For easy organization, place the library project and all related APK projects under the same parent folder. Also remember that each APK needs to have the same package name, although they don't necessarily need to share the package name with the library. If you were to have 3 APKs following the scheme described earlier, your root directory might look like this:

```
alexlucas:~/code/multi-apks-root$ ls
foo-blue
foo-green
foo-lib
foo-red
```

Once the projects are created, add the library project as a reference to each APK project. If possible, define your starting Activity in the library project, and extend that Activity in your APK project. Having a starting activity defined in the library project gives you a chance to put all your application initialization in one place, so that each individual APK doesn't have to re-implement "universal" tasks like initializing Analytics, running licensing checks, and any other initialization procedures that don't change much from APK to APK.

## Adjust the Manifests

---

When a user downloads an application which uses multiple APKs through Google Play, the correct APK to use is chosen using two simple rules:

- The manifest has to show that particular APK is eligible
- Of the eligible APKs, highest version number wins

By way of example, let's take the set of multiple APKs described earlier, and assume that we haven't set a max API level for any of the APKs. Taken individually, the possible range of each APK would look like this:

3	4	5	6	7	8	9	10	11	12	13	+
3	4	5	6	7	8	9	10	11	12	13	+
3	4	5	6	7	8	9	10	11	12	13	+

Because it is required that an APK with a higher minSdkVersion also have a higher version code, we know that in terms of versionCode values, red  $\geq$  green  $\geq$  blue. Therefore we can effectively collapse the chart to look like this:

3	4	5	6	7	8	9	10	11	12	13	+
---	---	---	---	---	---	---	----	----	----	----	---

Now, let's further assume that the Red APK has some requirement on it that the other two don't. [Filters on Google Play \(/guide/google/play/filters.html\)](#) page of the Android Developer guide has a whole list of possible culprits. For the sake of example, let's assume that red requires a front-facing camera. In fact, the entire point of the red APK is to combine the front-facing camera with sweet new functionality that was added in API 11. But, it turns out, not all devices that support API 11

even HAVE front-facing cameras! The horror!

Fortunately, if a user is browsing Google Play from one such device, Google Play will look at the manifest, see that Red lists the front-facing camera as a requirement, and quietly ignore it, having determined that Red and that device are not a match made in digital heaven. It will then see that Green is not only forward-compatible with devices with API 11 (since no maxSdkVersion was defined), but also doesn't care whether or not there's a front-facing camera! The app can still be downloaded from Google Play by the user, because despite the whole front-camera mishap, there was still an APK that supported that particular API level.

In order to keep all your APKs on separate "tracks", it's important to have a good version code scheme. The recommended one can be found on the [Version Codes \(/guide/google/play/publishing/multiple-apks.html#VersionCodes\)](#) area of our developer guide. Since the example set of APKs is only dealing with one of 3 possible dimensions, it would be sufficient to separate each APK by 1000, set the first couple digits to the minSdkVersion for that particular APK, and increment from there. This might look like:

Blue: 03001, 03002, 03003, 03004...

Green: 07001, 07002, 07003, 07004...

Red: 11001, 11002, 11003, 11004...

Putting this all together, your Android Manifests would likely look something like the following:

Blue:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="03001" android:versionName="1.0" package="com.example.fo
    <uses-sdk android:minSdkVersion="3" />
    ...

```

Green:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="07001" android:versionName="1.0" package="com.example.fo
    <uses-sdk android:minSdkVersion="7" />
    ...

```

Red:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="11001" android:versionName="1.0" package="com.example.fo
    <uses-sdk android:minSdkVersion="11" />
    ...

```

## Go Over Pre-launch Checklist

---

Before uploading to Google Play, double-check the following items. Remember that these are specifically relevant to multiple APKs, and in no way represent a complete checklist for all applications being uploaded to Google Play.

- All APKs must have the same package name
- All APKs must be signed with the same certificate
- If the APKs overlap in platform version, the one with the higher minSdkVersion must have a higher version code
- Double check your manifest filters for conflicting information (an APK that only supports cupcake on XLARGE screens isn't going to be seen by anybody)
- Each APK's manifest must be unique across at least one of supported screen, openGL texture, or platform version
- Try to test each APK on at least one device. Barring that, you have one of the most customizable device emulators in the business sitting on your development machine. Go nuts!

It's also worth inspecting the compiled APK before pushing to market, to make sure there aren't any surprises that could hide your application on Google Play. This is actually quite simple using the "aapt" tool. Aapt (the Android Asset Packaging Tool) is part of the build process for creating and packaging your Android applications, and is also a very handy tool for inspecting them.

```
>aapt dump badging
package: name='com.example.hello' versionCode='1' versionName='1.0'
sdkVersion:'11'
uses-permission:'android.permission.SEND_SMS'
application-label:'Hello'
application-icon-120:'res/drawable-ldpi/icon.png'
application-icon-160:'res/drawable-mdpi/icon.png'
application-icon-240:'res/drawable-hdpi/icon.png'
application: label='Hello' icon='res/drawable-mdpi/icon.png'
launchable-activity: name='com.example.hello.HelloActivity' label='Hello' icon=''
uses-feature:'android.hardware.telephony'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '120' '160' '240'
```

When you examine aapt output, be sure to check that you don't have conflicting values for supports-screens and compatible-screens, and that you don't have unintended "uses-feature" values that were added as a result of permissions you set in the manifest. In the example above, the APK won't be visible to very many devices.

Why? By adding the required permission SEND\_SMS, the feature requirement of android.hardware.telephony was implicitly added. Since API 11 is Honeycomb (the version of Android optimized specifically for tablets), and no Honeycomb devices have telephony hardware in them, Google Play will filter out this APK in all cases, until future devices come along which are higher in API level AND possess telephony hardware.

Fortunately this is easily fixed by adding the following to your manifest:

```
<uses-feature android:name="android.hardware.telephony" android:required="false"
```

The android.hardware.touchscreen requirement is also implicitly added. If you want your APK to be visible on TVs which are non-touchscreen devices you should add the following to your manifest:

```
<uses-feature android:name="android.hardware.touchscreen" android:required="false"
```

Once you've completed the pre-launch checklist, upload your APKs to Google Play. It may take a bit for the application to show up when browsing Google Play, but when it does, perform one last check. Download the application onto any test devices you may have, to make sure that the APKs are targeting the intended devices. Congratulations, you're done!

# Creating Multiple APKs for Different Screen Sizes

When developing your Android application to take advantage of multiple APKs on Google Play, it's important to adopt some good practices from the get-go, and prevent unnecessary headaches further into the development process. This lesson shows you how to create multiple APKs of your app, each covering a different class of screen size. You will also gain some tools necessary to make maintaining a multiple APK codebase as painless as possible.

## Confirm You Need Multiple APKs

When trying to create an application that works across multiple sizes of Android devices, naturally you want your application to take advantage of all the available space on larger devices, without sacrificing compatibility or usability on the smaller screens. It may seem at the outset as though multiple APK support is the best solution, but this often isn't the case. The [Using Single APK Instead \(/guide/google/play/publishing/multiple-apks.html#ApiLevelOptions\)](#) section of the multiple APK developer guide includes some useful information on how to accomplish this with a single APK, including use of our support library. You should also read the guide to [supporting multiple screens \(/guide/practices/screens\\_support.html\)](#), and there's even a [support library \(http://android-developers.blogspot.com/2011/03/fragments-for-all.html\)](#) you can download using the Android SDK, which lets you use fragments on pre-Honeycomb devices (making multiple-screen support in a single APK much easier).

If you can manage it, confining your application to a single APK has several advantages, including:

- Publishing and testing are easier
- There's only one codebase to maintain
- Your application can adapt to device configuration changes
- App restore across devices just works
- You don't have to worry about market preference, behavior from "upgrades" from one APK to the next, or which APK goes with which class of devices

The rest of this lesson assumes that you've researched the topic, studiously absorbed the material in the resources linked, and determined that multiple APKs are the right path for your application.

## Chart Your Requirements

Start off by creating a simple chart to quickly determine how many APKs you need, and what screen size(s) each APK covers. Fortunately, it's easy to chart out your requirements quickly and easily, and have a reference for later. Start out with a row of cells representing the various screen

### THIS LESSON TEACHES YOU TO

1. [Confirm You Need Multiple APKs](#)
2. [Chart Your Requirements](#)
3. [Put All Common Code and Resources in a Library Project.](#)
4. [Create New APK Projects](#)
5. [Adjust the Manifests](#)
6. [Go Over Pre-launch Checklist](#)

### YOU SHOULD ALSO READ

- [Multiple APK Support](#)
- [Supporting Multiple Screens](#)

sizes available on the Android platform.

small	normal	large	xlarge
-------	--------	-------	--------

Now just color in the chart such that each color represents an APK. Here's one example of how you might apply each APK to a certain range of screen sizes.

small	normal	large	xlarge
-------	--------	-------	--------

Depending on your needs, you could also have two APKs, "small and everything else" or "xlarge and everything else". Coloring in the chart also makes intra-team communication easier—You can now simply refer to each APK as "blue", "green", or "red", no matter how many different screen types it covers.

## Put All Common Code and Resources in a Library Project.

---

Whether you're modifying an existing Android application or starting one from scratch, this is the first thing that you should do to the codebase, and by far the most important. Everything that goes into the library project only needs to be updated once (think language-localized strings, color themes, bugs fixed in shared code), which improves your development time and reduces the likelihood of mistakes that could have been easily avoided.

Note: While the implementation details of how to create and include library projects are beyond the scope of this lesson, you can get up to speed quickly on their creation at the following links:

- [Setting up a library project \(Eclipse\)](#)
- [Setting up a library project \(Command line\)](#)

If you're converting an existing application to use multiple APK support, scour your codebase for every localized string file, list of values, theme colors, menu icons and layout that isn't going to change across APKs, and put it all in the library project. Code that isn't going to change much should also go in the library project. You'll likely find yourself extending these classes to add a method or two from APK to APK.

If, on the other hand, you're creating the application from scratch, try as much as possible to write code in the library project *first*, then only move it down to an individual APK if necessary. This is much easier to manage in the long run than adding it to one, then another, then another, then months later trying to figure out whether this blob can be moved up to the library section without screwing anything up.

## Create New APK Projects

---

There should be a separate Android project for each APK you're going to release. For easy organization, place the library project and all related APK projects under the same parent folder. Also remember that each APK needs to have the same package name, although they don't necessarily need to share the package name with the library. If you were to have 3 APKs following the scheme described earlier, your root directory might look like this:

```
alex lucas:~/code/multi-apks-root$ ls
foo-blue
foo-green
foo-lib
foo-red
```

Once the projects are created, add the library project as a reference to each APK project. If possible, define your starting Activity in the library project, and extend that Activity in your APK project. Having a starting activity defined in the library project gives you a chance to put all your application initialization in one place, so that each individual APK doesn't have to re-implement "universal" tasks like initializing Analytics, running licensing checks, and any other initialization procedures that don't change much from APK to APK.

## Adjust the Manifests

---

When a user downloads an application which uses multiple APKs through Google Play, the correct APK to use is chosen using two simple rules:

- The manifest has to show that particular APK is eligible
- Of the eligible APKs, highest version number wins

By way of example, let's take the set of multiple APKs described earlier, and assume that each APK has been set to support all screen sizes larger than its "target" screen size. Taken individually, the possible range of each APK would look like this:

small	normal	large	xlarge
small	normal	large	xlarge
small	normal	large	xlarge

However, by using the "highest version number wins" rule, if we set the `versionCode` attribute in each APK such that red  $\geq$  green  $\geq$  blue, the chart effectively collapses down to this:

small	normal	large	xlarge
-------	--------	-------	--------

Now, let's further assume that the Red APK has some requirement on it that the other two don't. The [Filters on Google Play \(/guide/google/play/filters.html\)](#) page of the Android Developer guide has a whole list of possible culprits. For the sake of example, let's assume that red requires a front-facing camera. In fact, the entire point of the red APK is to use the extra available screen space to do entertaining things with that front-facing camera. But, it turns out, not all xlarge devices even HAVE front-facing cameras! The horror!

Fortunately, if a user is browsing Google Play from one such device, Google Play will look at the manifest, see that Red lists the front-facing camera as a requirement, and quietly ignore it, having determined that Red and that device are not a match made in digital heaven. It will then see that Green is not only compatible with xlarge devices, but also doesn't care whether or not there's a front-facing camera! The app can still be downloaded from Google Play by the user, because

despite the whole front-camera mishap, there was still an APK that supported that particular screen size.

In order to keep all your APKs on separate "tracks", it's important to have a good version code scheme. The recommended one can be found on the [Version Codes \(/guide/google/play/publishing/multiple-apks.html#VersionCodes\)](#) area of our developer guide. Since the example set of APKs is only dealing with one of 3 possible dimensions, it would be sufficient to separate each APK by 1000 and increment from there. This might look like:

Blue: 1001, 1002, 1003, 1004...

Green: 2001, 2002, 2003, 2004...

Red: 3001, 3002, 3003, 3004...

Putting this all together, your Android Manifests would likely look something like the following:

Blue:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1001" android:versionName="1.0" package="com.example.foo"
<supports-screens android:smallScreens="true"
    android:normalScreens="true"
    android:largeScreens="true"
    android:xlargeScreens="true" />
...

```

Green:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="2001" android:versionName="1.0" package="com.example.foo"
<supports-screens android:smallScreens="false"
    android:normalScreens="false"
    android:largeScreens="true"
    android:xlargeScreens="true" />
...

```

Red:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="3001" android:versionName="1.0" package="com.example.foo"
<supports-screens android:smallScreens="false"
    android:normalScreens="false"
    android:largeScreens="false"
    android:xlargeScreens="true" />
...

```

Note that technically, multiple APK's will work with either the supports-screens tag, or the compatible-screens tag. Supports-screens is generally preferred, and it's generally a really bad idea to use both tags in the same manifest. It makes things needlessly complicated, and increases

the opportunity for errors. Also note that instead of taking advantage of the default values (small and normal are always true by default), the manifests explicitly set the value for each screen size. This can save you headaches down the line. For instance, a manifest with a target SDK of < 9 will have `xlarge` automatically set to false, since that size didn't exist yet. So be explicit!

## Go Over Pre-launch Checklist

---

Before uploading to Google Play, double-check the following items. Remember that these are specifically relevant to multiple APKs, and in no way represent a complete checklist for all applications being uploaded to Google Play.

- All APKs must have the same package name
- All APKs must be signed with the same certificate
- Every screen size you want your APK to support, set to true in the manifest. Every screen size you want it to avoid, set to false
- Double check your manifest filters for conflicting information (an APK that only supports cupcake on XLARGE screens isn't going to be seen by anybody)
- Each APK's manifest must be unique across at least one of supported screen, openGL texture, or platform version
- Try to test each APK on at least one device. Barring that, you have one of the most customizable device emulators in the business sitting on your development machine. Go nuts!

It's also worth inspecting the compiled APK before pushing to market, to make sure there aren't any surprises that could hide your application on Google Play. This is actually quite simple using the "aapt" tool. Aapt (the Android Asset Packaging Tool) is part of the build process for creating and packaging your Android applications, and is also a very handy tool for inspecting them.

```
>aapt dump badging
package: name='com.example.hello' versionCode='1' versionName='1.0'
sdkVersion:'11'
uses-permission:'android.permission.SEND_SMS'
application-label:'Hello'
application-icon-120:'res/drawable-ldpi/icon.png'
application-icon-160:'res/drawable-mdpi/icon.png'
application-icon-240:'res/drawable-hdpi/icon.png'
application: label='Hello' icon='res/drawable-mdpi/icon.png'
launchable-activity: name='com.example.hello.HelloActivity' label='Hello' icon=''
uses-feature:'android.hardware.telephony'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '120' '160' '240'
```

When you examine aapt output, be sure to check that you don't have conflicting values for `supports-screens` and `compatible-screens`, and that you don't have unintended "uses-feature" values that were added as a result of permissions you set in the manifest. In the example above, the APK will be invisible to most, if not all devices.

Why? By adding the required permission `SEND_SMS`, the feature requirement of `android.hardware.telephony` was implicitly added. Since most (if not all) `xlarge` devices are tablets without telephony hardware in them, Google Play will filter out this APK in these cases, until future devices come along which are both large enough to report as `xlarge` screen size, and possess telephony hardware.

Fortunately this is easily fixed by adding the following to your manifest:

```
<uses-feature android:name="android.hardware.telephony" android:required="false"
```

The `android.hardware.touchscreen` requirement is also implicitly added. If you want your APK to be visible on TVs which are non-touchscreen devices you should add the following to your manifest:

```
<uses-feature android:name="android.hardware.touchscreen" android:required="false"
```

Once you've completed the pre-launch checklist, upload your APKs to Google Play. It may take a bit for the application to show up when browsing Google Play, but when it does, perform one last check. Download the application onto any test devices you may have to make sure that the APKs are targeting the intended devices. Congratulations, you're done!

# Creating Multiple APKs for Different GL Textures

When developing your Android application to take advantage of multiple APKs on Google Play, it's important to adopt some good practices from the get-go, and prevent unnecessary headaches further into the development process. This lesson shows you how to create multiple APKs of your app, each supporting a different subset of OpenGL texture formats. You will also gain some tools necessary to make maintaining a multiple APK codebase as painless as possible.

## Confirm You Need Multiple APKs

When trying to create an application that works across all available Android-powered devices, naturally you want your application look its best on each individual device, regardless of the fact they don't all support the same set of GL textures. It may seem at the outset as though multiple APK support is the best solution, but this often isn't the case. The [Using Single APK Instead \(/guide/google/play/publishing/multiple-apks.html#ApiLevelOptions\)](#) section of the multiple APK developer guide includes some useful information on how to accomplish this with a single APK, including how to [detect supported texture formats at runtime \(/guide/google/play/publishing/multiple-apks.html#TextureOptions\)](#). Depending on your situation, it might be easier to bundle all formats with your application, and simply pick which one to use at runtime.

If you can manage it, confining your application to a single APK has several advantages, including:

- Publishing and Testing are easier
- There's only one codebase to maintain
- Your application can adapt to device configuration changes
- App restore across devices just works
- You don't have to worry about market preference, behavior from "upgrades" from one APK to the next, or which APK goes with which class of devices

The rest of this lesson assumes that you've researched the topic, studiously absorbed the material in the resources linked, and determined that multiple APKs are the right path for your application.

## Chart Your Requirements

The Android Developer Guide provides a handy reference of some of common supported textures on the [supports-gl-texture page \(/guide/topics/manifest/supports-gl-texture-element.html\)](#). This page also contains some hints as to which phones (or families of phones) support particular texture formats. Note that it's generally a good idea for one of your APKs to support ETC1, as that texture format is supported by all Android-powered devices that support the OpenGL ES 2.0 spec.

### THIS LESSON TEACHES YOU TO

1. [Confirm You Need Multiple APKs](#)
2. [Chart Your Requirements](#)
3. [Put All Common Code and Resources in a Library Project](#)
4. [Create New APK Projects](#)
5. [Adjust the Manifests](#)
6. [Go Over Pre-launch Checklist](#)

### YOU SHOULD ALSO READ

- [Multiple APK Support](#)

Since most Android-powered devices support more than one texture format, you need to establish an order of preference. Create a chart including all the formats that your application is going to support. The left-most cell is going to be the lowest priority (It will probably be ETC1, a really solid default in terms of performance and compatibility). Then color in the chart such that each cell represents an APK.

ETC1	ATI	PowerVR
------	-----	---------

Coloring in the chart does more than just make this guide less monochromatic - It also has a way of making intra-team communication easier- You can now simply refer to each APK as "blue", "green", or "red", instead of "The one that supports ETC1 texture formats", etc.

## Put All Common Code and Resources in a Library Project

---

Whether you're modifying an existing Android application or starting one from scratch, this is the first thing that you should do to the codebase, and by far the most important. Everything that goes into the library project only needs to be updated once (think language-localized strings, color themes, bugs fixed in shared code), which improves your development time and reduces the likelihood of mistakes that could have been easily avoided.

**Note:** While the implementation details of how to create and include library projects are beyond the scope of this lesson, you can get up to speed quickly on their creation at the following links:

- [Setting up a library project \(Eclipse\)](#)
- [Setting up a library project \(Command line\)](#)

If you're converting an existing application to use multiple APK support, scour your codebase for every localized string file, list of values, theme colors, menu icons and layout that isn't going to change across APKs, and put it all in the library project. Code that isn't going to change much should also go in the library project. You'll likely find yourself extending these classes to add a method or two from APK to APK.

If, on the other hand, you're creating the application from scratch, try as much as possible to write code in the library project *first*, then only move it down to an individual APK if necessary. This is much easier to manage in the long run than adding it to one, then another, then another, then months later trying to figure out whether this blob can be moved up to the library section without screwing anything up.

## Create New APK Projects

---

There should be a separate Android project for each APK you're going to release. For easy organization, place the library project and all related APK projects under the same parent folder. Also remember that each APK needs to have the same package name, although they don't necessarily need to share the package name with the library. If you were to have 3 APKs following the scheme described earlier, your root directory might look like this:

```
alex lucas:~/code/multi-apks-root$ ls
foo-blue
```

```
foo-green
foo-lib
foo-red
```

Once the projects are created, add the library project as a reference to each APK project. If possible, define your starting Activity in the library project, and extend that Activity in your APK project. Having a starting activity defined in the library project gives you a chance to put all your application initialization in one place, so that each individual APK doesn't have to re-implement "universal" tasks like initializing Analytics, running licensing checks, and any other initialization procedures that don't change much from APK to APK.

## Adjust the Manifests

---

When a user downloads an application which uses multiple APKs through Google Play, the correct APK to use is chosen using some simple rules:

- The manifest has to show that particular APK is eligible
- Of the eligible APKs, highest version number wins
- If *any* of the texture formats listed in your APK are supported by the device on market, that device is considered eligible

With regards to GL Textures, that last rule is important. It means that you should, for instance, be *very* careful about using different GL formats in the same application. If you were to use PowerVR 99% of the time, but use ETC1 for, say, your splash screen... Then your manifest would necessarily indicate support for both formats. A device that *only* supported ETC1 would be deemed compatible, your app would download, and the user would see some thrilling crash messages. The common case is going to be that if you're using multiple APKs specifically to target different devices based on GL texture support, it's going to be one texture format per APK.

This actually makes texture support a little bit different than the other two multiple APK dimensions, API level and screen size. Any given device only has one API level, and one screen size, and it's up to the APK to support a range of them. With textures, the APK will generally support one texture, and the device will support many. There will often be overlap in terms of one device supporting many APKs, but the solution is the same: Version codes.

By way of example, take a few devices, and see how many of the APKs defined earlier fit each device.

FooPhone	Nexus S	Evo
ETC1	ETC1	ETC1
	PowerVR	ATI TC

Assuming that PowerVR and ATI formats are both preferred over ETC1 when available, then according to the "highest version number wins" rule, if we set the `versionCode` attribute in each APK such that  $\text{red} \geq \text{green} \geq \text{blue}$ , then both Red and Green will always be chosen over Blue on devices which support them, and should a device ever come along which supports both Red and Green, red will be chosen.

In order to keep all your APKs on separate "tracks," it's important to have a good version code scheme. The recommended one can be found on the Version Codes area of our developer guide. Since the example set of APKs is only dealing with one of 3 possible dimensions, it would be sufficient to separate each APK by 1000 and increment from there. This might look like:

Blue: 1001, 1002, 1003, 1004...

Green: 2001, 2002, 2003, 2004...

Red: 3001, 3002, 3003, 3004...

Putting this all together, your Android Manifests would likely look something like the following:

Blue:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1001" android:versionName="1.0" package="com.example.foo"
    <supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />
    ...
    ...
```

Green:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="2001" android:versionName="1.0" package="com.example.foo"
    <supports-gl-texture android:name="GL_AMD_compressed_ATC_texture" />
    ...
    ...
```

Red:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="3001" android:versionName="1.0" package="com.example.foo"
    <supports-gl-texture android:name="GL_IMG_texture_compression_pvrtc" />
    ...
    ...
```

## Go Over Pre-launch Checklist

---

Before uploading to Google Play, double-check the following items. Remember that these are specifically relevant to multiple APKs, and in no way represent a complete checklist for all applications being uploaded to Google Play.

- All APKs must have the same package name
- All APKs must be signed with the same certificate
- Double check your manifest filters for conflicting information (an APK that only supports cupcake on XLARGE screens isn't going to be seen by anybody)
- Each APK's manifest must be unique across at least one of supported screen, OpenGL texture, or platform version
- Try to test each APK on at least one device. Barring that, you have one of the most customizable device emulators in the business sitting on your development machine. Go nuts!

It's also worth inspecting the compiled APK before pushing to market, to make sure there aren't any surprises that could hide your application on Google Play. This is actually quite simple using the "aapt" tool. Aapt (the Android Asset Packaging Tool) is part of the build process for creating and packaging your Android applications, and is also a very handy tool for inspecting them.

```
>aapt dump badging
package: name='com.example.hello' versionCode='1' versionName='1.0'
sdkVersion:'11'
uses-permission:'android.permission.SEND_SMS'
application-label:'Hello'
application-icon-120:'res/drawable-ldpi/icon.png'
application-icon-160:'res/drawable-mdpi/icon.png'
application-icon-240:'res/drawable-hdpi/icon.png'
application: label='Hello' icon='res/drawable-mdpi/icon.png'
launchable-activity: name='com.example.hello.HelloActivity' label='Hello' icon=''
uses-feature:'android.hardware.telephony'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '120' '160' '240'
```

When you examine aapt output, be sure to check that you don't have conflicting values for supports-screens and compatible-screens, and that you don't have unintended "uses-feature" values that were added as a result of permissions you set in the manifest. In the example above, the APK will be invisible to most, if not all devices.

Why? By adding the required permission SEND\_SMS, the feature requirement of android.hardware.telephony was implicitly added. Since most (if not all) xlarge devices are tablets without telephony hardware in them, Google Play will filter out this APK in these cases, until future devices come along which are both large enough to report as xlarge screen size, and possess telephony hardware.

Fortunately this is easily fixed by adding the following to your manifest:

```
<uses-feature android:name="android.hardware.telephony" android:required="false"
```

The android.hardware.touchscreen requirement is also implicitly added. If you want your APK to be visible on TVs which are non-touchscreen devices you should add the following to your manifest:

```
<uses-feature android:name="android.hardware.touchscreen" android:required="false"
```

Once you've completed the pre-launch checklist, upload your APKs to Google Play. It may take a bit for the application to show up when browsing Google Play, but when it does, perform one last check. Download the application onto any test devices you may have to make sure that the APKs are targeting the intended devices. Congratulations, you're done!

# Creating Multiple APKs with 2+ Dimensions

When developing your Android application to take advantage of multiple APKs on Google Play, it's important to adopt some good practices from the get-go, and prevent unnecessary headaches further into the development process. This lesson shows you how to create multiple APKs of your app, each covering a different class of screen size. You will also gain some tools necessary to make maintaining a multiple APK codebase as painless as possible.

## Confirm You Need Multiple APKs

When trying to create an application that works across the huge range of available Android devices, naturally you want your application look its best on each individual device. You want to take advantage of the space of large screens but still work on small ones, to use new Android API features or visual textures available on cutting edge devices but not abandon older ones. It may seem at the outset as though multiple APK support is the best solution, but this often isn't the case. The [Using Single APK Instead \(/guide/google/play/publishing/multiple-apks.html#ApiLevelOptions\)](#) section of the multiple APK guide includes some useful information on how to accomplish all of this with a single APK, including use of our [support library \(http://android-developers.blogspot.com/2011/03/fragments-for-all.html\)](#), and links to resources throughout the Android Developer guide.

If you can manage it, confining your application to a single APK has several advantages, including:

- Publishing and Testing are easier
- There's only one codebase to maintain
- Your application can adapt to device configuration changes
- App restore across devices just works
- You don't have to worry about market preference, behavior from "upgrades" from one APK to the next, or which APK goes with which class of devices

The rest of this lesson assumes that you've researched the topic, studiously absorbed the material in the resources linked, and determined that multiple APKs are the right path for your application.

## Chart Your Requirements

Start off by creating a simple chart to quickly determine how many APKs you need, and what screen size(s) each APK covers. Fortunately, it's easy to chart out your requirements quickly, easily, and have an easy reference for later. Let's say you want to split your APKs across two dimensions, API and screen size. Create a table with a row and column for each possible pair of values, and color in some "blobs", each color representing one APK.

### THIS LESSON TEACHES YOU TO

1. [Confirm You Need Multiple APKs](#)
2. [Chart Your Requirements](#)
3. [Put All Common Code and Resources in a Library Project.](#)
4. [Create New APK Projects](#)
5. [Adjust the Manifests](#)
6. [Go Over Pre-launch Checklist](#)

### YOU SHOULD ALSO READ

- [Multiple APK Support](#)

	3	4	5	6	7	8	9	10	11	12	+
small											
normal											
large											
xlarge											

Above is an example with four APKs. Blue is for all small/normal screen devices, Green is for large screen devices, and Red is for xlarge screen devices, all with an API range of 3-10. Purple is a special case, as it's for all screen sizes, but only for API 11 and up. More importantly, just by glancing at this chart, you immediately know which APK covers any given API/screen-size combo. To boot, you also have swanky codenames for each one, since "Have we tested red on the ?" is a lot easier to ask your cubie than "Have we tested the 3-to-10 xlarge APK against the Xoom?" Print this chart out and hand it to every person working on your codebase. Life just got a lot easier.

## Put All Common Code and Resources in a Library Project.

---

Whether you're modifying an existing Android application or starting one from scratch, this is the first thing that you should do to the codebase, and by far the most important. Everything that goes into the library project only needs to be updated once (think language-localized strings, color themes, bugs fixed in shared code), which improves your development time and reduces the likelihood of mistakes that could have been easily avoided.

Note: While the implementation details of how to create and include library projects are beyond the scope of this lesson, you can get up to speed quickly on their creation at the following links:

- [Setting up a library project \(Eclipse\)](#)
- [Setting up a library project \(Command line\)](#)

If you're converting an existing application to use multiple APK support, scour your codebase for every localized string file, list of values, theme colors, menu icons and layout that isn't going to change across APKs, and put it all in the library project. Code that isn't going to change much should also go in the library project. You'll likely find yourself extending these classes to add a method or two from APK to APK.

If, on the other hand, you're creating the application from scratch, try as much as possible to write code in the library project *first*, then only move it down to an individual APK if necessary. This is much easier to manage in the long run than adding it to one, then another, then another, then months later trying to figure out whether this blob can be moved up to the library section without screwing anything up.

## Create New APK Projects

---

There should be a separate Android project for each APK you're going to release. For easy

organization, place the library project and all related APK projects under the same parent folder. Also remember that each APK needs to have the same package name, although they don't necessarily need to share the package name with the library. If you were to have 3 APKs following the scheme described earlier, your root directory might look like this:

```
alex lucas:~/code/multi-apks-root$ ls
foo-blue
foo-green
foo-lib
foo-purple
foo-red
```

Once the projects are created, add the library project as a reference to each APK project. If possible, define your starting Activity in the library project, and extend that Activity in your APK project. Having a starting activity defined in the library project gives you a chance to put all your application initialization in one place, so that each individual APK doesn't have to re-implement "universal" tasks like initializing Analytics, running licensing checks, and any other initialization procedures that don't change much from APK to APK.

## Adjust the Manifests

---

When a user downloads an application which uses multiple APKs through Google Play, the correct APK to use is chosen using two simple rules:

- The manifest has to show that particular APK is eligible
- Of the eligible APKs, highest version number wins.

By way of example, let's take the set of multiple APKs described earlier, and assume that each APK has been set to support all screen sizes larger than its "target" screen size. Let's look at the sample chart from earlier:

	3	4	5	6	7	8	9	10	11	12	+
small											
normal											
large											
xlarge											

Since it's okay for coverage to overlap, we can describe the area covered by each APK like so:

- Blue covers all screens, minSDK 3.
- Green covers Large screens and higher, minSDK 3.
- Red covers XLarge screens (generally tablets), minSDK of 9.
- Purple covers all screens, minSDK of 11.

Note that there's a *lot* of overlap in those rules. For instance, an XLarge device with API 11 can

conceivably run any one of the 4 APKs specified. However, by using the "highest version number wins" rule, we can set an order of preference as follows:

Purple  $\geq$  Red  $\geq$  Green  $\geq$  Blue

Why allow all the overlap? Let's pretend that the Purple APK has some requirement on it that the other two don't. The [Filters on Google Play \(/guide/google/play/filters.html\)](#) page of the Android Developer guide has a whole list of possible culprits. For the sake of example, let's assume that Purple requires a front-facing camera. In fact, the entire point of Purple is to use entertaining things with the front-facing camera! But, it turns out, not all API 11+ devices even HAVE front-facing cameras! The horror!

Fortunately, if a user is browsing Google Play from one such device, Google Play will look at the manifest, see that Purple lists the front-facing camera as a requirement, and quietly ignore it, having determined that Purple and that device are not a match made in digital heaven. It will then see that Red is not only compatible with `xlarge` devices, but also doesn't care whether or not there's a front-facing camera! The app can still be downloaded from Google Play by the user, because despite the whole front-camera mishap, there was still an APK that supported that particular API level.

In order to keep all your APKs on separate "tracks", it's important to have a good version code scheme. The recommended one can be found on the [Version Codes \(/guide/google/play/publishing/multiple-apks.html#VersionCodes\)](#) area of our developer guide. It's worth reading the whole section, but the basic gist is for this set of APKs, we'd use two digits to represent the minSDK, two to represent the min/max screen size, and 3 to represent the build number. That way, when the device upgraded to a new version of Android, (say, from 10 to 11), any APKs that are now eligible and preferred over the currently installed one would be seen by the device as an "upgrade". The version number scheme, when applied to the example set of APKs, might look like:

Blue: 0304001, 0304002, 0304003...

Green: 0334001, 0334002, 0334003

Red: 0344001, 0344002, 0344003...

Purple: 1104001, 1104002, 1104003...

Putting this all together, your Android Manifests would likely look something like the following:

Blue:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    android:versionCode="0304001" android:versionName="1.0" package="com.example.  
    <uses-sdk android:minSdkVersion="3" />  
    <supports-screens android:smallScreens="true"  
        android:normalScreens="true"  
        android:largeScreens="true"  
        android:xlargeScreens="true" />  
    ...
```

## Green:

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

```

    android:versionCode="0334001" android:versionName="1.0" package="com.example.
<uses-sdk android:minSdkVersion="3" />
<supports-screens android:smallScreens="false"
    android:normalScreens="false"
    android:largeScreens="true"
    android:xlargeScreens="true" />
...

```

Red:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="0344001" android:versionName="1.0" package="com.example.
<uses-sdk android:minSdkVersion="3" />
<supports-screens android:smallScreens="false"
    android:normalScreens="false"
    android:largeScreens="false"
    android:xlargeScreens="true" />
...

```

Purple:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1104001" android:versionName="1.0" package="com.example.
<uses-sdk android:minSdkVersion="11" />
<supports-screens android:smallScreens="true"
    android:normalScreens="true"
    android:largeScreens="true"
    android:xlargeScreens="true" />
...

```

Note that technically, multiple APK's will work with either the supports-screens tag, or the compatible-screens tag. Supports-screens is generally preferred, and it's generally a really bad idea to use both- It makes things needlessly complicated, and increases the opportunity for errors. Also note that instead of taking advantage of the default values (small and normal are always true by default), the manifests explicitly set the value for each screen size. This can save you headaches down the line - By way of example, a manifest with a target SDK of < 9 will have xlarge automatically set to false, since that size didn't exist yet. So be explicit!

## Go Over Pre-launch Checklist

---

Before uploading to Google Play, double-check the following items. Remember that these are specifically relevant to multiple APKs, and in no way represent a complete checklist for all applications being uploaded to Google Play.

- All APKs must have the same package name.
- All APKs must be signed with the same certificate.

- If the APKs overlap in platform version, the one with the higher minSdkVersion must have a higher version code.
- Every screen size you want your APK to support, set to true in the manifest. Every screen size you want it to avoid, set to false.
- Double check your manifest filters for conflicting information (an APK that only supports cupcake on XLARGE screens isn't going to be seen by anybody)
- Each APK's manifest must be unique across at least one of supported screen, OpenGL texture, or platform version.
- Try to test each APK on at least one device. Barring that, you have one of the most customizable device emulators in the business sitting on your development machine. Go nuts!

It's also worth inspecting the compiled APK before pushing to market, to make sure there aren't any surprises that could hide your application on Google Play. This is actually quite simple using the "aapt" tool. Aapt (the Android Asset Packaging Tool) is part of the build process for creating and packaging your Android applications, and is also a very handy tool for inspecting them.

```
>aapt dump badging
package: name='com.example.hello' versionCode='1' versionName='1.0'
sdkVersion:'11'
uses-permission:'android.permission.SEND_SMS'
application-label:'Hello'
application-icon-120:'res/drawable-ldpi/icon.png'
application-icon-160:'res/drawable-mdpi/icon.png'
application-icon-240:'res/drawable-hdpi/icon.png'
application: label='Hello' icon='res/drawable-mdpi/icon.png'
launchable-activity: name='com.example.hello.HelloActivity' label='Hello' icon=''
uses-feature:'android.hardware.telephony'
uses-feature:'android.hardware.touchscreen'
main
supports-screens: 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '120' '160' '240'
```

When you examine aapt output, be sure to check that you don't have conflicting values for supports-screens and compatible-screens, and that you don't have unintended "uses-feature" values that were added as a result of permissions you set in the manifest. In the example above, the APK will be invisible to most, if not all devices.

Why? By adding the required permission SEND\_SMS, the feature requirement of android.hardware.telephony was implicitly added. Since most (if not all) xlarge devices are tablets without telephony hardware in them, Google Play will filter out this APK in these cases, until future devices come along which are both large enough to report as xlarge screen size, and possess telephony hardware.

Fortunately this is easily fixed by adding the following to your manifest:

```
<uses-feature android:name="android.hardware.telephony" android:required="false"
```

The android.hardware.touchscreen requirement is also implicitly added. If you want your APK to

be visible on TVs which are non-touchscreen devices you should add the following to your manifest:

```
<uses-feature android:name="android.hardware.touchscreen" android:required="false"
```

Once you've completed the pre-launch checklist, upload your APKs to Google Play. It may take a bit for the application to show up when browsing Google Play, but when it does, perform one last check. Download the application onto any test devices you may have to make sure that the APKs are targeting the intended devices. Congratulations, you're done!

# Creating Backward-Compatible UIs

This class demonstrates how to use UI components and APIs available in newer versions of Android in a backward-compatible way, ensuring that your application still runs on previous versions of the platform.

Throughout this class, the new [ActionBar Tabs](#) ([/guide/topics/ui/actionbar.html#Tabs](#)) feature introduced in Android 3.0 (API level 11) serves as the guiding example, but you can apply these techniques to other UI components and API features.

## Lessons

### Abstracting the New APIs

Determine which features and APIs your application needs. Learn how to define application-specific, intermediary Java interfaces that abstract the implementation of the UI component to your application.

### Proxying to the New APIs

Learn how to create an implementation of your interface that uses newer APIs.

### Creating an Implementation with Older APIs

Learn how to create a custom implementation of your interface that uses older APIs.

### Using the Version-Aware Component

Learn how to choose an implementation to use at runtime, and begin using the interface in your application.

### DEPENDENCIES AND PREREQUISITES

- API level 5
- [The Android Support Package](#)

### YOU SHOULD ALSO READ

- [ActionBarCompat](#)
- [How to have your \(Cup\)cake and eat it too](#)

### TRY IT OUT

[Download the sample app](#)

TabCompat.zip

# Abstracting the New APIs

Suppose you want to use [action bar tabs \(/guide/topics/ui/actionbar.html#Tabs\)](#) as the primary form of top-level navigation in your application. Unfortunately, the [ActionBar \(/reference/android/app/ActionBar.html\)](#) APIs are only available in Android 3.0 or later (API level 11+). Thus, if you want to distribute your application to devices running earlier versions of the platform, you need to provide an implementation that supports the newer API while providing a fallback mechanism that uses older APIs.

In this class, you build a tabbed user interface (UI) component that uses abstract classes with version-specific implementations to provide backward-compatibility. This lesson describes how to create an abstraction layer for the new tab APIs as the first step toward building the tab component.

## THIS LESSON TEACHES YOU TO:

- [Prepare for Abstraction](#)
- [Create an Abstract Tab Interface](#)
- [Abstract ActionBar.Tab](#)
- [Abstract ActionBar Tab Methods](#)

## YOU SHOULD ALSO READ

- [Action Bar](#)
- [Action Bar Tabs](#)

## TRY IT OUT

[Download the sample app](#)

TabCompat.zip

## Prepare for Abstraction

[Abstraction \(http://en.wikipedia.org/wiki/Abstraction\\_\(computer\\_science\)\)](#) in the Java programming language involves the creation of one or more interfaces or abstract classes to hide implementation details. In the case of newer Android APIs, you can use abstraction to build version-aware components that use the current APIs on newer devices, and fallback to older, more compatible APIs on older devices.

When using this approach, you first determine what newer classes you want to be able to use in a backward compatible way, then create abstract classes, based on the public interfaces of the newer classes. In defining the abstraction interfaces, you should mirror the newer API as much as possible. This maximizes forward-compatibility and makes it easier to drop the abstraction layer in the future when it is no longer necessary.

After creating abstract classes for these new APIs, any number of implementations can be created and chosen at runtime. For the purposes of backward-compatibility, these implementations can vary by required API level. Thus, one implementation may use recently released APIs, while others can use older APIs.

## Create an Abstract Tab Interface

In order to create a backward-compatible version of tabs, you should first determine which features and specific APIs your application requires. In the case of top-level section tabs, suppose you have the following functional requirements:

1. Tab indicators should show text and an icon.

2. Tabs can be associated with a fragment instance.
3. The activity should be able to listen for tab changes.

Preparing these requirements in advance allows you to control the scope of your abstraction layer. This means that you can spend less time creating multiple implementations of your abstraction layer and begin using your new backward-compatible implementation sooner.

The key APIs for tabs are in [ActionBar](#) ([/reference/android/app/ActionBar.html](#)) and [ActionBar.Tab](#) ([/reference/android/app/ActionBar.Tab.html](#)). These are the APIs to abstract in order to make your tabs version-aware. The requirements for this example project call for compatibility back to Eclair (API level 5) while taking advantage of the new tab features in Honeycomb (API Level 11). A diagram of the class structure to support these two implementations and their abstract base classes (or interfaces) is shown below.

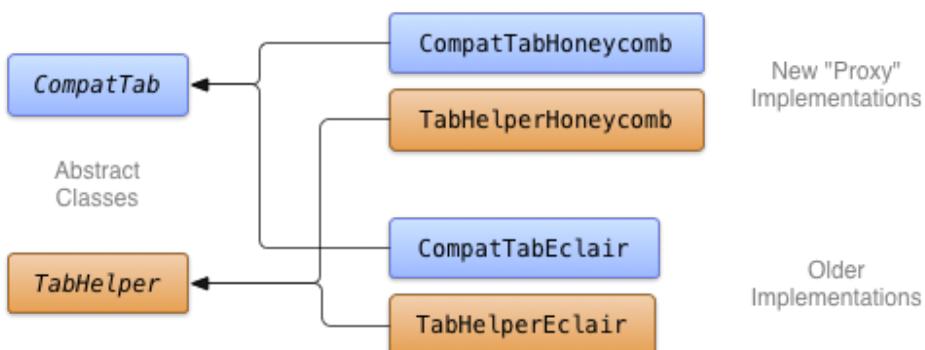


Figure 1. Class diagram of abstract base classes and version-specific implementations.

## Abstract ActionBar.Tab

Get started on building your tab abstraction layer by creating an abstract class representing a tab, that mirrors the [ActionBar.Tab](#) ([/reference/android/app/ActionBar.Tab.html](#)) interface:

```
public abstract class CompatTab {  
    ...  
    public abstract CompatTab setText(int resId);  
    public abstract CompatTab setIcon(int resId);  
    public abstract CompatTab setTabListener(  
        CompatTabListener callback);  
    public abstract CompatTab setFragment(Fragment fragment);  
  
    public abstract CharSequence getText();  
    public abstract Drawable getIcon();  
    public abstract CompatTabListener getCallback();  
    public abstract Fragment getFragment();  
    ...  
}
```

You can use an abstract class instead of an interface here to simplify the implementation of common features such as association of tab objects with activities (not shown in the code snippet).

## Abstract ActionBar Tab Methods

Next, define an abstract class that allows you to create and add tabs to an activity, like `ActionBar.newTab()` ([/reference/android/app/ActionBar.html#newTab\(\)](/reference/android/app/ActionBar.html#newTab())) and `ActionBar.addTab()` ([/reference/android/app/ActionBar.html#addTab\(android.app.ActionBar.Tab\)](/reference/android/app/ActionBar.html#addTab(android.app.ActionBar.Tab))):

```
public abstract class TabHelper {  
    ...  
  
    public CompatTab newTab(String tag) {  
        // This method is implemented in a later lesson.  
    }  
  
    public abstract void addTab(CompatTab tab);  
  
    ...  
}
```

In the next lessons, you create implementations for `TabHelper` and `CompatTab` that work across both older and newer platform versions.

# Proxying to the New APIs

This lesson shows you how to subclass the CompatTab and TabHelper abstract classes and use new APIs. Your application can use this implementation on devices running a platform version that supports them.

## Implement Tabs Using New APIs

The concrete classes for CompatTab and TabHelper that use newer APIs are a *proxy* implementation. Since the abstract classes defined in the previous lesson mirror the new APIs (class structure, method signatures, etc.), the concrete classes that use these newer APIs simply proxy method calls and their results.

You can directly use newer APIs in these concrete classes—and not crash on earlier devices—because of lazy class loading. Classes are loaded and initialized on first access—instantiating the class or accessing one of its static fields or methods for the first time. Thus, as long as you don't instantiate the Honeycomb-specific implementations on pre-Honeycomb devices, the Dalvik VM won't throw any [VerifyError](#) ([/reference/java/lang/VerifyError.html](#)) exceptions.

A good naming convention for this implementation is to append the API level or platform version code name corresponding to the APIs required by the concrete classes. For example, the native tab implementation can be provided by CompatTabHoneycomb and TabHelperHoneycomb classes, since they rely on APIs available in Android 3.0 (API level 11) or later.

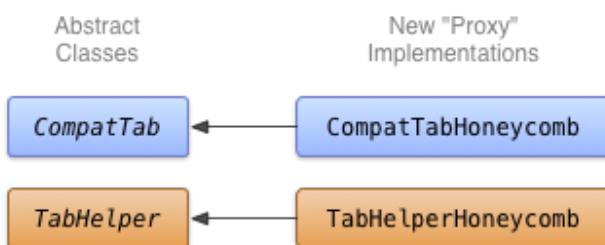


Figure 1. Class diagram for the Honeycomb implementation of tabs.

## Implement CompatTabHoneycomb

CompatTabHoneycomb is the implementation of the CompatTab abstract class that TabHelperHoneycomb uses to reference individual tabs. CompatTabHoneycomb simply proxies all method calls to its contained [ActionBar.Tab](#) ([/reference/android/app/ActionBar.Tab.html](#)) object.

Begin implementing CompatTabHoneycomb using the new [ActionBar.Tab](#) ([/reference/android...](#)

### THIS LESSON TEACHES YOU TO:

1. [Implement Tabs Using New APIs](#)
2. [Implement CompatTabHoneycomb](#)
3. [Implement TabHelperHoneycomb](#)

### YOU SHOULD ALSO READ

- [Action Bar](#)
- [Action Bar Tabs](#)

### TRY IT OUT

[Download the sample app](#)

TabCompat.zip

[/app/ActionBar.Tab.html](#) APIs:

```
public class CompatTabHoneycomb extends CompatTab {  
    // The native tab object that this CompatTab acts as a proxy for.  
    ActionBar.Tab mTab;  
  
    ...  
  
    protected CompatTabHoneycomb(FragmentActivity activity, String tag) {  
        ...  
        // Proxy to new ActionBar.newTab API  
        mTab = activity.getActionBar().newTab();  
    }  
  
    public CompatTab setText(int resId) {  
        // Proxy to new ActionBar.Tab.setText API  
        mTab.setText(resId);  
        return this;  
    }  
  
    ...  
    // Do the same for other properties (icon, callback, etc.)  
}
```

## Implement TabHelperHoneycomb

TabHelperHoneycomb is the implementation of the TabHelper abstract class that proxies method calls to an actual [ActionBar](#) ([/reference/android/app/ActionBar.html](#)), obtained from its contained [Activity](#) ([/reference/android/app/Activity.html](#)).

Implement TabHelperHoneycomb, proxying method calls to the [ActionBar](#) ([/reference/android/app/ActionBar.html](#)) API:

```
public class TabHelperHoneycomb extends TabHelper {  
    ActionBar mActionBar;  
  
    ...  
  
    protected void setUp() {  
        if (mActionBar == null) {  
            mActionBar = mActivity.getActionBar();  
            mActionBar.setNavigationMode(  
                ActionBar.NAVIGATION_MODE_TABS);  
        }  
    }  
  
    public void addTab(CompatTab tab) {  
        ...  
    }  
}
```

```
// Tab is a CompatTabHoneycomb instance, so its  
// native tab object is an ActionBar.Tab.  
mActionBar.addTab((ActionBar.Tab) tab.getTab());  
}  
  
// The other important method, newTab() is part of  
// the base implementation.  
}
```

# Creating an Implementation with Older APIs

This lesson discusses how to create an implementation that mirrors newer APIs yet supports older devices.

## Decide on a Substitute Solution

The most challenging task in using newer UI features in a backward-compatible way is deciding on and implementing an older (fallback) solution for older platform versions. In many cases, it's possible to fulfill the purpose of these newer UI components using older UI framework features. For example:

- Action bars can be implemented using a horizontal [LinearLayout](#) containing image buttons, either as custom title bars or as views in your activity layout. Overflow actions can be presented under the device *Menu* button.
- Action bar tabs can be implemented using a horizontal [LinearLayout](#) containing buttons, or using the [TabWidget](#) UI element.
- [NumberPicker](#) and [Switch](#) widgets can be implemented using [Spinner](#) and [ToggleButton](#) widgets, respectively.
- [ListPopupWindow](#) and [PopupMenu](#) widgets can be implemented using [PopupWindow](#) widgets.

There generally isn't a one-size-fits-all solution for backporting newer UI components to older devices. Be mindful of the user experience: on older devices, users may not be familiar with newer design patterns and UI components. Give some thought as to how the same functionality can be delivered using familiar elements. In many cases this is less of a concern—if newer UI components are prominent in the application ecosystem (such as the action bar), or where the interaction model is extremely simple and intuitive (such as swipe views using a [ViewPager](#)).

### THIS LESSON TEACHES YOU TO:

1. [Decide on a Substitute Solution](#)
2. [Implement Tabs Using Older APIs](#)

### TRY IT OUT

[Download the sample app](#)

TabCompat.zip

## Implement Tabs Using Older APIs

To create an older implementation of action bar tabs, you can use a [TabWidget](#) and [TabHost](#) (although one can alternatively use horizontally laid-out [Button](#) widgets). Implement this in classes called `TabHelperEclair` and `CompatTabEclair`, since this

implementation uses APIs introduced no later than Android 2.0 (Eclair).

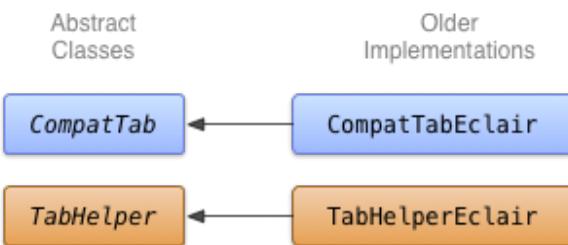


Figure 1. Class diagram for the Eclair implementation of tabs.

The `CompatTabEclair` implementation stores tab properties such as the tab text and icon in instance variables, since there isn't an `ActionBar.Tab` ([/reference/android/app/ActionBar.Tab.html](#)) object available to handle this storage:

```

public class CompatTabEclair extends CompatTab {
    // Store these properties in the instance,
    // as there is no ActionBar.Tab object.
    private CharSequence mText;
    ...

    public CompatTab setText(int resId) {
        // Our older implementation simply stores this
        // information in the object instance.
        mText = mActivity.getResources().getText(resId);
        return this;
    }

    ...
    // Do the same for other properties (icon, callback, etc.)
}
  
```

The `TabHelperEclair` implementation makes use of methods on the `TabHost` ([/reference/android/widget/TabHost.html](#)) widget for creating `TabHost.TabSpec` ([/reference/android/widget/TabHost.TabSpec.html](#)) objects and tab indicators:

```

public class TabHelperEclair extends TabHelper {
    private TabHost mTabHost;
    ...

    protected void setUp() {
        if (mTabHost == null) {
            // Our activity layout for pre-Honeycomb devices
            // must contain a TabHost.
            mTabHost = (TabHost) mActivity.findViewById(
                android.R.id.tabhost);
            mTabHost.setup();
        }
    }
  
```

```
        }

    public void addTab(CompatTab tab) {
        ...
        TabSpec spec = mTabHost
            .newTabSpec(tag)
            .setIndicator(tab.getText()); // And optional icon
        ...
        mTabHost.addTab(spec);
    }

    // The other important method, newTab() is part of
    // the base implementation.
}
```

You now have two implementations of CompatTab and TabHelper: one that works on devices running Android 3.0 or later and uses new APIs, and another that works on devices running Android 2.0 or later and uses older APIs. The next lesson discusses using these implementations in your application.

# Using the Version-Aware Component

Now that you have two implementations of TabHelper and CompatTab—one for Android 3.0 and later and one for earlier versions of the platform—it's time to do something with these implementations. This lesson discusses creating the logic for switching between these implementations, creating version-aware layouts, and finally using the backward-compatible UI component.

## Add the Switching Logic

The TabHelper abstract class acts as a [factory](http://en.wikipedia.org/wiki/Factory_(software_concept)) ([http://en.wikipedia.org/wiki/Factory\\_\(software\\_concept\)](http://en.wikipedia.org/wiki/Factory_(software_concept))) for creating version-appropriate TabHelper and CompatTab instances, based on the current device's platform version:

```
public abstract class TabHelper {
    ...
    // Usage is TabHelper.createInstance(activity)
    public static TabHelper createInstance(FragmentActivity activity) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
            return new TabHelperHoneycomb(activity);
        } else {
            return new TabHelperEclair(activity);
        }
    }

    // Usage is mTabHelper.newTab("tag")
    public CompatTab newTab(String tag) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
            return new CompatTabHoneycomb(mActivity, tag);
        } else {
            return new CompatTabEclair(mActivity, tag);
        }
    }
    ...
}
```

### THIS LESSON TEACHES YOU TO:

1. [Add the Switching Logic](#)
2. [Create a Version-Aware Activity Layout](#)
3. [Use TabHelper in Your Activity](#)

### TRY IT OUT

[Download the sample app](#)

TabCompat.zip

## Create a Version-Aware Activity Layout

The next step is to provide layouts for your activity that can support the two tab implementations. For the older implementation (TabHelperEclair), you need to ensure that your activity layout

contains a [TabWidget](#) ([/reference/android/widget/TabWidget.html](#)) and [TabHost](#) ([/reference/android/widget/TabHost.html](#)), along with a container for tab contents:

res/layout/main.xml:

```
<!-- This layout is for API level 5-10 only. -->
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/tabhost"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="5dp">

        <TabWidget
            android:id="@+id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <FrameLayout
            android:id="@+id/tabcontent"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1" />

    </LinearLayout>
</TabHost>
```

For the TabHelperHoneycomb implementation, all you need is a [FrameLayout](#) ([/reference/android/widget/FrameLayout.html](#)) to contain the tab contents, since the tab indicators are provided by the [ActionBar](#) ([/reference/android/app/ActionBar.html](#)):

res/layout-v11/main.xml:

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

At runtime, Android will decide which version of the main.xml layout to inflate depending on the platform version. This is the same logic shown in the previous section to determine which TabHelper implementation to use.

## Use TabHelper in Your Activity

In your activity's `onCreate()` ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method, you can obtain a `TabHelper` object and add tabs with the following code:

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    setContentView(R.layout.main);  
  
    TabHelper tabHelper = TabHelper.createInstance(this);  
    tabHelper.setUp();  
  
    CompatTab photosTab = tabHelper  
        .newTab("photos")  
        .setText(R.string.tab_photos);  
    tabHelper.addTab(photosTab);  
  
    CompatTab videosTab = tabHelper  
        .newTab("videos")  
        .setText(R.string.tab_videos);  
    tabHelper.addTab(videosTab);  
}
```

When running the application, this code inflates the correct activity layout and instantiates either a `TabHelperHoneycomb` or `TabHelperEclair` object. The concrete class that's actually used is opaque to the activity, since they share the common `TabHelper` interface.

Below are two screenshots of this implementation running on an Android 2.3 and Android 4.0 device.

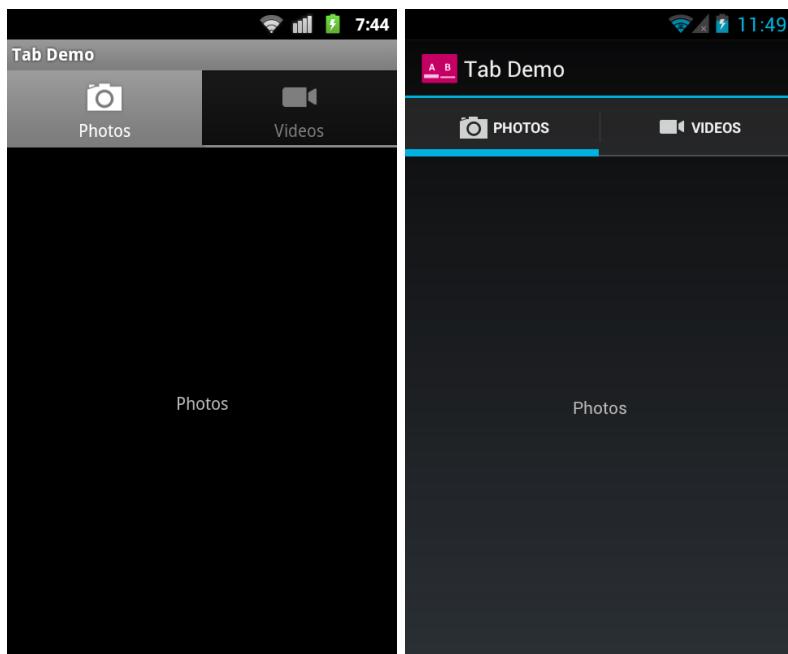


Figure 1. Example screenshots of backward-compatible tabs running on an Android 2.3 device (using `TabHelperEclair`) and an Android 4.0 device (using `TabHelperHoneycomb`).

# Developing for Enterprise

In this class, you'll learn APIs and techniques you can use when developing applications for the enterprise.

## Lessons

---

### [Enhancing Security with Device Management Policies](#)

In this lesson, you will learn how to create a security-aware application that manages access to its content by enforcing device management policies

#### **DEPENDENCIES AND PREREQUISITES**

---

- Android 2.2 (API Level 8) or higher

#### **YOU SHOULD ALSO READ**

---

- [Device Administration](#)

#### **TRY IT OUT**

---

[Download the sample](#)

DeviceManagement.zip

# Enhancing Security with Device Management Policies

Since Android 2.2 (API level 8), the Android platform offers system-level device management capabilities through the Device Administration APIs.

In this lesson, you will learn how to create a security-aware application that manages access to its content by enforcing device management policies. Specifically, the application can be configured such that it ensures a screen-lock password of sufficient strength is set up before displaying restricted content to the user.

## Define and Declare Your Policy

First, you need to define the kinds of policy to support at the functional level. Policies may cover screen-lock password strength, expiration timeout, encryption, etc.

You must declare the selected policy set, which will be enforced by the application, in the `res/xml/device_admin.xml` file. The Android manifest should also reference the declared policy set.

Each declared policy corresponds to some number of related device policy methods in [DevicePolicyManager](#) ([/reference/android/app/admin/DevicePolicyManager.html](#)) (defining minimum password length and minimum number of uppercase characters are two examples). If an application attempts to invoke methods whose corresponding policy is not declared in the XML, this will result in a [SecurityException](#) ([/reference/java/lang/SecurityException.html](#)) at runtime. Other permissions, such as force-lock, are available if the application intends to manage other kinds of policy. As you'll see later, as part of the device administrator activation process, the list of declared policies will be presented to the user on a system screen.

The following snippet declares the limit password policy in `res/xml/device_admin.xml`:

```
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-policies>
        <limit-password />
    </uses-policies>
</device-admin>
```

Policy declaration XML referenced in Android manifest:

```
<receiver android:name=".Policy$PolicyAdmin"
```

### THIS LESSON TEACHES YOU TO

1. [Define and Declare Your Policy](#)
2. [Create a Device Administration Receiver](#)
3. [Activate the Device Administrator](#)
4. [Implement the Device Policy Controller](#)

### YOU SHOULD ALSO READ

- [Device Administration](#)

### TRY IT OUT

[Download the sample](#)

DeviceManagement.zip

```
    android:permission="android.permission.BIND_DEVICE_ADMIN">
    <meta-data android:name="android.app.device_admin"
        android:resource="@xml/device_admin" />
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

## Create a Device Administration Receiver

Create a Device Administration broadcast receiver, which gets notified of events related to the policies you've declared to support. An application can selectively override callback methods.

In the sample application, Device Admin, when the device administrator is deactivated by the user, the configured policy is erased from the shared preference. You should consider implementing business logic that is relevant to your use case. For example, the application might take some actions to mitigate security risk by implementing some combination of deleting sensitive data on the device, disabling remote synchronization, alerting an administrator, etc.

For the broadcast receiver to work, be sure to register it in the Android manifest as illustrated in the above snippet.

```
public static class PolicyAdmin extends DeviceAdminReceiver {

    @Override
    public void onDisabled(Context context, Intent intent) {
        // Called when the app is about to be deactivated as a device administrator
        // Deletes previously stored password policy.
        super.onDisabled(context, intent);
        SharedPreferences prefs = context.getSharedPreferences(APP_PREF, Activity
        prefs.edit().clear().commit();
    }
}
```

## Activate the Device Administrator

Before enforcing any policies, the user needs to manually activate the application as a device administrator. The snippet below illustrates how to trigger the settings activity in which the user can activate your application. It is good practice to include the explanatory text to highlight to users why the application is requesting to be a device administrator, by specifying the [EXTRA\\_ADD\\_EXPLANATION](#) ([/reference/android/app/admin/DevicePolicyManager.html#EXTRA\\_ADD\\_EXPLANATION](#)) extra in the intent.

```
if (!mPolicy.isAdminActive()) {
```

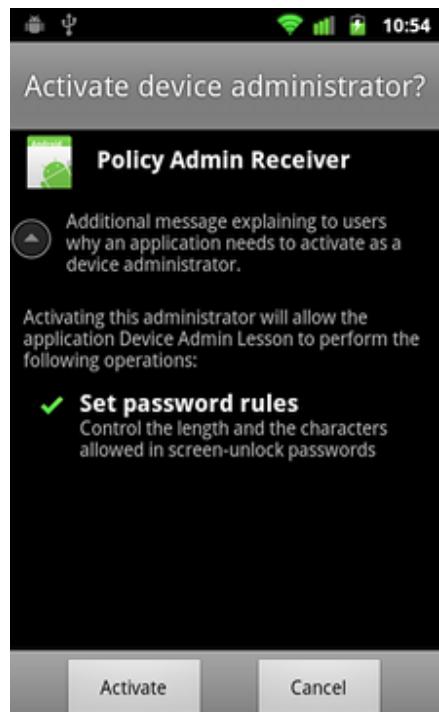
```

Intent activateDeviceAdminIntent =
    new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN)
    .putExtra(
        DevicePolicyManager.EXTRA_DEVICE_ADMIN,
        mPolicy.getPolicyAdmin());
}

// It is good practice to include the optional explanation
// explain to user why the application is requesting
// administrator. The system will display this message
// screen.
activateDeviceAdminIntent.putExtra(
    DevicePolicyManager.EXTRA_ADD_EXPLANATION,
    getResources().getString(R.string.device_admin_explanation));

startActivityForResult(activateDeviceAdminIntent,
    REQ_ACTIVATE_DEVICE_ADMIN);
}

```



If the user chooses "Activate," the application becomes a device administrator and can begin configuring and enforcing the policy.

The application also needs to be prepared to handle set back situations where the user abandons the activation process by hitting the Cancel button, the Back key, or the Home key. Therefore, [onResume\(\) /reference android/app/Activity.html#onResume\(\)](#) in the Policy Set Up Activity needs to have logic to reevaluate the condition and present the Device Administrator Activation option to the user if needed.

## Implement the Device Policy Controller

After the device administrator is activated successfully, the application then configures Device Policy Manager with the requested policy. Keep in mind that new policies are being added to Android with each release. It is appropriate to perform version checks in your application if using new policies while supporting older versions of the platform. For example, the Password Minimum Upper Case policy is only available with API level 11 (Honeycomb) and above. The following code demonstrates how you can check the version at runtime.

```

DevicePolicyManager mDPM = (DevicePolicyManager)
    context.getSystemService(Context.DEVICE_POLICY_SERVICE);
ComponentName mPolicyAdmin = new ComponentName(context, PolicyAdmin.class);
...
mDPM.setPasswordQuality(mPolicyAdmin, PASSWORD_QUALITY_VALUES[mPasswordQuality]);
mDPM.setPasswordMinimumLength(mPolicyAdmin, mPasswordLength);
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    mDPM.setPasswordMinimumUpperCase(mPolicyAdmin, mPasswordMinUpperCase);
}

```

At this point, the application is able to enforce the policy. While the application has no access to the actual screen-lock password used, through the Device Policy Manager API it can determine whether the existing password satisfies the required policy. If it turns out that the existing screen-lock password is not sufficient, the device administration API does not automatically take corrective action. It is the application's responsibility to explicitly launch the system password-change screen in the Settings app. For example:

```
if (!mDPM.isActivePasswordSufficient()) {  
    ...  
    // Triggers password change screen in Settings.  
    Intent intent =  
        new Intent(DevicePolicyManager.ACTION_SET_NEW_PASSWORD);  
    startActivity(intent);  
}
```

Normally, the user can select from one of the available lock mechanisms, such as None, Pattern, PIN (numeric), or Password (alphanumeric). When a password policy is configured, those password types that are weaker than those defined in the policy are disabled. For example, if the "Numeric" password quality is configured, the user can select either PIN (numeric) or Password (alphanumeric) password only.

Once the device is properly secured by setting up a proper screen-lock password, the application allows access to the secured content.

```
if (!mDPM.isAdminActive(..)) {  
    // Activates device administrator.  
    ...  
} else if (!mDPM.isActivePasswordSufficient()) {  
    // Launches password set-up screen in Settings.  
    ...  
} else {  
    // Grants access to secure content.  
    ...  
    startActivity(new Intent(context, SecureActivity.class));  
}
```

# Monetizing Your App

Apart from offering paid apps, there are a number of other ways to monetize your mobile applications. In this class, we are going to examine a number of typical methods (more lessons are to come) and their associated technical best practices. Obviously, each application is different and you should experiment with different combinations of these and other monetization methods to determine what works best for you.

## Lessons

---

### [Advertising without Compromising User Experience](#)

In this lesson, you will learn how to monetize your application with mobile advertisements.

#### **DEPENDENCIES AND PREREQUISITES**

---

- Android 1.0 or higher
- Experience with [XML layouts](#)

#### **TRY IT OUT**

---

[Download the sample app](#)

MobileAds.zip

# Advertising without Compromising User Experience

Advertising is one of the means to monetize (make money with) mobile applications. In this lesson, you are going to learn how to incorporate banner ads in your Android application.

While this lesson and the sample application use [AdMob](#) (<http://code.google.com/mobile/ads/>) to serve ads, the Android platform doesn't impose any restrictions on the choice of mobile advertising network. To the extent possible, this lesson generically highlights concepts that are similar across advertising networks.

For example, each advertising network may have some network-specific configuration settings such as geo-targeting and ad-text font size, which may be configurable on some networks but not on others. This lesson does not touch not these topics in depth and you should consult documentation provided by the network you choose.

## Obtain a Publisher Account and Ad SDK

In order to integrate advertisements in your application, you first must become a publisher by registering a publishing account with the mobile advertising network. Typically, an identifier is provisioned for each application serving advertisements. This is how the advertising network correlates advertisements served in applications. In the case of AdMob, the identifier is known as the Publisher ID. You should consult your advertising networks for details.

Mobile advertising networks typically distribute a specific Android SDK, which consists of code that takes care of communication, ad refresh, look-and-feel customization, and so on.

Most advertising networks distribute their SDK as a JAR file. Setting up ad network JAR file in your Android project is no different from integrating any third-party JAR files. First, copy the JAR files to the `libs/` directory of your project. If you're using Eclipse as IDE, be sure to add the JAR file to the Build Path. It can be done through **Properties > Java Build Path > Libraries > Add JARs**.

### THIS LESSON TEACHES YOU TO

1. [Obtain a Publisher Account and Ad SDK](#)
2. [Declare Proper Permissions](#)
3. [Set Up Ad Placement](#)
4. [Initialize the Ad](#)
5. [Enable Test Mode](#)
6. [Implement Ad Event Listeners](#)

### YOU SHOULD ALSO READ

- [AdMob SDK](#)

### TRY IT OUT

[Download the sample app](#)

MobileAds.zip

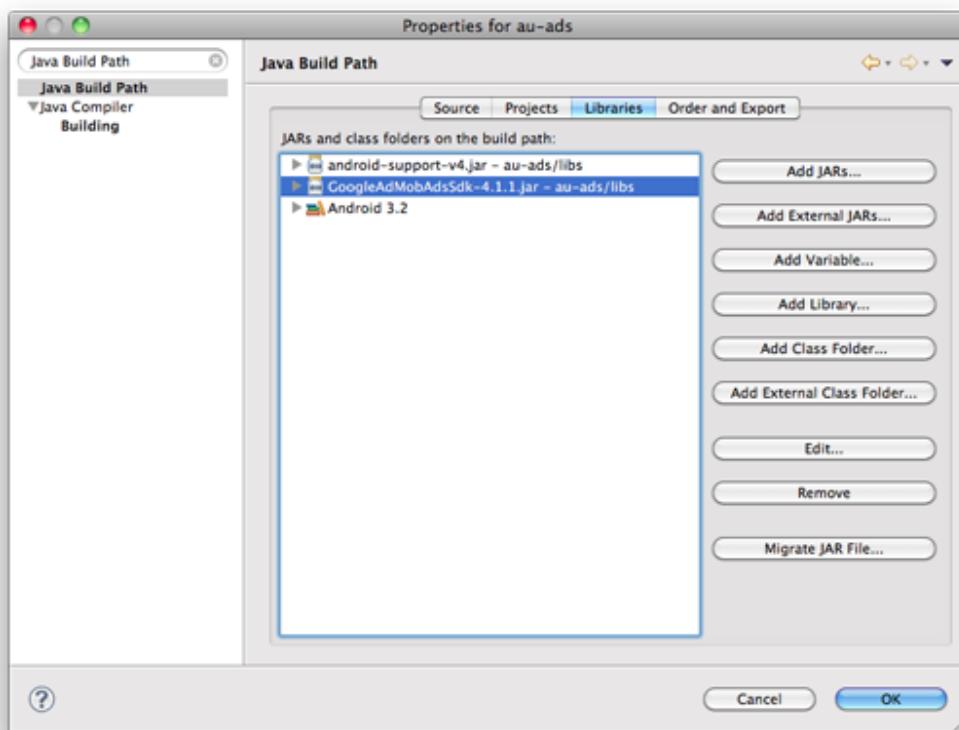


Figure 1. Eclipse build path settings.

## Declare Proper Permissions

Because the mobile ads are fetched over the network, mobile advertising SDKs usually require the declaration of related permissions in the Android manifest. Other kinds of permissions may also be required.

For example, here's how you can request the [INTERNET \(/reference/android/Manifest.permission.html#INTERNET\)](#) permission:

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

## Set Up Ad Placement

Banner ads typically are implemented as a custom [WebView \(/reference/android/webkit/WebView.html\)](#) (a view for viewing web pages). Ads also come in different dimensions and shapes. Once you've decided to put an ad on a particular screen, you can add it in your activity's XML layout. The XML snippet below illustrates a banner ad displayed on top of a screen.

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/ad_catalog_layout"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <com.google.ads.AdView
        xmlns:googleads="http://schemas.android.com/apk/res/com.google.ads"
        android:id="@+id/ad"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        googleads:adSize="BANNER"
        googleads:adUnitId="@string/admob_id" />
    <TextView android:id="@+id/title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/banner_top" />
    <TextView android:id="@+id/status"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>

```

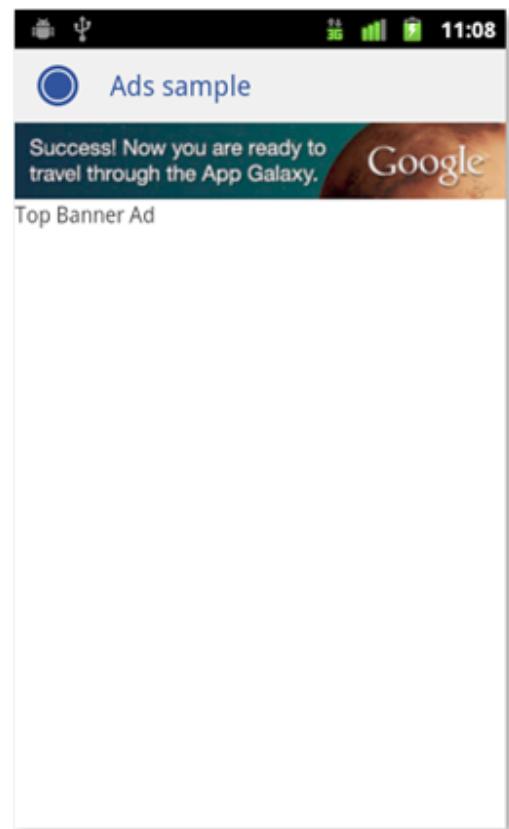


Figure 2. Screenshot of the ad layout in the Mobile Ads sample.

You should consider using alternative ad sizes based on various configurations such as screen size or screen orientation. This can easily be addressed by providing [alternative resources](#) ([/guide/topics/resources/providing-resources.html#AlternativeResources](#)). For instance, the above sample layout might placed under the `res/layout/` directory as the default layout. If larger ad sizes are available, you can consider using them for "large" (and above) screens. For example, the following snippet comes from a layout file in the `res/layout-large/` directory, which renders a larger ad for "large" screen sizes.

```

...
<com.google.ads.AdView
    xmlns:googleads="http://schemas.android.com/apk/res/com.google.ads"
    android:id="@+id/ad"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    googleads:adSize="IAB_LEADERBOARD"
    googleads:adUnitId="@string/admob_id" />
...

```

Notice that the custom view name and it's configuration attributes are network-specific. Ad networks might support configurations with XML layout attributes (as shown above), runtime APIs, or both. In the sample application, Mobile Ads, the AdView ad size (`googleads:adSize`) and publisher ID (`googleads:adUnitId`) are set up in the XML layout.

When deciding where to place ads within your application, you should carefully consider user-experience. For example, you don't want to fill the screen with multiple ads that will quite

likely annoy your users. In fact, this practice is banned by some ad networks. Also, avoid placing ads too closely to UI controls to avoid inadvertent clicks.

Figures 3 and 4 illustrate what not to do.

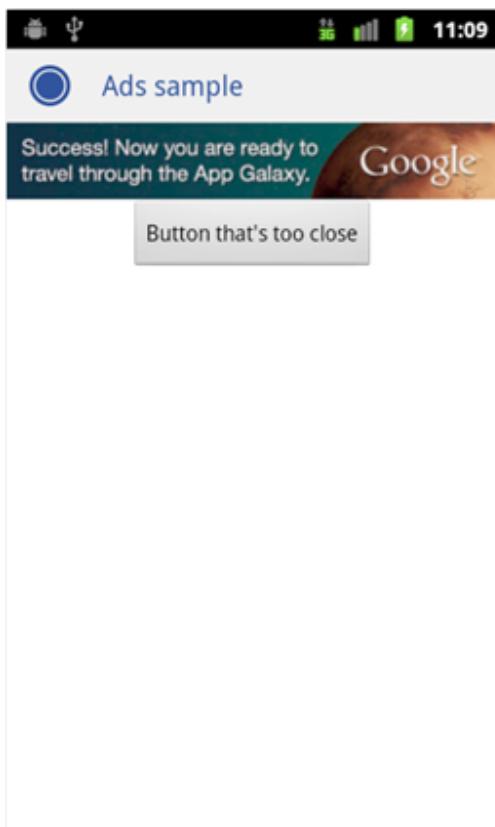


Figure 3. Avoid putting UI inputs too closely to an ad banner to prevent inadvertent ad clicks.

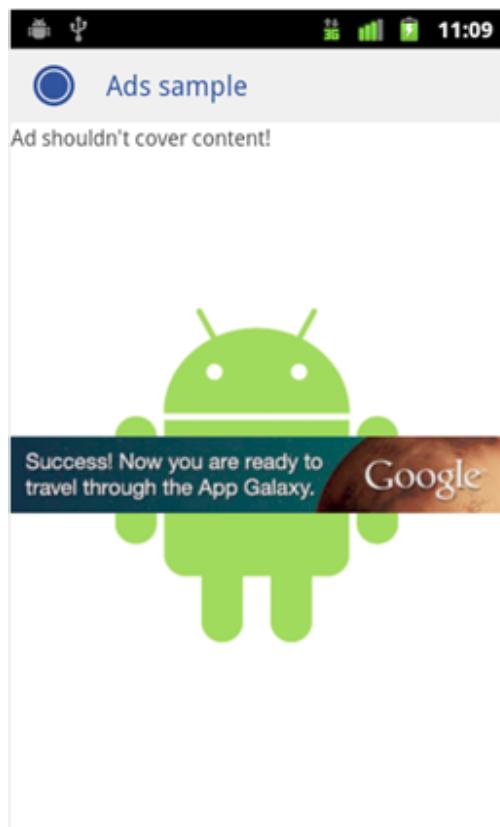


Figure 4. Don't overlay ad banner on useful content.

## Initialize the Ad

---

After setting up the ad in the XML layout, you can further customize the ad in `Activity.onCreate()` ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) or `Fragment.onCreateView()` ([/reference/android/app/Fragment.html#onCreateView\(android.view.LayoutInflater, android.view.ViewGroup, android.os.Bundle\)](#)) based on how your application is architected. Depending on the ad network, possible configuration parameters are: ad size, font color, keyword, demographics, location targeting, and so on.

It is important to respect user privacy if certain parameters, such as demographics or location, are passed to ad networks for targeting purposes. Let your users know and give them a chance to opt out of these features.

In the below code snippet, keyword targeting is used. After the keywords are set, the application calls `loadAd()` to begin serving ads.

```

public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    ...
    View v = inflater.inflate(R.layout.main, container, false);
    mAdStatus = (TextView) v.findViewById(R.id.status);
    mAdView = (AdView) v.findViewById(R.id.ad);
    mAdView.setAdListener(new MyAdListener());
    AdRequest adRequest = new AdRequest();
    adRequest.addKeyword("sporting goods");
    mAdView.loadAd(adRequest);
    return v;
}

```

## Enable Test Mode

Some ad networks provide a test mode. This is useful during development and testing in which ad impressions and clicks are not counted.

**Important:** Be sure to turn off test mode before publishing your application.

## Implement Ad Event Listeners

Where available, you should consider implementing ad event listeners, which provide callbacks on various ad-serving events associated with the ad view. Depending on the ad network, the listener might provide notifications on events such as before the ad is loaded, after the ad is loaded, whether the ad fails to load, or other events. You can choose to react to these events based on your specific situation. For example, if the ad fails to load, you can display a custom banner within the application or create a layout such that the rest of content fills up the screen.

For example, here are some event callbacks available from AdMob's AdListener interface:

```

private class MyAdListener implements AdListener {
    ...
    @Override
    public void onFailedToReceiveAd(Ad ad, ErrorCode errorCode) {
        mAdStatus.setText(R.string.error_receive_ad);
    }
    @Override
    public void onReceiveAd(Ad ad) {
        mAdStatus.setText("");
    }
}

```

# Designing Effective Navigation

One of the very first steps to designing and developing an Android application is to determine what users are able to see and do with the app. Once you know what kinds of data users are interacting with in the app, the next step is to design the interactions that allow users to navigate across, into, and back out from the different pieces of content within the app.

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. Each lesson covers various stages in the interaction design process for navigation in Android applications, in roughly chronological order. After going through the lessons in this class, you should be able to apply the methodology and navigation paradigms outlined here to your own applications, providing a coherent navigation experience for your users.

## DEPENDENCIES AND PREREQUISITES

This class is not specific to any particular version of the Android platform. It is also primarily design-focused and does not require knowledge of the Android SDK. That said, you should have experience using an Android device for a better understanding of the context in which Android applications run.

You should also have basic familiarity with the Action Bar ([pattern docs \(/design/patterns/actionbar.html\)](#) at [Android Design](#)), used across most applications in devices running Android 3.0 and later.

## Lessons

### Planning Screens and Their Relationships

Learn how to choose which screens your application should contain. Also learn how to choose which screens should be directly reachable from others. This lesson introduces a hypothetical news application to serve as an example for later lessons.

### Planning for Multiple Touchscreen Sizes

Learn how to group related screens together on larger-screen devices to optimize use of available screen space.

### Providing Descendant and Lateral Navigation

Learn about techniques for allowing users to navigate deep into, as well as across, your content hierarchy. Also learn about pros and cons of, and best practices for, specific navigational UI elements for various situations.

### Providing Ancestral and Temporal Navigation

Learn how to allow users to navigate upwards in the content hierarchy. Also learn about best practices for the *Back* button and temporal navigation, or navigation to previous screens that may not be hierarchically related.

### Putting it All Together: Wireframing the Example App

Learn how to create screen wireframes (low-fidelity graphic mockups) representing the screens in a news application based on the desired information model. These wireframes utilize navigational elements discussed in previous lessons to demonstrate intuitive and efficient navigation.

# Planning Screens and Their Relationships

Most apps have an inherent information model that can be expressed as a tree or graph of object types. In more obvious terms, you can draw a diagram of different kinds of information that represents the types of things users interact with in your app. Software engineers and data architects often use entity-relationship diagrams (ERDs) to describe an application's information model.

## THIS LESSON TEACHES YOU TO

1. [Create a Screen List](#)
2. [Diagram Screen Relationships](#)
3. [Go Beyond a Simplistic Design](#)

Let's consider an example application that allows users to browse through a set of categorized news stories and photos. One possible model for such an app is shown below in the form of an ERD.

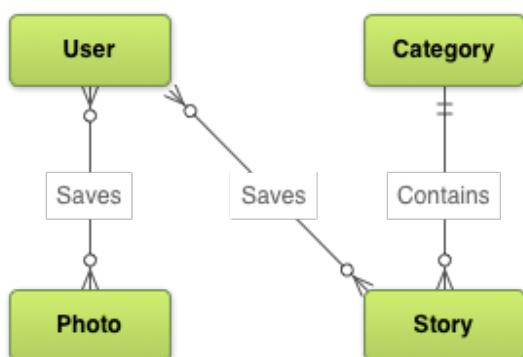


Figure 1. Entity-relationship diagram for the example news application.

## Create a Screen List

Once you define the information model, you can begin to define the contexts necessary to enable users to effectively discover, view, and act upon the data in your application. In practice, one way to do this is to *determine the exhaustive set of screens* needed to allow users to navigate to and interact with the data. The set of screens we actually expose should generally vary depending on the target device; it's important to consider this early in the design process to ensure that the application can adapt to its environment.

In our example application, we want to enable users to view, save, and share *categorized* stories and photos. Below is an exhaustive list of screens that covers these use cases.

- Home or "launchpad" screen for accessing stories and photos
- List of categories
- List of news stories for a given category
- Story detail view (from which we can save and share)
- List of photos, uncategorized
- Photo detail view (from which we can save and share)
- List of all saved items
- List of saved photos

- List of saved stories

## Diagram Screen Relationships

---

Now we can define the directed relationships between screens; an arrow from one screen *A* to another screen *B* implies that screen *B* should be directly reachable via some user interaction in screen *A*. Once we define both the set of screens and the relationships between them, we can express these in concert as a screen map, which shows all of your screens and their relationships:

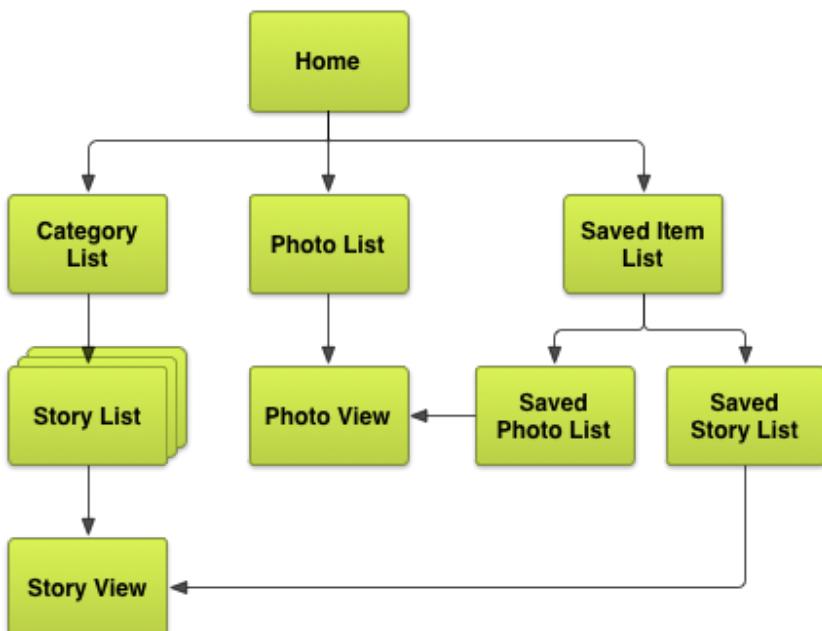


Figure 2. Exhaustive screen map for the example news application.

If we later wanted to allow users to submit news stories or upload photos, we could add additional screens to this diagram.

## Go Beyond a Simplistic Design

---

At this point, it's possible to design a completely functional application from this exhaustive screen map. A simplistic user interface could consist of lists and buttons leading to child screens:

- Buttons leading to different sections (e.g., stories, photos, saved items)
- Vertical lists representing collections (e.g., story lists, photo lists, etc.)
- Detail information (e.g., story view, full-screen photo view, etc.)

However, you can use screen grouping techniques and more sophisticated navigation elements to present content in a more intuitive and device-sensitive way. In the next lesson, we explore screen grouping techniques, such as providing multi-pane layouts for tablet devices. Later, we'll dive into the various navigation patterns common on Android.

# Planning for Multiple Touchscreen Sizes

The exhaustive screen map from the previous lesson isn't tied to a particular device form factor, although it can generally look and work okay on a handset or similar-size device. But Android applications need to adapt to a number of different types of devices, from 3" handsets to 10" tablets to 42" TVs. In this lesson we explore reasons and tactics for grouping together multiple screens from the exhaustive map.

Note: Designing applications for television sets also requires attention to other factors, including interaction methods (i.e., the lack of a touch screen), legibility of text at large reading distances, and more. Although this discussion is outside the scope of this class, you can find more information on designing for TVs in the [Google TV](https://developers.google.com/tv/) (<https://developers.google.com/tv/>) documentation for [design patterns](#) ([https://developers.google.com/tv/android/docs/gtv\\_android\\_patterns](https://developers.google.com/tv/android/docs/gtv_android_patterns)).

## THIS LESSON TEACHES YOU TO

1. [Group Screens with Multi-pane Layouts](#)
2. [Design for Multiple Tablet Orientations](#)
3. [Group Screens in the Screen Map](#)

## YOU SHOULD ALSO READ

- [Android Design: Multi-pane Layouts](#)
- [Designing for Multiple Screens](#)

## Group Screens with Multi-pane Layouts

### Multi-pane Layout Design

For design guidelines, read Android Design's [Multi-pane Layouts](/design/patterns/multi-pane-layouts.html) (</design/patterns/multi-pane-layouts.html>) pattern guide.

3 to 4-inch screens are generally only suitable for showing a single vertical pane of content at a time, be it a list of items, or detail information about an item, etc. Thus on such devices, screens generally map one-to-one with levels in the information hierarchy (*categories* → *object list* → *object detail*).

Larger screens such as those found on tablets and TVs, on the other hand, generally have much more available screen space and are able to present multiple panes of content. In landscape, panes are usually ordered from left to right in increasing detail order. Users are especially accustomed to multiple panes on larger screens from years and years of desktop application and desktop web site use. Many desktop applications and websites offer a left-hand navigation pane or use a master/detail two-pane layout.

In addition to addressing these user expectations, it's usually necessary to provide multiple panes of information on tablets to avoid leaving too much whitespace or unwittingly introducing awkward interactions, for example 10 x 0.5-inch buttons.

The following figures demonstrate some of the problems that can arise when moving a UI (user interface) design into a larger layout and how to address these issues with multi-pane layouts:



Figure 1. Single pane layouts on large screens in landscape lead to awkward whitespace and exceedingly long line lengths.

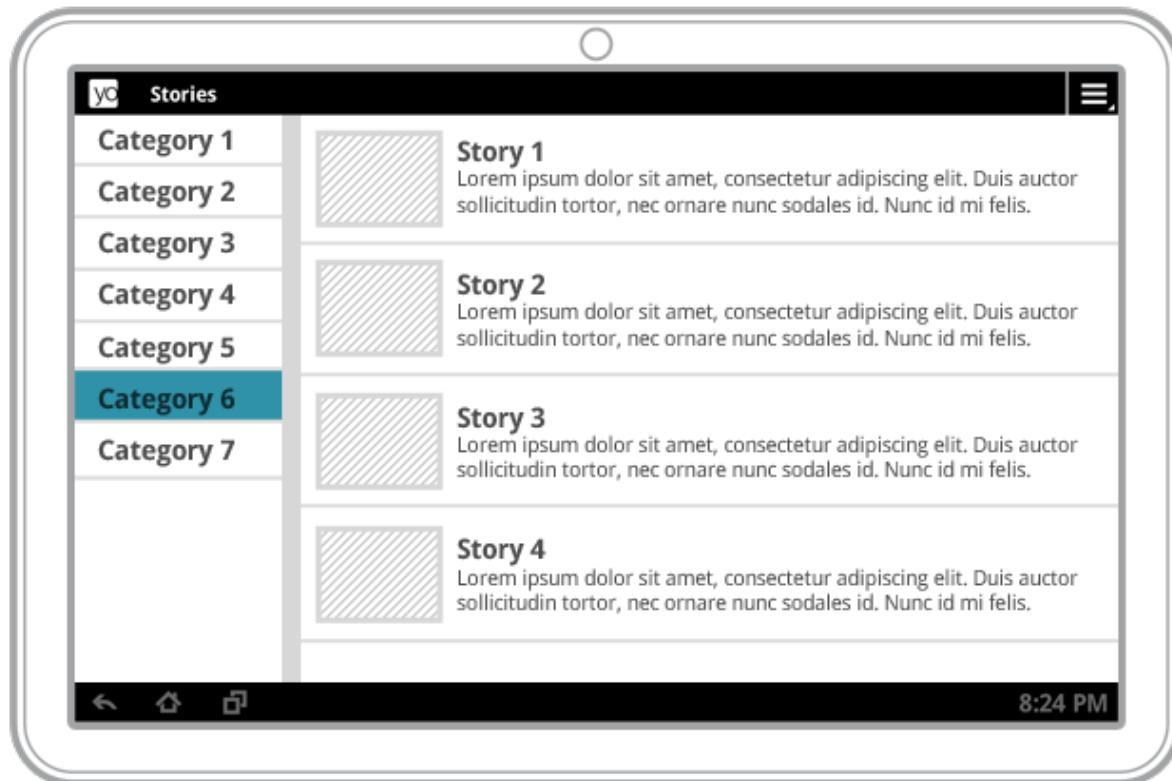


Figure 2. Multi-pane layouts in landscape result in a better visual balance while offering more utility and legibility.

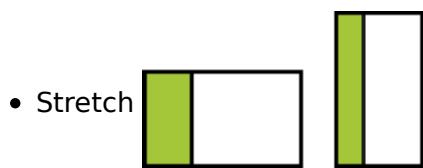
**Implementation Note:** After deciding on the screen size at which to draw the line between single-pane and multi-pane layouts, you can provide different layouts containing one or multiple panes for devices in varying screen size buckets (such as `large/xlarge`) or varying minimum screen widths (such as `sw600dp`).

**Implementation Note:** While a single screen is implemented as an [`Activity`](#) ([/reference/android/app/Activity.html](#)) subclass, individual content panes can be implemented as [`Fragment`](#) ([/reference/android/app/Fragment.html](#)) subclasses. This maximizes code re-use across different form factors and across screens that share content.

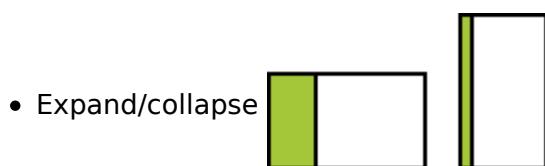
## Design for Multiple Tablet Orientations

Although we haven't begun arranging user interface elements on our screens yet, this is a good time to consider how your multi-pane screens will adapt to different device orientations. Multi-pane layouts in landscape work quite well because of the large amount of available horizontal space. However, in the portrait orientation, your horizontal space is more limited, so you may need to design a separate layout for this orientation.

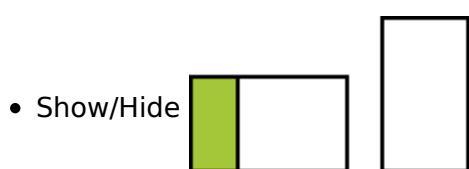
Below are a few common strategies for creating portrait tablet layouts.



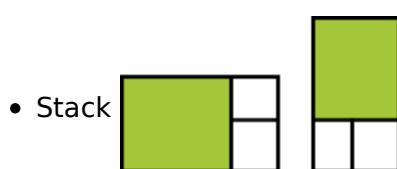
The most straightforward strategy is to simply stretch each pane's width to best present the content in each pane in the portrait orientation. Panes could have fixed widths or take a certain percentage of the available screen width.



A variation on the stretch strategy is to collapse the contents of the left pane when in portrait. This works quite well with master/detail panes where the left (master) pane contains easily collapsible list items. An example would be for a realtime chat application. In landscape, the left list could contain chat contact photos, names, and online statuses. In portrait, horizontal space could be collapsed by hiding contact names and only showing photos and online status indicator icons. Optionally also provide an expand control that allows the user to expand the left pane content to its larger width and vice versa.



In this scenario, the left pane is completely hidden in portrait mode. However, *to ensure the functional parity* of your screen in portrait and landscape, the left pane should be made available via an onscreen affordance (such as a button). It's usually appropriate to use the *Up* button in the Action Bar ([pattern docs \(/design/patterns/actionbar.html\)](#) at Android Design) to show the left pane, as is discussed in a later lesson ([ancestral-temporal.html](#)).



The last strategy is to vertically stack your normally horizontally-arranged panes in portrait. This strategy works well when your panes aren't simple text-based lists, or when there are multiple blocks of content running along the primary content pane. Be careful to avoid the awkward whitespace problem discussed above when using this strategy.

## Group Screens in the Screen Map

---

Now that we are able to group individual screens together by providing multi-pane layouts on

larger-screen devices, let's apply this technique to our exhaustive screen map from the [previous lesson \(screen-planning.html\)](#) to get a better sense of our application's hierarchy on such devices:

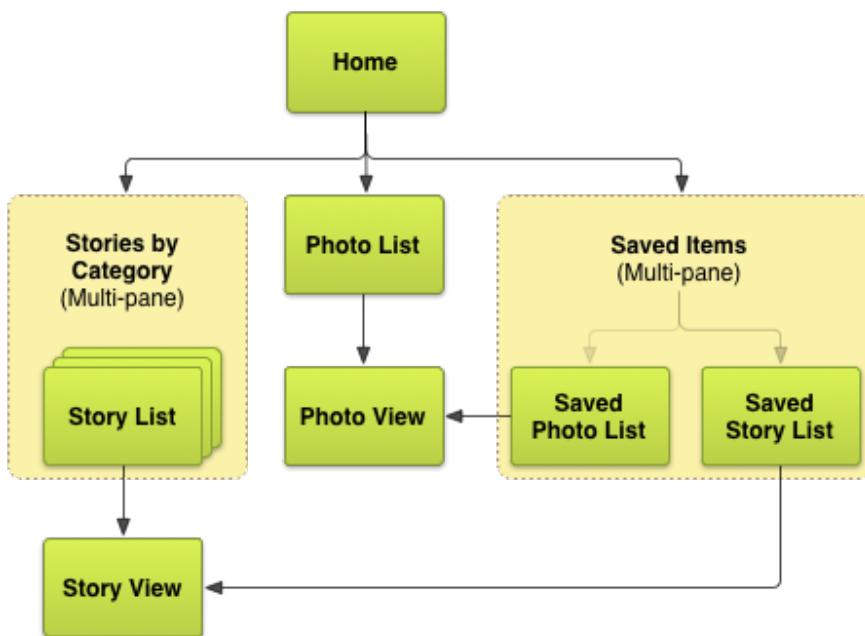


Figure 3. Updated example news application screen map for tablets.

In the next lesson we discuss *descendant* and *lateral* navigation, and explore more ways of grouping screens to maximize the intuitiveness and speed of content access in the application's user interface.

# Providing Descendant and Lateral Navigation

One way of providing access to the full range of an application's screens is to expose hierarchical navigation. In this lesson we discuss *descendant navigation*, allowing users to descend 'down' a screen hierarchy into a child screen, and *lateral navigation*, allowing users to access sibling screens.

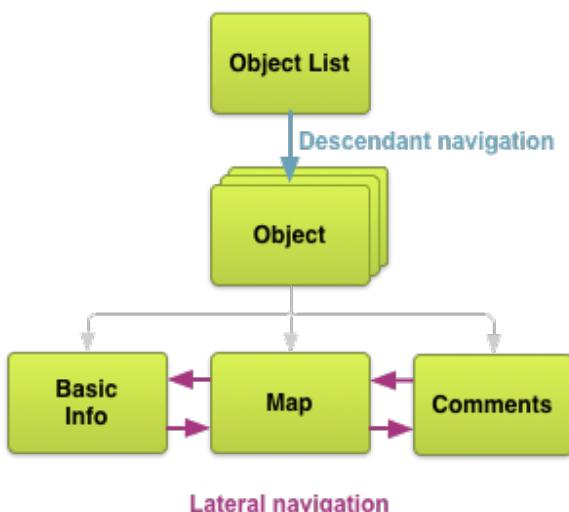


Figure 1. Descendant and lateral navigation.

## THIS LESSON TEACHES YOU ABOUT:

1. [Buttons and Simple Targets](#)
2. [Lists, Grids, Carousels, and Stacks](#)
3. [Tabs](#)
4. [Horizontal Paging \(Swipe Views\)](#)

## YOU SHOULD ALSO READ

- [Android Design: Buttons](#)
- [Android Design: Lists](#)
- [Android Design: Grid Lists](#)
- [Android Design: Tabs](#)
- [Android Design: Swipe Views](#)

There are two types of sibling screens: collection-related and section-related screens. *Collection-related* screens represent individual items in the collection represented by the parent. *Section-related* screens represent different sections of information about the parent. For example, one section may show textual information about an object while another may provide a map of the object's geographic location. The number of section-related screens for a given parent is generally small.

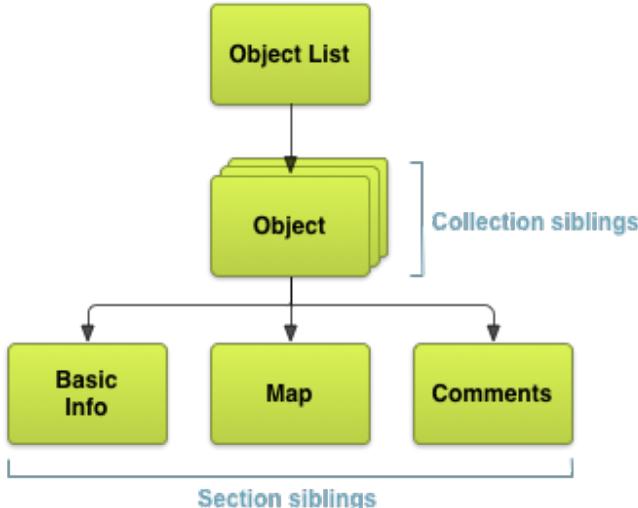


Figure 2. Collection-related children and section-related children.

Descendant and lateral navigation can be provided using lists, tabs, and other user interface patterns. *User interface patterns*, much like software design patterns, are generalized, common solutions to recurring interaction design problems. We explore a few common lateral navigation patterns in the sections below.

## Buttons and Simple Targets

### Button Design

For design guidelines, read Android Design's [Buttons \(/design/building-blocks/buttons.html\)](#) guide.

For section-related screens, offering touchable and keyboard-focusable targets in the parent is generally the most straightforward and familiar kind of touch-based navigation interface. Examples of such targets include buttons, fixed-size list views, or text links, although the latter is not an ideal UI (user interface) element for touch-based navigation. Upon selecting one of these targets, the child screen is opened, replacing the current context (screen) entirely. Buttons and other simple targets are rarely used for representing items in a collection.

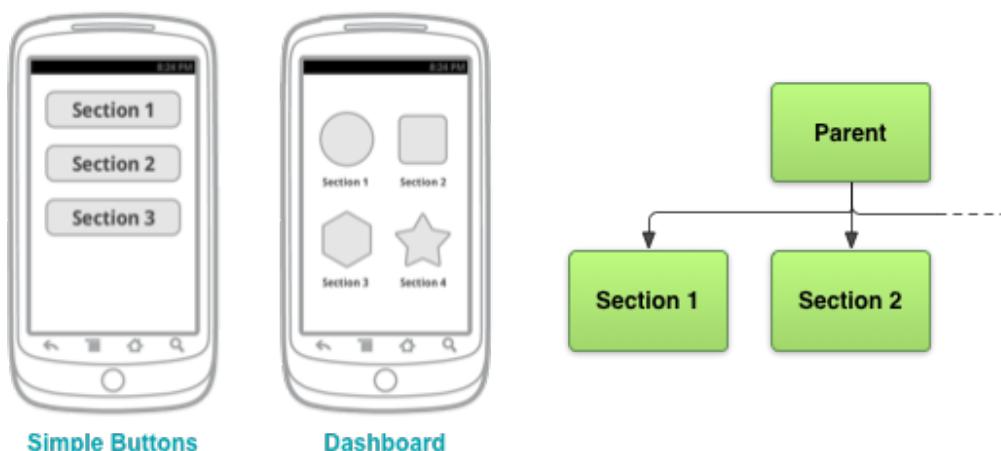


Figure 3. Example button-based navigation interface with relevant screen map excerpt. Also shows dashboard pattern discussed below.

A common, button-based pattern for accessing different top-level application sections, is the dashboard pattern. A *dashboard* is a grid of large, iconic buttons that constitutes the entirety, or most of, the parent screen. The grid generally has either 2 or 3 rows and columns, depending on the number of top-level sections in the app. This pattern is a great way to present all the sections of the app in a visually rich way. The large touch targets also make this UI very easy to use. Dashboards are best used when each section is equally important, as determined by product decisions or better yet, real-world usage. However, this pattern doesn't visually work well on larger screens, and requires users to take an extra step to jump directly into the app's content.

More sophisticated user interfaces can make use of a variety of other user interaction patterns to improve content immediacy and presentation uniqueness, all the while remaining intuitive.

## Lists, Grids, Carousels, and Stacks

## List and Grid List Design

For design guidelines, read Android Design's [Lists \(/design/building-blocks/lists.html\)](#) and [Grid Lists \(/design/building-blocks/grid-lists.html\)](#) guides.

For collection-related screens, and especially for textual information, vertically scrolling lists are often the most straightforward and familiar kind of interface. For more visual or media-rich content items such as photos or videos, vertically scrolling grids of items, horizontally scrolling lists (sometimes referred to as *carousels*), or stacks (sometimes referred to as *cards*) can be used instead. These UI elements are generally best used for presenting item collections or large sets of child screens (for example, a list of stories or a list of 10 or more news topics), rather than a small set of unrelated, sibling child screens.

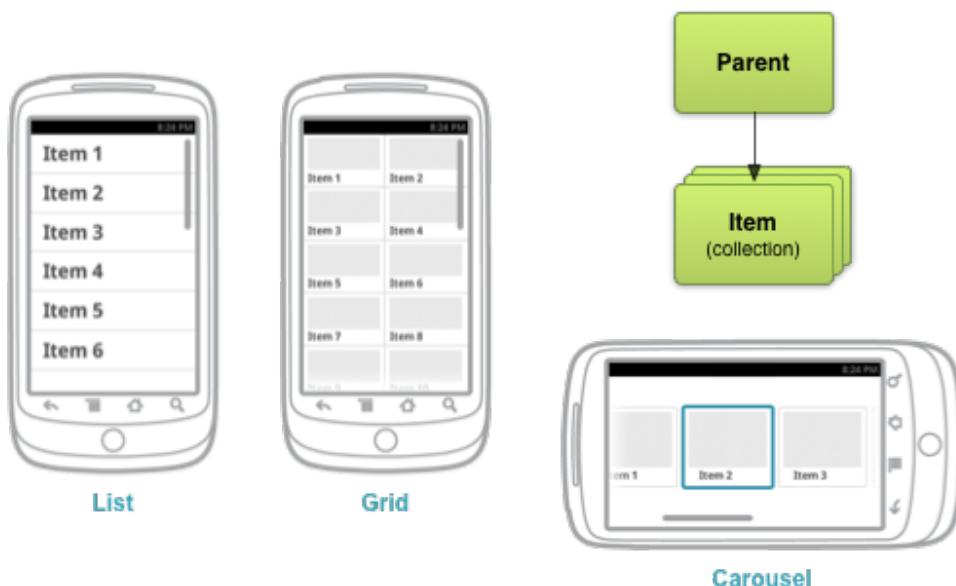


Figure 4. Example list-, grid-, and carousel-based navigation interfaces with relevant screen map excerpt.

There are several issues with this pattern. Deep, list-based navigation, known as *drill-down list navigation*, where lists lead to more lists which lead to even more lists, is often inefficient and cumbersome. The number of touches required to access a piece of content with this kind of navigation is generally very high, leading to a poor user experience—especially for users on-the-go.

Using vertical lists can also lead to awkward user interactions and poor use of whitespace on larger screens, as list items generally span the entire width of the screen yet have a fixed height. One way to alleviate this is to provide additional information, such as text summaries, that fills the available horizontal space. Another way is to provide additional information in a separate horizontal pane adjacent to the list.

## Tabs

### Tab Design

For design guidelines, read Android Design's [Tabs \(/design/building-blocks/tabs.html\)](#) guide.

Using tabs is a very popular solution for lateral navigation. This pattern allows grouping of sibling

screens, in that the tab content container in the parent screen can embed child screens that otherwise would be entirely separate contexts. Tabs are most appropriate for small sets (4 or fewer) of section-related screens.

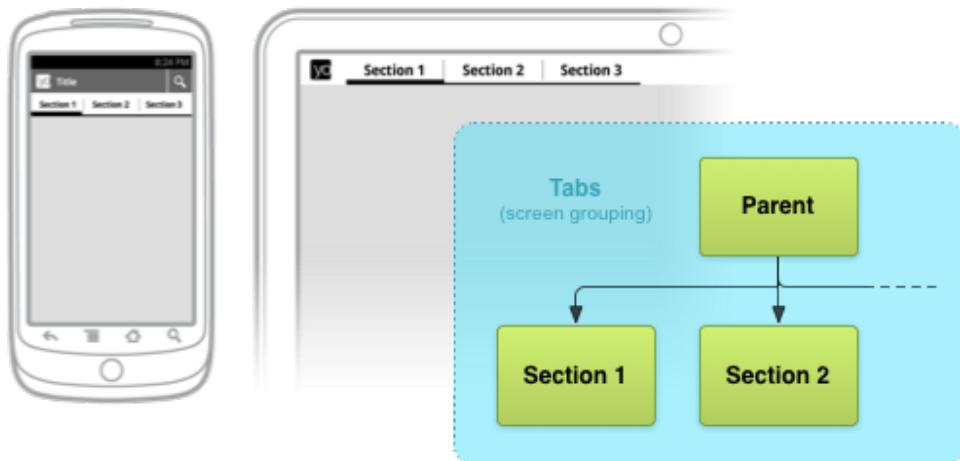


Figure 5. Example phone and tablet tab-based navigation interfaces with relevant screen map excerpt.

Several best practices apply when using tabs. Tabs should be persistent across immediate related screens. Only the designated content region should change when selecting a tab, and tab indicators should remain available at all times. Additionally, tab switches should not be treated as history. For example, if a user switches from a tab A to another tab B, pressing the *Back* button (more on that in the [next lesson \(ancestral-temporal.html\)](#)) should not re-select tab A. Tabs are usually laid out horizontally, although other presentations of tab navigation such as using a drop-down list in the Action Bar ([pattern docs \(/design/patterns/actionbar.html\)](#) at Android Design) are sometimes appropriate. Lastly, and most importantly, *tabs should always run along the top of the screen*, and should not be aligned to the bottom of the screen.

There are some obvious immediate benefits of tabs over simpler list- and button-based navigation:

- Since there is a single, initially-selected tab, users have immediate access to that tab's content from the parent screen.
- Users can navigate quickly between related screens, without needing to first revisit the parent.

**Note:** when switching tabs, it is important to maintain this tab-switching immediacy; do not block access to tab indicators by showing modal dialogs while loading content.

A common criticism is that space must be reserved for the tab indicators, detracting from the space available to tab contents. This consequence is usually acceptable, and the tradeoff commonly weighs in favor of using this pattern. You should also feel free to customize tab indicators, showing text and/or icons to make optimal use of vertical space. When adjusting indicator heights however, ensure that tab indicators are large enough for a human finger to touch without error.

## Horizontal Paging (Swipe Views)

Swipe Views Design

For design guidelines, read Android Design's [Swipe Views \(/design/patterns/swipe-views.html\)](#) pattern guides.

Another popular lateral navigation pattern is horizontal paging, also referred to as swipe views. This pattern applies best to collection-related sibling screens, such as a list of categories (world, business, technology, and health stories). Like tabs, this pattern also allows grouping screens in that the parent presents the contents of child screens embedded within its own layout.

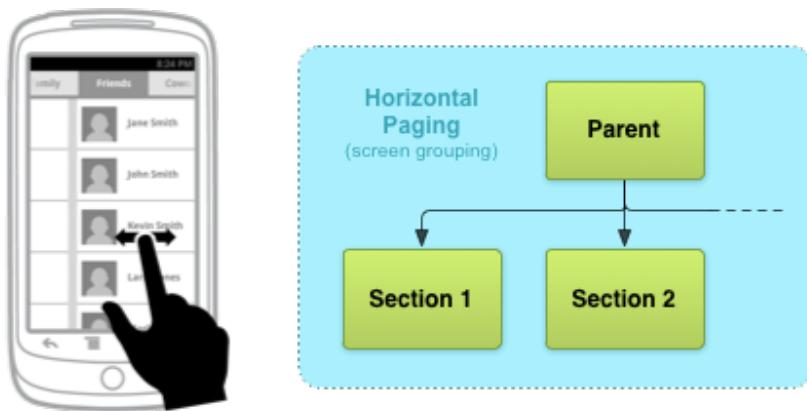


Figure 6. Example horizontal paging navigation interface with relevant screen map excerpt.

In a horizontal paging UI, a single child screen (referred to as a *page* here) is presented one at a time. Users are able to navigate to sibling screens by touching and dragging the screen horizontally in the direction of the desired adjacent page. This gestural interaction is often complemented by another UI element indicating the current page and available pages, to aid discoverability and provide more context to the user. This practice is especially necessary when using this pattern for lateral navigation of section-related sibling screens. Examples of such elements include tick marks, scrolling labels, and tabs.

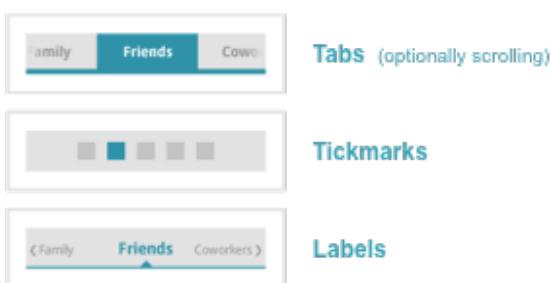


Figure 7. Example paging companion UI elements.

It's also best to avoid this pattern when child screens contain horizontal panning surfaces (such as maps), as these conflicting interactions may deter your screen's usability.

Additionally, for sibling-related screens, horizontal paging is most appropriate where there is some similarity in content type and when the number of siblings is relatively small. In these cases, this pattern can be used along with tabs above the content region to maximize the interface's intuitiveness. For collection-related screens, horizontal paging is most intuitive when there is a natural ordered relationship between screens, for example if each page represents consecutive

calendar days. For infinite collections (again, calendar days), especially those with content in both directions, this paging mechanism can work quite well.

In the next lesson, we discuss mechanisms for allowing users to navigate up our information hierarchy and back, to previously visited screens.

# Providing Ancestral and Temporal Navigation

Now that users can navigate [deep into \(descendant-lateral.html\)](#) the application's screen hierarchy, we need to provide a method for navigating up the hierarchy, to parent and ancestor screens. Additionally, we should ensure that temporal navigation via the *Back* button is respected to respect Android conventions.

## Back/Up Navigation Design

For design guidelines, read Android Design's [Navigation \(/design/patterns/navigation.html\)](#) pattern guide.

### THIS LESSON TEACHES YOU TO:

1. [Support Temporal Navigation: Back](#)
2. [Provide Ancestral Navigation: Up and Home](#)

### YOU SHOULD ALSO READ

- [Android Design: Navigation](#)
- [Tasks and Back Stack](#)

## Support Temporal Navigation: Back

Temporal navigation, or navigation between historical screens, is deeply rooted in the Android system. All Android users expect the *Back* button to take them to the previous screen, regardless of other state. The set of historical screens is always rooted at the user's Launcher application (the phone's "home" screen). That is, pressing *Back* enough times should land you back at the Launcher, after which the *Back* button will do nothing.



Figure 1. The *Back* button behavior after entering the Email app from the People (or Contacts) app.

Applications generally don't have to worry about managing the *Back* button themselves; the system handles [tasks and the back stack \(/guide/components/tasks-and-back-stack.html\)](#), or the list of previous screens, automatically. The *Back* button by default simply traverses this list of screens, removing the current screen from the list upon being pressed.

There are, however, cases where you may want to override the behavior for *Back*. For example, if your screen contains an embedded web browser where users can interact with page elements to navigate between web pages, you may wish to trigger the embedded browser's default *back* behavior when users press the device's *Back* button. Upon reaching the beginning of the browser's internal history, you should always defer to the system's default behavior for the *Back* button.

## Provide Ancestral Navigation: Up and Home

Before Android 3.0, the most common form of ancestral navigation was the *Home* metaphor. This

was generally implemented as a *Home* item accessible via the device's *Menu* button, or a *Home* button at the top-left of the screen, usually as a component of the Action Bar ([pattern docs \(/design/patterns/actionbar.html\)](#) at Android Design). Upon selecting *Home*, the user would be taken to the screen at the top of the screen hierarchy, generally known as the application's home screen.

Providing direct access to the application's home screen can give the user a sense of comfort and security. Regardless of where they are in the application, if they get lost in the app, they can select *Home* to arrive back at the familiar home screen.

Android 3.0 introduced the *Up* metaphor, which is presented in the Action Bar as a substitute for the *Home* button described above. Upon tapping *Up*, the user should be taken to the parent screen in the hierarchy. This navigation step is usually the previous screen (as described with the *Back* button discussion above), but this is not universally the case. Thus, developers must ensure that *Up* for each screen navigates to a single, predetermined parent screen.

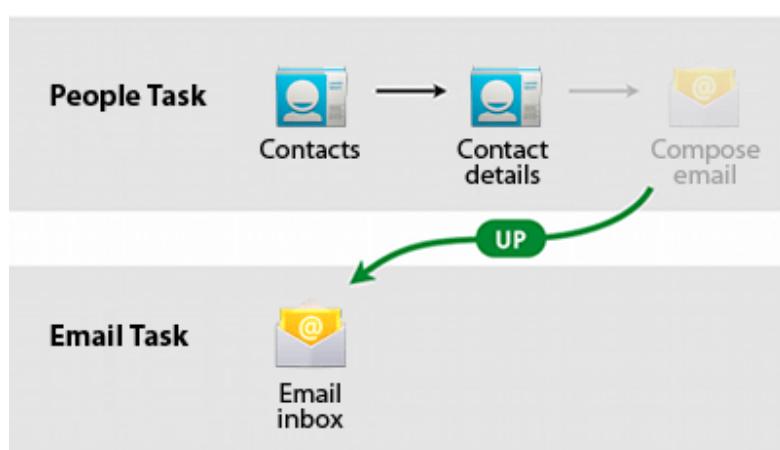


Figure 2. Example behavior for up navigation after entering the Email app from the People app.

In some cases, it's appropriate for *Up* to perform an action rather than navigating to a parent screen. Take for example, the Gmail application for Android 3.0-based tablets. When viewing a mail conversation while holding the device in landscape, the conversation list, as well as the conversation details are presented side-by-side. This is a form of parent-child screen grouping, as discussed in a [previous lesson \(multiple-sizes.html\)](#). However, when viewing a mail conversation in the portrait orientation, only the conversation details are shown. The *Up* button is used to temporarily show the parent pane, which slides in from the left of the screen. Pressing the *Up* button again while the left pane is visible exits the context of the individual conversation, up to a full-screen list of conversations.

**Implementation Note:** As a best practice, when implementing either *Home* or *Up*, make sure to clear the back stack of any descendant screens. For *Home*, the only remaining screen on the back stack should be the home screen. For *Up* navigation, the current screen should be removed from the back stack, unless *Back* navigates across screen hierarchies. You can use the [FLAG\\_ACTIVITY\\_CLEAR\\_TOP \(/reference/android/content/Intent.html#FLAG\\_ACTIVITY\\_CLEAR\\_TOP\)](#) and [FLAG\\_ACTIVITY\\_NEW\\_TASK \(/reference/android/content/Intent.html#FLAG\\_ACTIVITY\\_NEW\\_TASK\)](#) intent flags together to achieve this.

In the last lesson, we apply the concepts discussed in all of the lessons so far to create interaction design wireframes for our example news application.

# Putting it All Together: Wireframing the Example App

Now that we have a solid understanding of navigation patterns and screen grouping techniques, it's time to apply them to our screens. Let's take another look at our exhaustive screen map for the example news application from the [first lesson \(screen-planning.html\)](#), below.

## THIS LESSON TEACHES YOU TO:

1. [Choose Patterns](#)
2. [Sketch and Wireframe](#)
3. [Create Digital Wireframes](#)

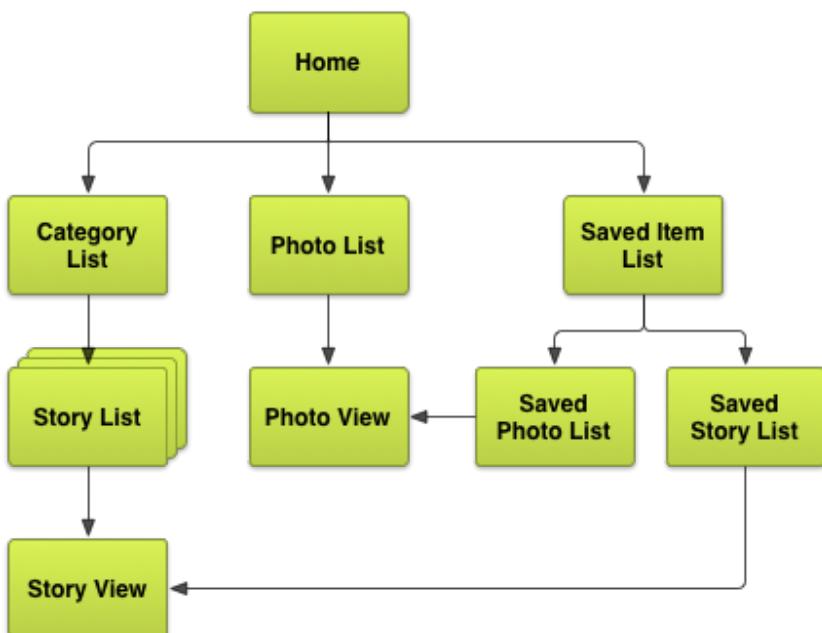


Figure 1. Exhaustive screen map for the example news application.

Our next step is to choose and apply navigation patterns discussed in the previous lessons to this screen map, maximizing navigation speed and minimizing the number of touches to access data, while keeping the interface intuitive and consistent with Android best practices. We also need to make different choices for our different target device form factors. For simplicity, let's focus on tablets and handsets.

## Choose Patterns

First, our second-level screens (*Story Category List*, *Photo List*, and *Saved Item List*) can be grouped together using tabs. Note that we don't necessarily have to use horizontally arranged tabs; in some cases a drop-down list UI element can serve as a suitable replacement, especially on devices with narrow screens such as handsets. We can also group the *Saved Photo List* and *Saved Story List* screens together using tabs on handsets, or use multiple vertical content panes on tablets.

Finally, let's look at how we present news stories. The first option to simplify navigation across different story categories is to use horizontal paging, with a set of labels above the horizontal swiping surface, indicating the currently visible and adjacently accessible categories. On tablets in

the landscape orientation, we can go a step further and present the horizontally-pageable *Story List* screen as a left pane, and the *Story View* screen as the primary content pane on the right.

Below are diagrams representing the new screen maps for handsets and tablets after applying these navigation patterns.

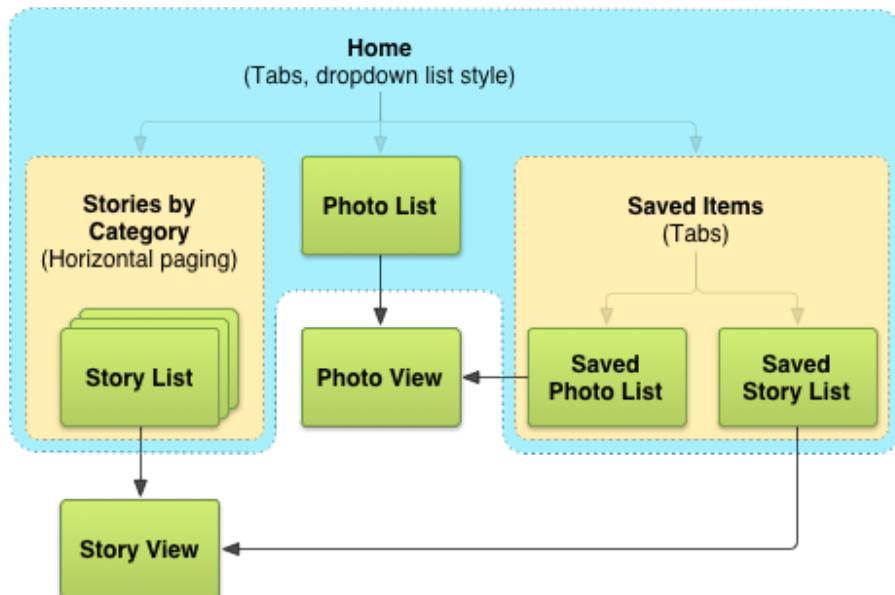


Figure 2. Final screen map for the example news application on handsets.

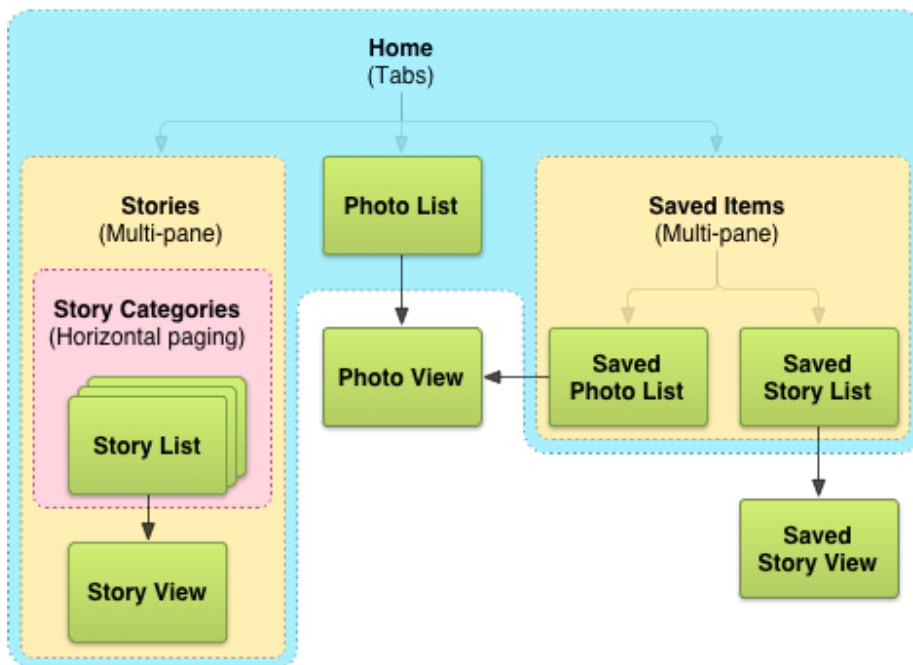


Figure 3. Final screen map for the example news application on tablets, in landscape.

At this point, it's a good idea to think of screen map variations, in case your chosen patterns don't apply well in practice (when you sketch the application's screen layouts). Below is an example screen map variation for tablets that presents story lists for different categories side-by-side, with story view screens remaining independent.

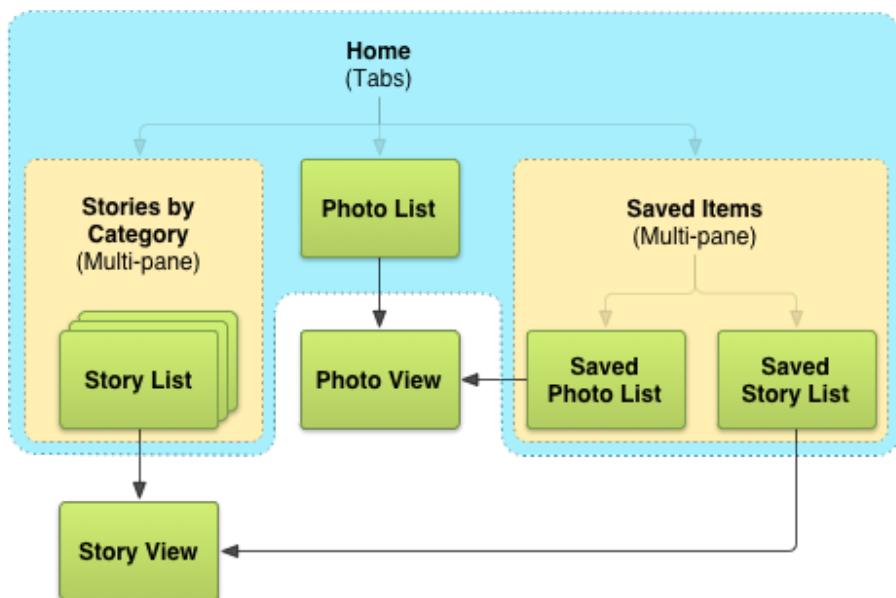


Figure 4. Example alternate screen map for tablets, in landscape.

## Sketch and Wireframe

---

Wireframing ([http://en.wikipedia.org/wiki/Website\\_wireframe](http://en.wikipedia.org/wiki/Website_wireframe)) is the step in the design process where you begin to lay out your screens. Get creative and begin imagining how to arrange UI elements to allow users to navigate your app. Keep in mind that at this point, pixel-perfect precision (creating high-fidelity mockups) is not important.

The easiest and fastest way to get started is to sketch out your screens by hand using paper and pencils. Once you begin sketching, you may uncover practicality issues in your original screen map or decisions on which patterns to use. In some cases, patterns may apply well to a given design problem in theory, but in practice they may break down and cause visual clutter or interactional issues (for example, if there are two rows of tabs on the screen). If that happens, explore other navigation patterns, or variations on chosen patterns, to arrive at a more optimal set of sketches.

After you're satisfied with initial sketches, it's a good idea to move on to digital wireframing using software such as Adobe® Illustrator, Adobe® Fireworks, OmniGraffle, or any other vector illustration tools. When choosing which tool to use, consider the following features:

- Are interactive wireframes possible? Tools such as Adobe® Fireworks offer this functionality.
- Is there screen 'master' functionality, allowing re-use of visual elements across different screens? For example, Action Bars should be visible on almost every screen in your application.
- What's the learning curve? Professional vector illustration tools may have a steep learning curve, while tools designed for wireframing may offer a smaller set of features that are more relevant to the task.

Lastly, the XML Layout Editor that comes with the Android Development Tools (ADT) ([/tools/help/adt.html](#)) plugin for Eclipse can often be used for prototyping. However, you should be careful to focus more on the high-level layout and less on visual design details at this point.

## Create Digital Wireframes

After sketching out layouts on paper and choosing a digital wireframing tool that works for you, you can create the digital wireframes that will serve as the starting point for your application's visual design. Below are example wireframes for our news application, corresponding one-to-one with our screen maps from earlier in this lesson.

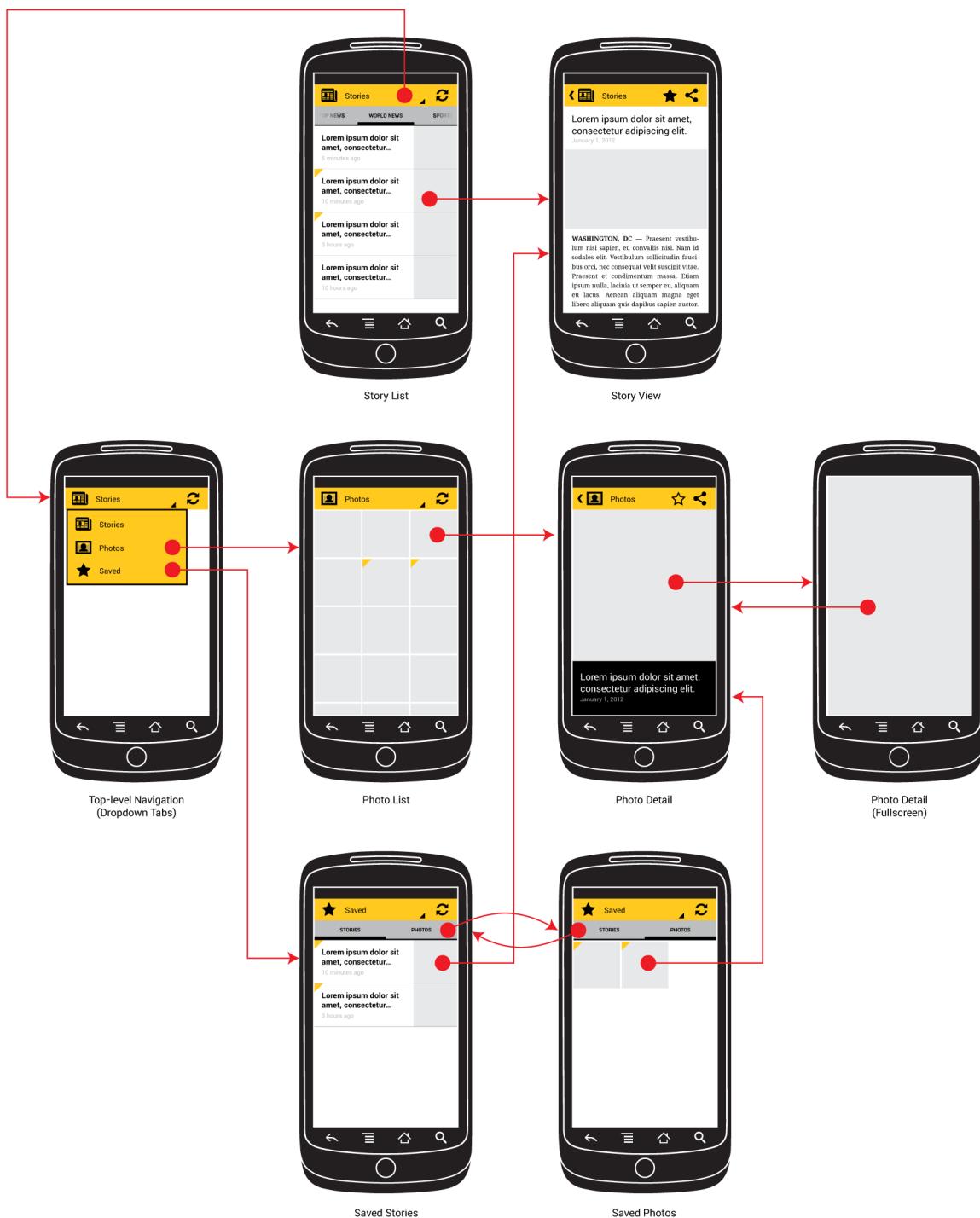


Figure 5. Example news application wireframes, for handsets in portrait. ([Download SVG \(example-wireframe-phone.svg\)](#))

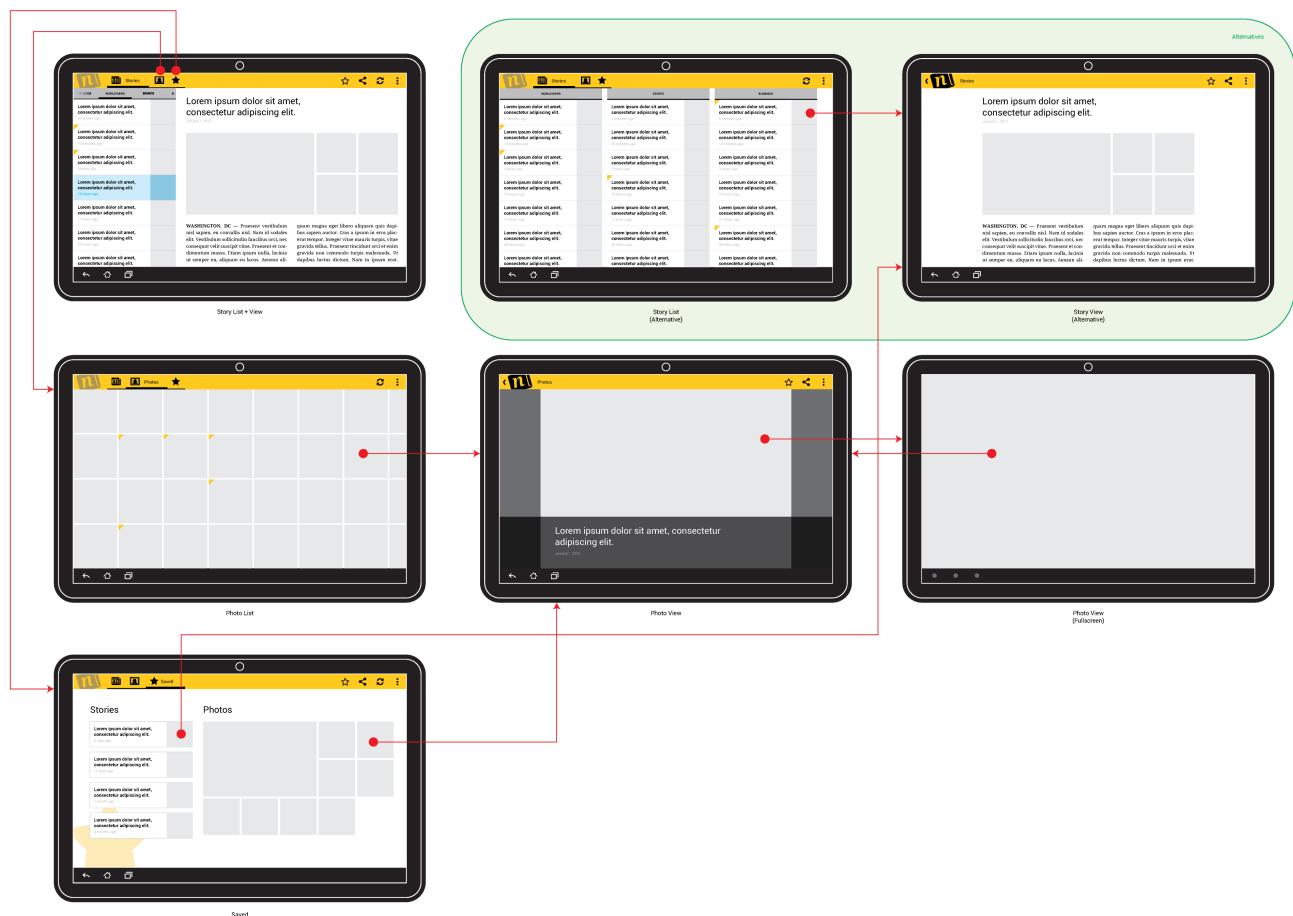


Figure 6. Example news application wireframes, for tablets in landscape. Also includes an alternate layout for presenting story lists. ([Download SVG \(example-wireframe-tablet.svg\)](#))

([Download SVG for device wireframe art \(example-wireframe-device-template.svg\)](#))

## Next Steps

Now that you've designed effective and intuitive intra-app navigation for your application, you can begin to spend time refining the user interface for each individual screen. For example, you can choose to use richer widgets in place of simple text labels, images, and buttons when displaying interactive content. You can also begin defining the visual styling of your application, incorporating elements from your brand's visual language in the process.

Lastly, it may be time to begin implementing your designs and writing the code for the application using the Android SDK. To get started, take a look at the following resources:

- [Developer's Guide: User Interface](#): learn how to implement your user interface designs using the Android SDK.
- [Action Bar](#): implement tabs, up navigation, on-screen actions, etc.

- Fragments: implement re-usable, multi-pane layouts
- Support Library: implement horizontal paging (swipe views) using ViewPager

# Implementing Effective Navigation

This class demonstrates how to implement the key navigation design patterns detailed in the [Designing Effective Navigation \(/training/design-navigation/index.html\)](#) class. The lessons in this class cover implementing navigation up, down, and across your application's [screen map \(/training/design-navigation/screen-planning.html#diagram-relationships\)](#).

After reading through the lessons in this class and exploring the associated sample application (see right), you should also have a basic understanding of how to use [ActionBar \(/reference/android/app/ActionBar.html\)](#) and [ViewPager \(/reference/android/support/v4/view/ViewPager.html\)](#), two components that are fundamental to core app navigation.

## Lessons

---

### [Implementing Lateral Navigation](#)

Learn how to implement tabs and horizontal paging (swipe views).

### [Implementing Ancestral Navigation](#)

Learn how to implement *Up* navigation.

### [Implementing Temporal Navigation](#)

Learn how to correctly handle the *Back* button.

### [Implementing Descendant Navigation](#)

Learn the finer points of implementing navigation into your application's information hierarchy.

---

### DEPENDENCIES AND PREREQUISITES

- API level 14
- Understanding of fragments and Android layouts
- [The Android Support Package](#)
- [Designing Effective Navigation](#)

---

### YOU SHOULD ALSO READ

- [Action Bar](#)
- [Fragments](#)
- [Designing for Multiple Screens](#)

---

### TRY IT OUT

[Download the sample app](#)

EffectiveNavigation.zip

# Implementing Lateral Navigation

*Lateral navigation* is navigation between sibling screens in the application's screen hierarchy (sometimes referred to as a screen map). The most prominent lateral navigation patterns are tabs and horizontal paging (also known as swipe views). This pattern and others are described in [Designing Effective Navigation](#) ([/training/design-navigation/descendant-lateral.html](#)). This lesson covers how to implement several of the primary lateral navigation patterns in Android.

## Implement Tabs

Tabs allow the user to navigate between sibling screens by selecting the appropriate tab indicator available at the top of the display. In Android 3.0 and later, tabs are implemented using the [ActionBar](#) ([/reference/android/app/ActionBar.html](#)) class, and are generally set up in [Activity.onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)). In some cases, such as when horizontal space is limited and/or the number of tabs is large, an appropriate alternate presentation for tabs is a dropdown list (sometimes implemented using a [Spinner](#) ([/reference/android/widget/Spinner.html](#))).

In previous versions of Android, tabs could be implemented using a [TabWidget](#) ([/reference/android/widget/TabWidget.html](#)) and [TabHost](#) ([/reference/android/widget/TabHost.html](#)). For details, see the [Hello, Views](#) ([/resources/tutorials/views/hello-tabwidget.html](#)) tutorial.

As of Android 3.0, however, you should use either [NAVIGATION\\_MODE\\_TABS](#) ([/reference/android/app/ActionBar.html#NAVIGATION\\_MODE\\_TABS](#)) or [NAVIGATION\\_MODE\\_LIST](#) ([/reference/android/app/ActionBar.html#NAVIGATION\\_MODE\\_LIST](#)) along with the [ActionBar](#) ([/reference/android/app/ActionBar.html](#)) class.

### Implement the Tabs Pattern with NAVIGATION\_MODE\_TABS

To create tabs, you can use the following code in your activity's [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method. Note that the exact presentation of tabs may vary per device and by the current device configuration, to make best use of available screen space. For example, Android may automatically collapse tabs into a dropdown list if tabs don't fit horizontally in the action bar.

```
@Override  
public void onCreate(Bundle savedInstanceState) {
```

### THIS LESSON TEACHES YOU TO

1. [Implement Tabs](#)
2. [Implement Horizontal Paging \(Swipe Views\)](#)
3. [Implement Swiping Between Tabs](#)

### YOU SHOULD ALSO READ

- [Providing Descendant and Lateral Navigation](#)
- [Android Design: Tabs](#)
- [Android Design: Swipe Views](#)

### TRY IT OUT

[Download the sample app](#)

EffectiveNavigation.zip

```
...
final ActionBar actionBar = getActionBar();

// Specify that tabs should be displayed in the action bar.
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

// Create a tab listener that is called when the user changes tabs.
ActionBar.TabListener tabListener = new ActionBar.TabListener() {
    public void onTabSelected(ActionBar.Tab tab,
        FragmentTransaction ft) { }

    public void onTabUnselected(ActionBar.Tab tab,
        FragmentTransaction ft) { }

    public void onTabReselected(ActionBar.Tab tab,
        FragmentTransaction ft) { }
};

// Add 3 tabs.
for (int i = 0; i < 3; i++) {
    actionBar.addTab(
        actionBar.newTab()
            .setText("Tab " + (i + 1))
            .setTabListener(tabListener));
}

...
}
```

## Implement the Tabs Pattern with NAVIGATION\_MODE\_LIST

To use a dropdown list instead, use the following code in your activity's [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method. Dropdown lists are often preferable in cases where more information must be shown per navigation item, such as unread message counts, or where the number of available navigation items is large.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...
    final ActionBar actionBar = getActionBar();

    // Specify that a dropdown list should be displayed in the action bar.
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);

    actionBar.setListNavigationCallbacks(
        // Specify a SpinnerAdapter to populate the dropdown list.
        new ArrayAdapter(
            actionBar.getThemedContext(),
            android.R.layout.simple_list_item_1,
```

```
        android.R.id.text1,
        new String[]{"Tab 1", "Tab 2", "Tab 3"}),  
  
        // Provide a listener to be called when an item is selected.
        new ActionBar.OnNavigationListener() {
            public boolean onNavigationItemSelected(
                int position, long id) {
                // Take action here, e.g. switching to the
                // corresponding fragment.
                return true;
            }
        });
    ...  
}
```

## Implement Horizontal Paging (Swipe Views)

Horizontal paging, or swipe views, allow users to [swipe \(/design/patterns/swipe-views.html\)](#) horizontally on the current screen to navigate to adjacent screens. This pattern can be implemented using the [ViewPager \(/reference/android/support/v4/view/ViewPager.html\)](#) widget, currently available as part of the [Android Support Package \(/tools/extras/support-library.html\)](#). For navigating between sibling screens representing a fixed number of sections, it's best to provide the [ViewPager \(/reference/android/support/v4/view/ViewPager.html\)](#) with a [FragmentPagerAdapter \(/reference/android/support/v4/app/FragmentPagerAdapter.html\)](#). For horizontal paging across collections of objects, it's best to use a [FragmentStatePagerAdapter \(/reference/android/support/v4/app/FragmentStatePagerAdapter.html\)](#), which destroys fragments as the user navigates to other pages, minimizing memory usage.

Below is an example of using a [ViewPager \(/reference/android/support/v4/view/ViewPager.html\)](#) to swipe across a collection of objects.

```
public class CollectionDemoActivity extends FragmentActivity {
    // When requested, this adapter returns a DemoObjectFragment,
    // representing an object in the collection.
    DemoCollectionPagerAdapter mDemoCollectionPagerAdapter;
    ViewPager mViewPager;

    public void onCreate(Bundle savedInstanceState) {
        // ViewPager and its adapters use support library
        // fragments, so use getSupportFragmentManager.
        mDemoCollectionPagerAdapter =
            new DemoCollectionPagerAdapter(
                getSupportFragmentManager());
        mViewPager = (ViewPager) findViewById(R.id.pager);
        mViewPager.setAdapter(mDemoCollectionPagerAdapter);
    }
}
```

```
// Since this is an object collection, use a FragmentStatePagerAdapter,  
// and NOT a FragmentPagerAdapter.  
public class DemoCollectionPagerAdapter extends  
    FragmentStatePagerAdapter {  
    public DemoCollectionPagerAdapter(FragmentManager fm) {  
        super(fm);  
    }  
  
    @Override  
    public Fragment getItem(int i) {  
        Fragment fragment = new DemoObjectFragment();  
        Bundle args = new Bundle();  
        // Our object is just an integer :-P  
        args.putInt(DemoObjectFragment.ARG_OBJECT, i + 1);  
        fragment.setArguments(args);  
        return fragment;  
    }  
  
    @Override  
    public int getCount() {  
        return 100;  
    }  
  
    @Override  
    public CharSequence getPageTitle(int position) {  
        return "OBJECT " + (position + 1);  
    }  
}  
  
// Instances of this class are fragments representing a single  
// object in our collection.  
public static class DemoObjectFragment extends Fragment {  
    public static final String ARG_OBJECT = "object";  
  
    @Override  
    public View onCreateView(LayoutInflater inflater,  
        ViewGroup container, Bundle savedInstanceState) {  
        // The last two arguments ensure LayoutParams are inflated  
        // properly.  
        View rootView = inflater.inflate(  
            R.layout.fragment_collection_object, container, false);  
        Bundle args = getArguments();  
        ((TextView) rootView.findViewById(android.R.id.text1)).setText(  
            Integer.toString(args.getInt(ARG_OBJECT)));  
        return rootView;  
    }  
}
```

You can also add indicators to your horizontal paging UI by adding a [PagerTitleStrip](#) ([/reference/android/support/v4/view/PagerTitleStrip.html](#)). Below is an example layout XML file for an activity whose entire contents are a [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#)) and a top-aligned [PagerTitleStrip](#) ([/reference/android/support/v4/view/PagerTitleStrip.html](#)) inside it. Individual pages (provided by the adapter) occupy the remaining space inside the [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#)).

```
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.v4.view.PagerTitleStrip
        android:id="@+id/pager_title_strip"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="top"
        android:background="#33b5e5"
        android:textColor="#fff"
        android:paddingTop="4dp"
        android:paddingBottom="4dp" />

</android.support.v4.view.ViewPager>
```

## Implement Swiping Between Tabs

One of the key design recommendations in Android 4.0 for tabs is to [allow swiping](#) ([/design/patterns/swipe-views.html](#)) between them where appropriate. This behavior enables users to swipe horizontally across the selected tab's contents to navigate to adjacent tabs, without needed to directly interact with the tabs themselves. To implement this, you can use a [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#)) in conjunction with the [ActionBar](#) ([/reference/android/app/ActionBar.html](#)) tabs API.

Upon observing the current page changing, select the corresponding tab. You can set up this behavior using an [ViewPager.OnPageChangeListener](#) ([/reference/android/support/v4/view/ViewPager.OnPageChangeListener.html](#)) in your activity's [onCreate\(\)](#) ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    ...
    mViewPager.setOnPageChangeListener(
        new ViewPager.SimpleOnPageChangeListener() {
            @Override
            public void onPageSelected(int position) {
```

```
// When swiping between pages, select the
// corresponding tab.
getActionBar().setSelectedNavigationItem(position);
}
});
...
}
```

And upon selecting a tab, switch to the corresponding page in the [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#)). To do this, add an [ActionBar.TabListener](#) ([/reference/android/app/ActionBar.TabListener.html](#)) to your tab when creating it using the [newTab\(\)](#) ([/reference/android/app/ActionBar.html#newTab\(\)](#)) method:

```
ActionBar.newTab()
...
.setTabListener(new ActionBar.TabListener() {
    public void onTabSelected(ActionBar.Tab tab,
        FragmentTransaction ft) {
        // When the tab is selected, switch to the
        // corresponding page in the ViewPager.
        mViewPager.setCurrentItem(tab.getPosition());
    }
...
}));
```

# Implementing Ancestral Navigation

Ancestral navigation is up the application's information hierarchy, where the top of the hierarchy (or root) is the application's home screen. This navigation concept is described in [Designing Effective Navigation \(/training/design-navigation/ancestral-temporal.html\)](#). This lesson discusses how to provide ancestral navigation using the *Up* button in the action bar.

## Implement Up Navigation

When implementing ancestral navigation, all screens in your application that aren't the home screen should offer a means of navigating to the immediate parent screen in the hierarchy via the *Up* button in the action bar.

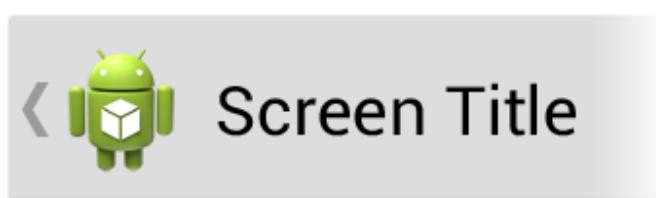


Figure 1. The *Up* button in the action bar.

Regardless of how the current screen was reached, pressing this button should always take the user to the same screen in the hierarchy.

To implement *Up*, enable it in the action bar in your activity's [onCreate\(\) \(/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)\)](#) method:

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    ...  
    getActionBar().setDisplayHomeAsUpEnabled(true);  
    ...  
}
```

You should also handle `android.R.id.home` in [onOptionsItemSelected\(\) \(/reference/android/app/Activity.html#onOptionsItemSelected\(android.view.MenuItem\)\)](#). This resource is the menu item ID for the *Home* (or *Up*) button. To ensure that a specific parent activity is shown, *DO NOT* simply call [finish\(\) \(/reference/android/app/Activity.html#finish\(\)\)](#). Instead, use an intent such as the one described below.

```
@Override
```

### THIS LESSON TEACHES YOU TO:

1. [Implement Up Navigation](#)
2. [Properly Handle the Application Home Screen](#)

### YOU SHOULD ALSO READ

- [Providing Ancestral and Temporal Navigation](#)
- [Tasks and Back Stack](#)
- [Android Design: Navigation](#)

### TRY IT OUT

[Download the sample app](#)

EffectiveNavigation.zip

```

public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case android.R.id.home:
            // This . Advanced Training
            // in th
            Intent p Making Your App
            parentAc Location Aware
            Performing Network
            Operations
            startAct
            finish() Transferring Data
            Without Draining the
            return t Battery
    }
    return super.onOptionsItemSelected(item);
}

```

When the current activity is reached via an intent from application with a synthesized navigation ([/design/patterns/navigation/v4/app/TaskStackBuilder.html](#))

The [NavUtils](#) ([/reference/android/support/v4/app/TaskStackBuilder.html](#)) provide helper classes for these two helper classes is

- Improving Layout Performance
- Managing Audio Playback
- Optimizing Battery Life
- Creating Custom Views
- Adding Search Functionality
- Remembering Users
- Sharing Content
- Capturing Photos

b) button is pressed

```

Intent(this, MyParentActivity.class)
.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP |
        Intent.FLAG_ACTIVITY_NEW_TASK |
        Intent.FLAG_ACTIVITY_BROUGHT_TO_FRONT);
);

```

nt application—for example if it was Up should create a new task for the is described in [Android Design: Task Stack](#) ([/reference/android/support/TaskStackBuilder.html](#))

[NavUtils](#) ([/reference/android/support/v4/app/TaskStackBuilder.html](#)) and [TaskStackBuilder](#) ([/reference/android/support/v4/app/TaskStackBuilder.html](#)) provide helper classes for these two helper classes is

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case android.R.id.home:
            Intent upIntent = new Intent(this, MyParentActivity.class);
            if (NavUtils.shouldUpRecreateTask(this, upIntent)) {
                // This activity is not part of the application's task, so create
                // with a synthesized back stack.
                TaskStackBuilder.from(this)
                    .addNextIntent(new Intent(this, MyGreatGrandParentActivity.class))
                    .addNextIntent(new Intent(this, MyGrandParentActivity.class))
                    .addNextIntent(upIntent)
                    .startActivities();
            }
            finish();
        } else {
            // This activity is part of the application's task, so simply
            // navigate up to the hierarchical parent activity.
            NavUtils.navigateUpTo(this, upIntent);
        }
    }
    return true;
}

```

```
    return super.onOptionsItemSelected(item);  
}
```

Get Started

Advanced Training

## Properly Handle the Home Button

By default, the *Home* button will navigate home—or up one level if you’re in a fragment.

```
getActionBar().setHomeActionModeEnabled(false);
```

Making Your App Location Aware

Performing Network Operations

Transferring Data Without Draining the Battery

Syncing to the Cloud

Designing for Multiple Screens

Improving Layout Performance

Managing Audio Playback

Optimizing Battery Life

Creating Custom Views

Adding Search Functionality

Remembering Users

Sharing Content

Capturing Photos

## Handle the Home Screen

Since it does not make much sense to have a “Home” screen, you should disable the button like so:

# Implementing Temporal Navigation

*Temporal navigation* is navigation to previously visited screens. Users can visit previous screens by pressing the device *Back* button. This user interface pattern is described further in [Providing Ancestral and Temporal Navigation](#) ([/training/design-navigation/ancestral-temporal.html](#)) in [Designing Effective Navigation](#) and in [Android Design: Navigation](#) ([/design/patterns/navigation.html](#)).

Android handles basic *Back* navigation for you (see [Tasks and Back Stack](#) ([/guide/components/tasks-and-back-stack.html](#)) for details on this behavior). This lesson discusses a number of cases where applications should provide specialized logic for the *Back* button.

## THIS LESSON TEACHES YOU TO:

1. [Implement Back Navigation with Fragments](#)
2. [Implement Back Navigation with WebViews](#)

## YOU SHOULD ALSO READ

- [Providing Ancestral and Temporal Navigation](#)
- [Tasks and Back Stack](#)
- [Android Design: Navigation](#)

## Implement Back Navigation with Fragments

When using fragments in your application, individual [FragmentTransaction](#) ([/reference/android/app/FragmentTransaction.html](#)) objects can represent context changes that should be added to the back stack. For example, if you are implementing a [master/detail flow](#) ([descendant.html#master-detail](#)) on a handset by swapping out fragments (thus emulating a [startActivity\(\)](#) ([/reference/android/app/Activity.html#startActivity\(android.content.Intent\)](#)) call), you should ensure that pressing the *Back* button on a detail screen returns the user to the master screen. To do so, you can use [addToBackStack\(\)](#) ([/reference/android/app/FragmentTransaction.html#addToBackStack\(java.lang.String\)](#)):

```
// Works with either the framework FragmentManager or the
// support package FragmentManager (getSupportFragmentManager).
getFragmentManager().beginTransaction()
    .add(detailFragment, "detail")

    // Add this transaction to the back stack and commit.
    .addToBackStack()
    .commit();
```

The activity's [FragmentManager](#) ([/reference/android/app/FragmentManager.html](#)) handles *Back* button presses if there are [FragmentTransaction](#) ([/reference/android/app/FragmentTransaction.html](#)) objects on the back stack. When this happens, the [FragmentManager](#) ([/reference/android/app/FragmentManager.html](#)) pops the most recent transaction off the back stack and performs the reverse action (e.g., removing a fragment if the transaction added it).

If your application updates other user interface elements to reflect the current state of your fragments, such as the action bar, remember to update the UI when you commit the transaction. You should update your user interface after the fragment manager back stack changes in addition

to when you commit the transaction. You can listen for when a `FragmentTransaction` is reverted by setting up an [`FragmentManager.OnBackStackChangedListener`](#) ([/reference/android/app/FragmentManager.OnBackStackChangedListener.html](#)):

```
getFragmentManager().addOnBackStackChangedListener(  
    new FragmentManager.OnBackStackChangedListener() {  
        public void onBackStackChanged() {  
            // Update your UI here.  
        }  
    });
```

## Implement Back Navigation with WebViews

---

If a part of your application is contained in a [`WebView`](#) ([/reference/android/webkit/WebView.html](#)), it may be appropriate for `Back` to traverse browser history. To do so, you can override [`onBackPressed\(\)`](#) ([/reference/android/app/Activity.html#onBackPressed\(\)](#)) and proxy to the `WebView` if it has history state:

```
@Override  
public void onBackPressed() {  
    if (mWebView.canGoBack()) {  
        mWebView.goBack();  
        return;  
    }  
  
    // Otherwise defer to system default behavior.  
    super.onBackPressed();  
}
```

Be careful when using this mechanism with highly dynamic web pages that can grow a large history. Pages that generate an extensive history, such as those that make frequent changes to the document hash, may make it tedious for users to get out of your activity.

# Implementing Descendant Navigation

*Descendant navigation* is navigation down the application's information hierarchy. This is described in [Designing Effective Navigation \(/training/design-navigation/descendant-lateral.html\)](#) and also covered in [Android Design: Application Structure \(/design/patterns/app-structure.html\)](#).

Descendant navigation is usually implemented using [Intent \(/reference/android/content/Intent.html\)](#) objects and [startActivity\(\) \(/reference/android/content/Context.html#startActivity\(android.content.Intent\)\)](#), or by adding fragments to an activity using [FragmentTransaction \(/reference/android/app/FragmentTransaction.html\)](#) objects. This lesson covers other interesting cases that arise when implementing descendant navigation.

## Implement Master/Detail Flows Across Handsets and Tablets

In a *master/detail* navigation flow, a *master* screen contains a list of items in a collection, and a *detail* screen shows detailed information about a specific item within that collection. Implementing navigation from the master screen to the detail screen is one form of descendant navigation.

Handset touchscreens are most suitable for displaying one screen at a time (either the master or the detail screen); this concern is further discussed in [Planning for Multiple Touchscreen Sizes \(/training/design-navigation/multiple-sizes.html\)](#). Descendant navigation in this case is often implemented using an [Intent \(/reference/android/content/Intent.html\)](#) that starts an activity representing the detail screen. On the other hand, tablet displays, especially when viewed in the landscape orientation, are best suited for showing multiple content panes at a time: the master on the left, and the detail to the right). Here, descendant navigation is usually implemented using a [FragmentTransaction \(/reference/android/app/FragmentTransaction.html\)](#) that adds, removes, or replaces the detail pane with new content.

The basics of implementing this pattern are described in the [Implementing Adaptive UI Flows \(/training/multiscreen/adaptui.html\)](#) lesson of the *Designing for Multiple Screens* class. The class describes how to implement a master/detail flow using two activities on a handset and a single activity on a tablet.

## Navigate into External Activities

There are cases where descending into your application's information hierarchy leads to activities

### THIS LESSON TEACHES YOU TO:

1. [Implement Master/Detail Flows Across Handsets and Tablets](#)
2. [Navigate into External Activities](#)

### YOU SHOULD ALSO READ

- [Providing Descendant and Lateral Navigation](#)
- [Android Design: App Structure](#)
- [Android Design: Multi-pane Layouts](#)

### TRY IT OUT

[Download the sample app](#)

EffectiveNavigation.zip

from other applications. For example, when viewing the contact details screen for an entry in the phone address book, a child screen detailing recent posts by the contact on a social network may belong to a social networking application.

When launching another application's activity to allow the user to say, compose an email or pick a photo attachment, you generally don't want the user to return to this activity if they relaunch your application from the Launcher (the device home screen). It would be confusing if touching your application icon brought the user to a "compose email" screen.

To prevent this from occurring, simply add the [FLAG\\_ACTIVITY\\_CLEAR\\_WHEN\\_TASK\\_RESET](#) ([Intent.html#FLAG\\_ACTIVITY\\_CLEAR\\_WHEN\\_TASK\\_RESET](#)) flag to the intent used to launch the external activity, like so:

```
Intent externalActivityIntent = new Intent(Intent.ACTION_PICK);
externalActivityIntent.setType("image/*");
externalActivityIntent.addFlags(
    Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET);
startActivity(externalActivityIntent);
```

# Designing for TV

Smart TVs powered by Android bring your favorite Android apps to the best screen in your house. Thousands of apps in the Google Play Store are already optimized for TVs. This class shows how you can optimize your Android app for TVs, including how to build a layout that works great when the user is ten feet away and navigating with a remote control.

## DEPENDENCIES AND PREREQUISITES

- Android 2.0 (API Level 5) or higher

## Lessons

---

### Optimizing Layouts for TV

Shows you how to optimize app layouts for TV screens, which have some unique characteristics such as:

- permanent "landscape" mode
- high-resolution displays
- "10 foot UI" environment.

### Optimizing Navigation for TV

Shows you how to design navigation for TVs, including:

- handling D-pad navigation
- providing navigational feedback
- providing easily-accessible controls on the screen.

### Handling features not supported on TV

Lists the hardware features that are usually not available on TVs. This lesson also shows you how to provide alternatives for missing features or check for missing features and disable code at run time.

# Optimizing Layouts for TV

When your application is running on a television set, you should assume that the user is sitting about ten feet away from the screen. This user environment is referred to as the [10-foot UI](http://en.wikipedia.org/wiki/10-foot_user_interface) ([http://en.wikipedia.org/wiki/10-foot\\_user\\_interface](http://en.wikipedia.org/wiki/10-foot_user_interface)). To provide your users with a usable and enjoyable experience, you should style and lay out your UI accordingly..

This lesson shows you how to optimize layouts for TV by:

- Providing appropriate layout resources for landscape mode.
- Ensuring that text and controls are large enough to be visible from a distance.
- Providing high resolution bitmaps and icons for HD TV screens.

## Design Landscape Layouts

TV screens are always in landscape orientation. Follow these tips to build landscape layouts optimized for TV screens:

- Put on-screen navigational controls on the left or right side of the screen and save the vertical space for content.
- Create UIs that are divided into sections, by using [Fragments](#) and use view groups like [GridView](#) instead of [ListView](#) to make better use of the horizontal screen space.
- Use view groups such as [RelativeLayout](#) or [LinearLayout](#) to arrange views. This allows the Android system to adjust the position of the views to the size, alignment, aspect ratio, and pixel density of the TV screen.
- Add sufficient margins between layout controls to avoid a cluttered UI.

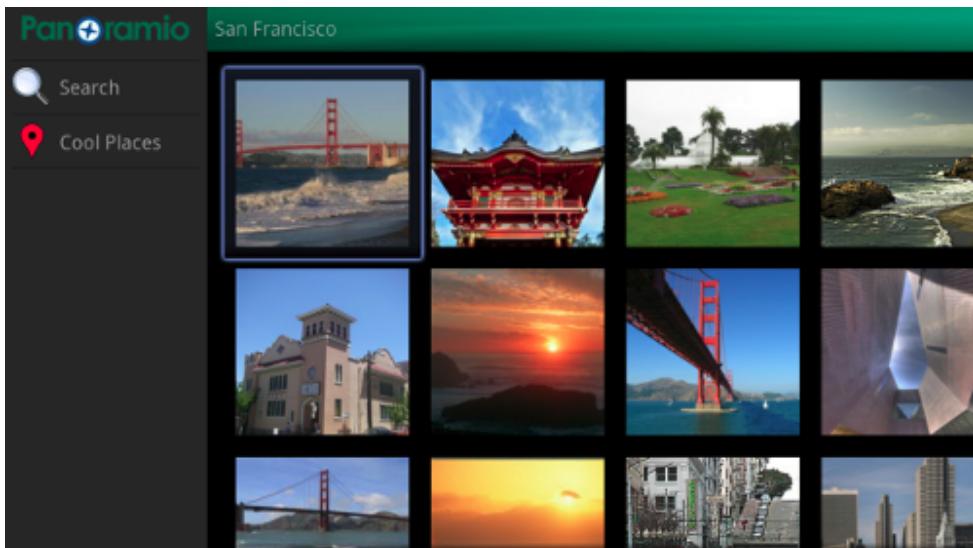
For example, the following layout is optimized for TV:

### THIS LESSON TEACHES YOU TO

1. [Design Landscape Layouts](#)
2. [Make Text and Controls Easy to See](#)
3. [Design for High-Density Large Screens](#)
4. [Design to Handle Large Bitmaps](#)

### YOU SHOULD ALSO READ

- [Supporting Multiple Screens](#)



In this layout, the controls are on the lefthand side. The UI is displayed within a [GridView](#) ([/reference/android/widget/GridView.html](#)), which is well-suited to landscape orientation. In this layout both GridView and Fragment have the width and height set dynamically, so they can adjust to the screen resolution. Controls are added to the left side Fragment programatically at runtime. The layout file for this UI is `res/layout-land-large/photagrid_tv.xml`. (This layout file is placed in `layout-land-large` because TVs have large screens with landscape orientation. For details refer to [Supporting Multiple Screens](#) ([/guide/practices/screens\\_support.html](#)).

`res/layout-land-large/photagrid_tv.xml`

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

    <fragment
        android:id="@+id/leftsidecontrols"
        android:layout_width="0dip"
        android:layout_marginLeft="5dip"
        android:layout_height="match_parent" />

    <GridView
        android:id="@+id/gridview"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</RelativeLayout>
```

To set up action bar items on the left side of the screen, you can also include the [Left navigation bar library](#) (<http://code.google.com/p/googletv-android-samples/source/browse/#git%2FLeftNavBarLibrary>) in your application to set up action items on the left side of the screen, instead of creating a custom Fragment to add controls:

```
LeftNavBar bar = (LeftNavBarService.instance()).getLeftNavBar(this);
```

When you have an activity in which the content scrolls vertically, always use a left navigation bar; otherwise, your users have to scroll to the top of the content to switch between the content view and the ActionBar. Look at the [Left navigation bar sample app](#) (<http://code.google.com/p/googletv-android-samples/source/browse/#git%2FLeftNavBarDemo>) to see how simple it is to include the left navigation bar in your app.

## Make Text and Controls Easy to See

The text and controls in a TV application's UI should be easily visible and navigable from a distance. Follow these tips to make them easier to see from a distance :

- Break text into small chunks that users can quickly scan.
- Use light text on a dark background. This style is easier to read on a TV.
- Avoid lightweight fonts or fonts that have both very narrow and very broad strokes. Use simple sans-serif fonts and use anti-aliasing to increase readability.
- Use Android's standard font sizes:

```
<TextView  
    android:id="@+id/atext"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:gravity="center_vertical"  
    android:singleLine="true"  
    android:textAppearance="?android:attr/textAppearanceMedium"/>
```

- Ensure that all your view widgets are large enough to be clearly visible to someone sitting 10 feet away from the screen (this distance is greater for very large screens). The best way to do this is to use layout-relative sizing rather than absolute sizing, and density-independent pixel units instead of absolute pixel units. For example, to set the width of a widget, use wrap\_content instead of a pixel measurement, and to set the margin for a widget, use dip instead of px values.

## Design for High-Density Large Screens

The common HDTV display resolutions are 720p, 1080i, and 1080p. Design your UI for 1080p, and then allow the Android system to downscale your UI to 720p if necessary. In general, downscaling (removing pixels) does not degrade the UI (Notice that the converse is not true; you should avoid upscaling because it degrades UI quality).

To get the best scaling results for images, provide them as [9-patch image](#) (</tools/help/draw9patch.html>) elements if possible. If you provide low quality or small images in your layouts, they will appear pixelated, fuzzy, or grainy. This is not a good experience for the user. Instead, use high-quality images.

For more information on optimizing apps for large screens see [Designing for multiple screens \(/training/multiscreen/index.html\)](#).

## Design to Handle Large Bitmaps

The Android system has a limited amount of memory, so downloading and storing high-resolution images can often cause out-of-memory errors in your app. To avoid this, follow these tips:

- Load images only when they're displayed on the screen. For example, when displaying multiple images in a [GridView](#) or [Gallery](#), only load an image when `getView()` is called on the View's [Adapter](#).
- Call `recycle()` on [Bitmap](#) views that are no longer needed.
- Use [WeakReference](#) for storing references to [Bitmap](#) objects in an in-memory [Collection](#).
- If you fetch images from the network, use [AsyncTask](#) to fetch them and store them on the SD card for faster access. Never do network transactions on the application's UI thread.
- Scale down really large images to a more appropriate size as you download them; otherwise, downloading the image itself may cause an "Out of Memory" exception. Here is sample code that scales down images while downloading:

```
// Get the source image's dimensions
BitmapFactory.Options options = new BitmapFactory.Options();
// This does not download the actual image, just downloads headers.
options.inJustDecodeBounds = true;
BitmapFactory.decodeFile(IMAGE_FILE_URL, options);
// The actual width of the image.
int srcWidth = options.outWidth;
// The actual height of the image.
int srcHeight = options.outHeight;

// Only scale if the source is bigger than the width of the destination view.
if(desiredWidth > srcWidth)
    desiredWidth = srcWidth;

// Calculate the correct inSampleSize/scale value. This helps reduce memory use
int inSampleSize = 1;
while(srcWidth / 2 > desiredWidth){
    srcWidth /= 2;
    srcHeight /= 2;
    inSampleSize *= 2;
}

float desiredScale = (float) desiredWidth / srcWidth;

// Decode with inSampleSize
options.inJustDecodeBounds = false;
options.inDither = false;
options.inSampleSize = inSampleSize;
options.inScaled = false;
```

```
// Ensures the image stays as a 32-bit ARGB_8888 image.  
// This preserves image quality.  
options.inPreferredConfig = Bitmap.Config.ARGB_8888;  
  
Bitmap sampledSrcBitmap = BitmapFactory.decodeFile(IMAGE_FILE_URL, options);  
  
// Resize  
Matrix matrix = new Matrix();  
matrix.postScale(desiredScale, desiredScale);  
Bitmap scaledBitmap = Bitmap.createBitmap(sampledSrcBitmap, 0, 0,  
    sampledSrcBitmap.getWidth(), sampledSrcBitmap.getHeight(), matrix, true);  
sampledSrcBitmap = null;  
  
// Save  
FileOutputStream out = new FileOutputStream(LOCAL_PATH_TO_STORE_IMAGE);  
scaledBitmap.compress(Bitmap.CompressFormat.JPEG, 100, out);  
scaledBitmap = null;
```

# Optimizing Navigation for TV

An important aspect of the user experience when operating a TV is the direct human interface: a remote control. As you optimize your Android application for TVs, you should pay special attention to how the user actually navigates around your application when using a remote control instead of a touchscreen.

This lesson shows you how to optimize navigation for TV by:

- Ensuring all layout controls are D-pad navigable.
- Providing highly obvious feedback for UI navigation.
- Placing layout controls for easy access.

## THIS LESSON TEACHES YOU TO

1. [Handle D-pad Navigation](#)
2. [Provide Clear Visual Indication for Focus and Selection](#)
3. [Design for Easy Navigation](#)

## YOU SHOULD ALSO READ

- [Designing Effective Navigation](#)

## Handle D-pad Navigation

On a TV, users navigate with controls on a TV remote, using either a D-pad or arrow keys. This limits movement to up, down, left, and right. To build a great TV-optimized app, you must provide a navigation scheme in which the user can quickly learn how to navigate your app using the remote.

When you design navigation for D-pad, follow these guidelines:

- Ensure that the D-pad can navigate to all the visible controls on the screen.
- For scrolling lists with focus, D-pad up/down keys scroll the list and Enter key selects an item in the list. Ensure that users can select an element in the list and that the list still scrolls when an element is selected.
- Ensure that movement between controls is straightforward and predictable.

Android usually handles navigation order between layout elements automatically, so you don't need to do anything extra. If the screen layout makes navigation difficult, or if you want users to move through the layout in a specific way, you can set up explicit navigation for your controls. For example, for an `android.widget.EditText`, to define the next control to receive focus, use:

```
<EditText android:id="@+id/LastNameField" android:nextFocusDown="@+id/FirstNameFi
```

The following table lists all of the available navigation attributes:

Attribute	Function
<code>nextFocusDown</code>	Defines the next view to receive focus when the user navigates down.
<code>nextFocusLeft</code>	Defines the next view to receive focus when the user navigates left.
<code>nextFocusRight</code>	Defines the next view to receive focus when the user navigates right.
<code>nextFocusUp</code>	Defines the next view to receive focus when the user navigates up.

To use one of these explicit navigation attributes, set the value to the ID (`android:id` value) of

another widget in the layout. You should set up the navigation order as a loop, so that the last control directs focus back to the first one.

Note: You should only use these attributes to modify the navigation order if the default order that the system applies does not work well.

## Provide Clear Visual Indication for Focus and Selection

Use appropriate color highlights for all navigable and selectable elements in the UI. This makes it easy for users to know whether the control is currently focused or selected when they navigate with a D-pad. Also, use uniform highlight scheme across your application.

Android provides [Drawable State List Resources](#) ([/guide/topics/resources/drawable-resource.html#StateList](#)) to implement highlights for selected and focused controls. For example:

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 above state list drawable to a [Button](#) ([/reference/android/widget/Button.html](#)):

```
<Button
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:background="@drawable/button" />
```

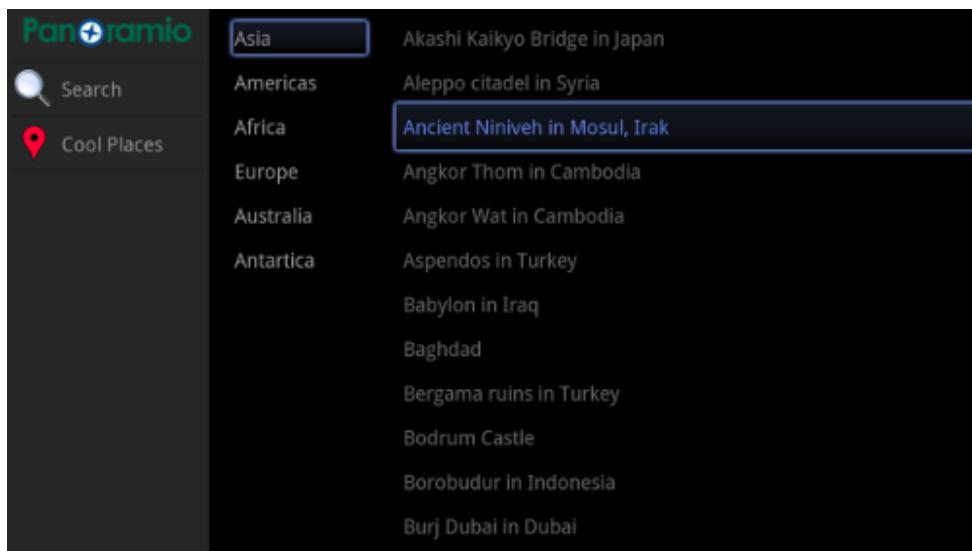
Provide sufficient padding within the focusable and selectable controls so that the highlights around them are clearly visible.

## Design for Easy Navigation

Users should be able to navigate to any UI control with a couple of D-pad clicks. Navigation should be easy and intuitive to understand. For any non-intuitive actions, provide users with written help, using a dialog triggered by a help button or action bar icon.

Predict the next screen that the user will want to navigate to and provide one click navigation to it. If the current screen UI is very sparse, consider making it a multi pane screen. Use fragments for

making multi-pane screens. For example, consider the multi-pane UI below with continent names on the left and list of cool places in each continent on the right.



The above UI consists of three Fragments - `left_side_action_controls`, `continents` and `places` - as shown in its layout xml file below. Such multi-pane UIs make D-pad navigation easier and make good use of the horizontal screen space for TVs.

`res/layout/cool_places.xml`

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    >
    <fragment
        android:id="@+id/left_side_action_controls"
        android:layout_width="0px"
        android:layout_height="match_parent"
        android:layout_marginLeft="10dip"
        android:layout_weight="0.2"/>
    <fragment
        android:id="@+id/continents"
        android:layout_width="0px"
        android:layout_height="match_parent"
        android:layout_marginLeft="10dip"
        android:layout_weight="0.2"/>
    <fragment
        android:id="@+id/places"
        android:layout_width="0px"
        android:layout_height="match_parent"
        android:layout_marginLeft="10dip"
        android:layout_weight="0.6"/>
```

```
</LinearLayout>
```

Also, notice in the UI layout above action controls are on the left hand side of a vertically scrolling list to make them easily accessible using D-pad. In general, for layouts with horizontally scrolling components, place action controls on left or right hand side and vice versa for vertically scrolling components.

# Handling Features Not Supported on TV

TVs are much different from other Android-powered devices:

- They're not mobile.
- Out of habit, people use them for watching media with little or no interaction.
- People interact with them from a distance.

Because TVs have a different purpose from other devices, they usually don't have hardware features that other Android-powered devices often have. For this reason, the Android system does not support the following features for a TV device:

Hardware	Android feature descriptor
Camera	android.hardware.camera
GPS	android.hardware.location.gps
Microphone	android.hardware.microphone
Near Field Communications (NFC)	android.hardware.nfc
Telephony	android.hardware.telephony
Touchscreen	android.hardware.touchscreen

This lesson shows you how to work around features that are not available on TV by:

- Providing workarounds for some non-supported features.
- Checking for available features at runtime and conditionally activating/deactivating certain code paths based on availability of those features.

## Work Around Features Not Supported on TV

Android doesn't support touchscreen interaction for TV devices, most TVs don't have touch screens, and interacting with a TV using a touchscreen is not consistent with the 10 foot environment. For these reasons, users interact with Android-powered TVs using a remote. In consideration of this, ensure that every control in your app can be accessed with the D-pad. Refer back to the previous two lessons [Optimizing Layouts for TV](#) ([/training/tv/optimizing-layouts-tv.html](#)) and [Optimize Navigation for TV](#) ([/training/tv/optimizing-navigation-tv.html](#)) for more details on this topic. The Android system assumes that a device has a touchscreen, so if you want your application to run on a TV, you must explicitly disable the touchscreen requirement in your manifest file:

```
<uses-feature android:name="android.hardware.touchscreen" android:required="false"
```

Although a TV doesn't have a camera, you can still provide a photography-related application on a TV. For example, if you have an app that takes, views and edits photos, you can disable its picture-taking functionality for TVs and still allow users to view and even edit photos. The next section talks about how to deactivate or activate specific functions in the application based on runtime

### THIS LESSON TEACHES YOU TO

1. [Work Around Features Not Supported on TV](#)
2. [Check for Available Features at Runtime](#)

device type detection.

Because TVs are stationary, indoor devices, they don't have built-in GPS. If your application uses location information, allow users to search for a location or use a "static" location provider to get a location from the zip code configured during the TV setup.

```
LocationManager locationManager = (LocationManager) this.getSystemService(Context);
Location location = locationManager.getLastKnownLocation("static");
Geocoder geocoder = new Geocoder(this);
Address address = null;

try {
    address = geocoder.getFromLocation(location.getLatitude(), location.getLongitude());
    Log.d("Zip code", address.getPostalCode());
} catch (IIOException e) {
    Log.e(TAG, "Geocoder error", e);
}
```

TVs usually don't support microphones, but if you have an application that uses voice control, you can create a mobile device app that takes voice input and then acts as a remote control for a TV.

## Check for Available Features at Runtime

To check if a feature is available at runtime, call `hasSystemFeature(String)` ([/reference/android/content/pm/PackageManager.html#hasSystemFeature\(java.lang.String\)](http://reference/android/content/pm/PackageManager.html#hasSystemFeature(java.lang.String))). This method takes a single argument : a string corresponding to the feature you want to check. For example, to check for touchscreen, use `hasSystemFeature(String)` ([/reference/android/content/pm/PackageManager.html#hasSystemFeature\(java.lang.String\)](http://reference/android/content/pm/PackageManager.html#hasSystemFeature(java.lang.String))) with the argument `FEATURE_TOUCHSCREEN` ([/reference/android/content/pm/PackageManager.html#FEATURE\\_TOUCHSCREEN](http://reference/android/content/pm/PackageManager.html#FEATURE_TOUCHSCREEN)).

The following code snippet demonstrates how to detect device type at runtime based on supported features:

```
// Check if android.hardware.telephony feature is available.
if (getPackageManager().hasSystemFeature("android.hardware.telephony")) {
    Log.d("Mobile Test", "Running on phone");
// Check if android.hardware.touchscreen feature is available.
} else if (getPackageManager().hasSystemFeature("android.hardware.touchscreen"))
    Log.d("Tablet Test", "Running on devices that don't support telephony but have");
} else {
    Log.d("TV Test", "Running on a TV!");
}
```

This is just one example of using runtime checks to deactivate app functionality that depends on features that aren't available on TVs.

# 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. If you're not careful, bitmaps can quickly consume your available memory budget leading to an application crash due to the dreaded exception:

```
java.lang.OutOfMemoryError: bitmap size exceeds VM budget.
```

There are a number of reasons why loading bitmaps in your Android application is tricky:

- Mobile devices typically have constrained system resources. Android devices can have as little as 16MB of memory available to a single application. The [Android Compatibility Definition Document \(CDD\)](#), *Section 3.7. Virtual Machine Compatibility* gives the required minimum application memory for various screen sizes and densities. Applications should be optimized to perform under this minimum memory limit. However, keep in mind many devices are configured with higher limits.
- Bitmaps take up a lot of memory, especially for rich images like photographs. For example, the camera on the [Galaxy Nexus](#) takes photos up to 2592x1936 pixels (5 megapixels). If the bitmap configuration used is [ARGB\\_8888](#) (the default from the Android 2.3 onward) then loading this image into memory takes about 19MB of memory ( $2592 \times 1936 \times 4$  bytes), immediately exhausting the per-app limit on some devices.
- Android app UI's frequently require several bitmaps to be loaded at once. Components such as [ListView](#), [GridView](#) and [ViewPager](#) commonly include multiple bitmaps on-screen at once with many more potentially off-screen ready to show at the flick of a finger.

## Lessons

### Loading Large Bitmaps Efficiently

This lesson walks you through decoding large bitmaps without exceeding the per application memory limit.

### Processing Bitmaps Off the UI Thread

Bitmap processing (resizing, downloading from a remote source, etc.) should never take place on the main UI thread. This lesson walks you through processing bitmaps in a background thread using [AsyncTask](#) and explains how to handle concurrency issues.

### Caching Bitmaps

This lesson walks you through using a memory and disk bitmap cache to improve the responsiveness and fluidity of your UI when loading multiple bitmaps.

### Displaying Bitmaps in Your UI

This lesson brings everything together, showing you how to load multiple bitmaps into components like [ViewPager](#) and [GridView](#) using a background thread and bitmap cache.

### DEPENDENCIES AND PREREQUISITES

- Android 2.1 (API Level 7) or higher
- [Support Library](#)

### TRY IT OUT

[Download the sample](#)

BitmapFun.zip

# Loading Large Bitmaps Efficiently

Images come in all shapes and sizes. In many cases they are larger than required for a typical application user interface (UI). For example, the system Gallery application displays photos taken using your Android device's camera which are typically much higher resolution than the screen density of your device.

Given that you are working with limited memory, ideally you only want to load a lower resolution version in memory. The lower resolution version should match the size of the UI component that displays it. An image with a higher resolution does not provide any visible benefit, but still takes up precious memory and incurs additional performance overhead due to additional on the fly scaling.

This lesson walks you through decoding large bitmaps without exceeding the per application memory limit by loading a smaller subsampled version in memory.

## Read Bitmap Dimensions and Type

The [BitmapFactory](#) ([/reference/android/graphics/BitmapFactory.html](#)) class provides several decoding methods ([decodeByteArray\(\)](#) ([/reference/android/graphics/BitmapFactory.html#decodeByteArray\(byte\[\], int, int, android.graphics.BitmapFactory.Options\)](#)), [decodeFile\(\)](#) ([/reference/android/graphics/BitmapFactory.html#decodeFile\(java.lang.String, android.graphics.BitmapFactory.Options\)](#)), [decodeResource\(\)](#) ([/reference/android/graphics/BitmapFactory.html#decodeResource\(android.content.res.Resources, int, android.graphics.BitmapFactory.Options\)](#)), etc.) for creating a [Bitmap](#) ([/reference/android/graphics/Bitmap.html](#)) from various sources. Choose the most appropriate decode method based on your image data source. These methods attempt to allocate memory for the constructed bitmap and therefore can easily result in an `OutOfMemory` exception. Each type of decode method has additional signatures that let you specify decoding options via the [BitmapFactory.Options](#) ([/reference/android/graphics/BitmapFactory.Options.html](#)) class. Setting the [inJustDecodeBounds](#) ([/reference/android/graphics/BitmapFactory.Options.html#inJustDecodeBounds](#)) property to `true` while decoding avoids memory allocation, returning null for the bitmap object but setting [outWidth](#) ([/reference/android/graphics/BitmapFactory.Options.html#outWidth](#)), [outHeight](#) ([/reference/android/graphics/BitmapFactory.Options.html#outHeight](#)) and [outMimeType](#) ([/reference/android/graphics/BitmapFactory.Options.html#outMimeType](#)). This technique allows you to read the dimensions and type of the image data prior to construction (and memory allocation) of the bitmap.

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inJustDecodeBounds = true;
BitmapFactory.decodeResource(getResources(), R.id.myimage, options);
```

### THIS LESSON TEACHES YOU TO

1. [Read Bitmap Dimensions and Type](#)
2. [Load a Scaled Down Version into Memory](#)

### TRY IT OUT

[Download the sample](#)

BitmapFun.zip

```
int imageHeight = options.outHeight;
int imageWidth = options.outWidth;
String imageType = options.outMimeType;
```

To avoid `java.lang.OutOfMemory` exceptions, check the dimensions of a bitmap before decoding it, unless you absolutely trust the source to provide you with predictably sized image data that comfortably fits within the available memory.

## Load a Scaled Down Version into Memory

Now that the image dimensions are known, they can be used to decide if the full image should be loaded into memory or if a subsampled version should be loaded instead. Here are some factors to consider:

- Estimated memory usage of loading the full image in memory.
- Amount of memory you are willing to commit to loading this image given any other memory requirements of your application.
- Dimensions of the target `ImageView` or UI component that the image is to be loaded into.
- Screen size and density of the current device.

For example, it's not worth loading a 1024x768 pixel image into memory if it will eventually be displayed in a 128x96 pixel thumbnail in an `ImageView` ([/reference/android/widget/ImageView.html](#)).

To tell the decoder to subsample the image, loading a smaller version into memory, set `inSampleSize` ([/reference/android/graphics/BitmapFactory.Options.html#inSampleSize](#)) to true in your `BitmapFactory.Options` ([/reference/android/graphics/BitmapFactory.Options.html](#)) object. For example, an image with resolution 2048x1536 that is decoded with an `inSampleSize` ([/reference/android/graphics/BitmapFactory.Options.html#inSampleSize](#)) of 4 produces a bitmap of approximately 512x384. Loading this into memory uses 0.75MB rather than 12MB for the full image (assuming a bitmap configuration of `ARGB_8888` ([/reference/android/graphics/Bitmap.Config.html](#))). Here's a method to calculate a the sample size value based on a target width and height:

```
public static int calculateInSampleSize(
    BitmapFactory.Options options, int reqWidth, int reqHeight) {
    // Raw height and width of image
    final int height = options.outHeight;
    final int width = options.outWidth;
    int inSampleSize = 1;

    if (height > reqHeight || width > reqWidth) {
        if (width > height) {
            inSampleSize = Math.round((float)height / (float)reqHeight);
        } else {
            inSampleSize = Math.round((float)width / (float)reqWidth);
        }
    }
    return inSampleSize;
```

}

Note: Using powers of 2 for `inSampleSize` ([/reference/android/graphics/BitmapFactory.Options.html#inSampleSize](#)) values is faster and more efficient for the decoder. However, if you plan to cache the resized versions in memory or on disk, it's usually still worth decoding to the most appropriate image dimensions to save space.

To use this method, first decode with `inJustDecodeBounds` ([/reference/android/graphics/BitmapFactory.Options.html#inJustDecodeBounds](#)) set to `true`, pass the options through and then decode again using the new `inSampleSize` ([/reference/android/graphics/BitmapFactory.Options.html#inSampleSize](#)) value and `inJustDecodeBounds` ([/reference/android/graphics/BitmapFactory.Options.html#inJustDecodeBounds](#)) set to `false`:

```
public static Bitmap decodeSampledBitmapFromResource(Resources res, int resId,
    int reqWidth, int reqHeight) {

    // First decode with inJustDecodeBounds=true to check dimensions
    final BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(res, resId, options);

    // Calculate inSampleSize
    options.inSampleSize = calculateInSampleSize(options, reqWidth, reqHeight);

    // Decode bitmap with inSampleSize set
    options.inJustDecodeBounds = false;
    return BitmapFactory.decodeResource(res, resId, options);
}
```

This method makes it easy to load a bitmap of arbitrarily large size into an `ImageView` ([/reference/android/widget/ImageView.html](#)) that displays a 100x100 pixel thumbnail, as shown in the following example code:

```
mImageView.setImageBitmap(
    decodeSampledBitmapFromResource(getResources(), R.id.myimage, 100, 100));
```

You can follow a similar process to decode bitmaps from other sources, by substituting the appropriate `BitmapFactory.decode*` ([/reference/android/graphics/BitmapFactory.html#decodeByteArray\(byte\[\], int, int, android.graphics.BitmapFactory.Options\)](#)) method as needed.

# Processing Bitmaps Off the UI Thread

The [BitmapFactory.decode\\*](#) ([/reference/android/graphics/BitmapFactory.html#decodeByteArray\(byte\[\], int, int, android.graphics.BitmapFactory.Options\)](#)) methods, discussed in the [Load Large Bitmaps Efficiently \(load-bitmap.html\)](#) lesson, should not be executed on the main UI thread if the source data is read from disk or a network location (or really any source other than memory). The time this data takes to load is unpredictable and depends on a variety of factors (speed of reading from disk or network, size of image, power of CPU, etc.). If one of these tasks blocks the UI thread, the system flags your application as non-responsive and the user has the option of closing it (see [Designing for Responsiveness \(/guide/practices/responsiveness.html\)](#) for more information).

This lesson walks you through processing bitmaps in a background thread using [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) and shows you how to handle concurrency issues.

## Use an AsyncTask

The [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) class provides an easy way to execute some work in a background thread and publish the results back on the UI thread. To use it, create a subclass and override the provided methods. Here's an example of loading a large image into an [ImageView](#) ([/reference/android/widget/ImageView.html](#)) using [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) and [decodeSampledBitmapFromResource\(\)](#) ([load-bitmap.html#decodeSampledBitmapFromResource](#)):

```
class BitmapWorkerTask extends AsyncTask {
    private final WeakReference imageViewReference;
    private int data = 0;

    public BitmapWorkerTask(ImageView imageView) {
        // Use a WeakReference to ensure the ImageView can be garbage collected
        imageViewReference = new WeakReference(imageView);
    }

    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        data = params[0];
        return decodeSampledBitmapFromResource(getResources(), data, 100, 100);
    }
}
```

### THIS LESSON TEACHES YOU TO

1. [Use an AsyncTask](#)
2. [Handle Concurrency](#)

### YOU SHOULD ALSO READ

- [Designing for Responsiveness](#)
- [Multithreading for Performance](#)

### TRY IT OUT

[Download the sample](#)

BitmapFun.zip

```

}

// Once complete, see if ImageView is still around and set bitmap.
@Override
protected void onPostExecute(Bitmap bitmap) {
    if (imageViewReference != null && bitmap != null) {
        final ImageView imageView = imageViewReference.get();
        if (imageView != null) {
            imageView.setImageBitmap(bitmap);
        }
    }
}
}

```

The [WeakReference](#) ([/reference/java/lang/ref/WeakReference.html](#)) to the [ImageView](#) ([/reference/android/widget/ImageView.html](#)) ensures that the [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) does not prevent the [ImageView](#) ([/reference/android/widget/ImageView.html](#)) and anything it references from being garbage collected. There's no guarantee the [ImageView](#) ([/reference/android/widget/ImageView.html](#)) is still around when the task finishes, so you must also check the reference in [onPostExecute\(\)](#) ([/reference/android/os/AsyncTask.html#onPostExecute\(Result\)](#)). The [ImageView](#) ([/reference/android/widget/ImageView.html](#)) may no longer exist, if for example, the user navigates away from the activity or if a configuration change happens before the task finishes.

To start loading the bitmap asynchronously, simply create a new task and execute it:

```

public void loadBitmap(int resId, ImageView imageView) {
    BitmapWorkerTask task = new BitmapWorkerTask(imageView);
    task.execute(resId);
}

```

## Handle Concurrency

Common view components such as [ListView](#) ([/reference/android/widget/ListView.html](#)) and [GridView](#) ([/reference/android/widget/GridView.html](#)) introduce another issue when used in conjunction with the [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) as demonstrated in the previous section. In order to be efficient with memory, these components recycle child views as the user scrolls. If each child view triggers an [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)), there is no guarantee that when it completes, the associated view has not already been recycled for use in another child view. Furthermore, there is no guarantee that the order in which asynchronous tasks are started is the order that they complete.

The blog post [Multithreading for Performance](#) (<http://android-developers.blogspot.com/2010/07/multithreading-for-performance.html>) further discusses dealing with concurrency, and offers a solution where the [ImageView](#) ([/reference/android/widget/ImageView.html](#)) stores a reference to the most recent [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) which can later be checked when the task completes. Using a similar method, the [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) from the previous section can be extended to follow a similar pattern.

Create a dedicated [Drawable](#) ([/reference/android/graphics/drawable/Drawable.html](#)) subclass to store a reference back to the worker task. In this case, a [BitmapDrawable](#) ([/reference/android/graphics/drawable/BitmapDrawable.html](#)) is used so that a placeholder image can be displayed in the [ImageView](#) ([/reference/android/widget/ImageView.html](#)) while the task completes:

```
static class AsyncDrawable extends BitmapDrawable {
    private final WeakReference<BitmapWorkerTask> bitmapWorkerTaskReference;

    public AsyncDrawable(Resources res, Bitmap bitmap,
                         BitmapWorkerTask bitmapWorkerTask) {
        super(res, bitmap);
        bitmapWorkerTaskReference =
            new WeakReference<BitmapWorkerTask>(bitmapWorkerTask);
    }

    public BitmapWorkerTask getBitmapWorkerTask() {
        return bitmapWorkerTaskReference.get();
    }
}
```

Before executing the [BitmapWorkerTask](#) (#[BitmapWorkerTask](#)), you create an [AsyncDrawable](#) (#[AsyncDrawable](#)) and bind it to the target [ImageView](#) ([/reference/android/widget/ImageView.html](#)):

```
public void loadBitmap(int resId, ImageView imageView) {
    if (cancelPotentialWork(resId, imageView)) {
        final BitmapWorkerTask task = new BitmapWorkerTask(imageView);
        final AsyncDrawable asyncDrawable =
            new AsyncDrawable(getResources(), mPlaceHolderBitmap, task);
        imageView.setImageDrawable(asyncDrawable);
        task.execute(resId);
    }
}
```

The [cancelPotentialWork](#) method referenced in the code sample above checks if another running task is already associated with the [ImageView](#) ([/reference/android/widget/ImageView.html](#)). If so, it attempts to cancel the previous task by calling [cancel\(\)](#) ([/reference/android/os/AsyncTask.html#cancel\(boolean\)](#)). In a small number of cases, the new task data matches the existing task and nothing further needs to happen. Here is the implementation of [cancelPotentialWork](#):

```
public static boolean cancelPotentialWork(int data, ImageView imageView) {
    final BitmapWorkerTask bitmapWorkerTask = getBitmapWorkerTask(imageView);

    if (bitmapWorkerTask != null) {
        final int bitmapData = bitmapWorkerTask.data;
        if (bitmapData != data) {
```

```

        // Cancel previous task
        bitmapWorkerTask.cancel(true);
    } else {
        // The same work is already in progress
        return false;
    }
}
// No task associated with the ImageView, or an existing task was cancelled
return true;
}

```

A helper method, `getBitmapWorkerTask()`, is used above to retrieve the task associated with a particular [ImageView](#) ([/reference/android/widget/ImageView.html](#)):

```

private static BitmapWorkerTask getBitmapWorkerTask(ImageView imageView) {
    if (imageView != null) {
        final Drawable drawable = imageView.getDrawable();
        if (drawable instanceof AsyncDrawable) {
            final AsyncDrawable asyncDrawable = (AsyncDrawable) drawable;
            return asyncDrawable.getBitmapWorkerTask();
        }
    }
    return null;
}

```

The last step is updating `onPostExecute()` in [BitmapWorkerTask \(#BitmapWorkerTask\)](#) so that it checks if the task is cancelled and if the current task matches the one associated with the [ImageView](#) ([/reference/android/widget/ImageView.html](#)):

```

class BitmapWorkerTask extends AsyncTask {
    ...
    @Override
    protected void onPostExecute(Bitmap bitmap) {
        if (isCancelled()) {
            bitmap = null;
        }

        if (imageViewReference != null && bitmap != null) {
            final ImageView imageView = imageViewReference.get();
            final BitmapWorkerTask bitmapWorkerTask =
                getBitmapWorkerTask(imageView);
            if (this == bitmapWorkerTask && imageView != null) {
                imageView.setImageBitmap(bitmap);
            }
        }
    }
}

```

```
}
```

This implementation is now suitable for use in [ListView](#) ([/reference/android/widget/ListView.html](#)) and [GridView](#) ([/reference/android/widget/GridView.html](#)) components as well as any other components that recycle their child views. Simply call `loadBitmap` where you normally set an image to your [ImageView](#) ([/reference/android/widget/ImageView.html](#)). For example, in a [GridView](#) ([/reference/android/widget/GridView.html](#)) implementation this would be in the `getView()` ([/reference/android/widget/Adapter.html# getView\(int, android.view.View, android.view.ViewGroup\)](#)) method of the backing adapter.

# Caching Bitmaps

Loading a single bitmap into your user interface (UI) is straightforward, however things get more complicated if you need to load a larger set of images at once. In many cases (such as with components like [ListView](#) ([/reference/android/widget/ListView.html](#)), [GridView](#) ([/reference/android/widget/GridView.html](#)) or [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#))), the total number of images on-screen combined with images that might soon scroll onto the screen are essentially unlimited.

Memory usage is kept down with components like this by recycling the child views as they move off-screen. The garbage collector also frees up your loaded bitmaps, assuming you don't keep any long lived references. This is all good and well, but in order to keep a fluid and fast-loading UI you want to avoid continually processing these images each time they come back on-screen. A memory and disk cache can often help here, allowing components to quickly reload processed images.

This lesson walks you through using a memory and disk bitmap cache to improve the responsiveness and fluidity of your UI when loading multiple bitmaps.

## Use a Memory Cache

A memory cache offers fast access to bitmaps at the cost of taking up valuable application memory. The [LruCache](#) ([/reference/android/util/LruCache.html](#)) class (also available in the [Support Library](#) ([/reference/android/support/v4/util/LruCache.html](#)) for use back to API Level 4) is particularly well suited to the task of caching bitmaps, keeping recently referenced objects in a strong referenced [LinkedHashMap](#) ([/reference/java/util/LinkedHashMap.html](#)) and evicting the least recently used member before the cache exceeds its designated size.

Note: In the past, a popular memory cache implementation was a [SoftReference](#) ([/reference/java/lang/ref/SoftReference.html](#)) or [WeakReference](#) ([/reference/java/lang/ref/WeakReference.html](#)) bitmap cache, however this is not recommended. Starting from Android 2.3 (API Level 9) the garbage collector is more aggressive with collecting soft/weak references which makes them fairly ineffective. In addition, prior to Android 3.0 (API Level 11), the backing data of a bitmap was stored in native memory which is not released in a predictable manner, potentially causing an application to briefly exceed its memory limits and crash.

In order to choose a suitable size for a [LruCache](#) ([/reference/android/util/LruCache.html](#)), a number of factors should be taken into consideration, for example:

- How memory intensive is the rest of your activity and/or application?

### THIS LESSON TEACHES YOU TO

1. [Use a Memory Cache](#)
2. [Use a Disk Cache](#)
3. [Handle Configuration Changes](#)

### YOU SHOULD ALSO READ

- [Handling Runtime Changes](#)

### TRY IT OUT

[Download the sample](#)

BitmapFun.zip

- How many images will be on-screen at once? How many need to be available ready to come on-screen?
- What is the screen size and density of the device? An extra high density screen (xhdpi) device like Galaxy Nexus will need a larger cache to hold the same number of images in memory compared to a device like Nexus S (hdpi).
- What dimensions and configuration are the bitmaps and therefore how much memory will each take up?
- How frequently will the images be accessed? Will some be accessed more frequently than others? If so, perhaps you may want to keep certain items always in memory or even have multiple LruCache objects for different groups of bitmaps.
- Can you balance quality against quantity? Sometimes it can be more useful to store a larger number of lower quality bitmaps, potentially loading a higher quality version in another background task.

There is no specific size or formula that suits all applications, it's up to you to analyze your usage and come up with a suitable solution. A cache that is too small causes additional overhead with no benefit, a cache that is too large can once again cause `java.lang.OutOfMemory` exceptions and leave the rest of your app little memory to work with.

Here's an example of setting up a LruCache (</reference/android/util/LruCache.html>) for bitmaps:

```
private LruCache mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Get memory class of this device, exceeding this amount will throw an
    // OutOfMemory exception.
    final int memClass = ((ActivityManager) context.getSystemService(
        Context.ACTIVITY_SERVICE)).getMemoryClass();

    // Use 1/8th of the available memory for this memory cache.
    final int cacheSize = 1024 * 1024 * memClass / 8;

    mMemoryCache = new LruCache(cacheSize) {
        @Override
        protected int sizeOf(String key, Bitmap bitmap) {
            // The cache size will be measured in bytes rather than number of items
            return bitmap.getByteCount();
        }
    };
    ...
}

public void addBitmapToMemoryCache(String key, Bitmap bitmap) {
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }
}
```

```
public Bitmap getBitmapFromMemCache(String key) {  
    return mMemoryCache.get(key);  
}
```

Note: In this example, one eighth of the application memory is allocated for our cache. On a normal/hdpi device this is a minimum of around 4MB (32/8). A full screen [GridView](#) ([/reference/android/widget/GridView.html](#)) filled with images on a device with 800x480 resolution would use around 1.5MB (800\*480\*4 bytes), so this would cache a minimum of around 2.5 pages of images in memory.

When loading a bitmap into an [ImageView](#) ([/reference/android/widget/ImageView.html](#)), the [LruCache](#) ([/reference/android/util/LruCache.html](#)) is checked first. If an entry is found, it is used immediately to update the [ImageView](#) ([/reference/android/widget/ImageView.html](#)), otherwise a background thread is spawned to process the image:

```
public void loadBitmap(int resId, ImageView imageView) {  
    final String imageKey = String.valueOf(resId);  
  
    final Bitmap bitmap = getBitmapFromMemCache(imageKey);  
    if (bitmap != null) {  
        mImageView.setImageBitmap(bitmap);  
    } else {  
        mImageView.setImageResource(R.drawable.image_placeholder);  
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);  
        task.execute(resId);  
    }  
}
```

The [BitmapWorkerTask](#) ([process-bitmap.html#BitmapWorkerTask](#)) also needs to be updated to add entries to the memory cache:

```
class BitmapWorkerTask extends AsyncTask {  
    ...  
    // Decode image in background.  
    @Override  
    protected Bitmap doInBackground(Integer... params) {  
        final Bitmap bitmap = decodeSampledBitmapFromResource(  
            getResources(), params[0], 100, 100));  
        addBitmapToMemoryCache(String.valueOf(params[0]), bitmap);  
        return bitmap;  
    }  
    ...  
}
```

## Use a Disk Cache

A memory cache is useful in speeding up access to recently viewed bitmaps, however you cannot rely on images being available in this cache. Components like [GridView](#) ([/reference/android/widget/GridView.html](#)) with larger datasets can easily fill up a memory cache. Your application could be interrupted by another task like a phone call, and while in the background it might be killed and the memory cache destroyed. Once the user resumes, your application it has to process each image again.

A disk cache can be used in these cases to persist processed bitmaps and help decrease loading times where images are no longer available in a memory cache. Of course, fetching images from disk is slower than loading from memory and should be done in a background thread, as disk read times can be unpredictable.

**Note:** A [ContentProvider](#) ([/reference/android/content/ContentProvider.html](#)) might be a more appropriate place to store cached images if they are accessed more frequently, for example in an image gallery application.

Included in the sample code of this class is a basic DiskLruCache implementation. However, a more robust and recommended DiskLruCache solution is included in the Android 4.0 source code ([libcore/luni/src/main/java/libcore/io/DiskLruCache.java](#)). Back-porting this class for use on previous Android releases should be fairly straightforward (a [quick search](#) (<http://www.google.com/search?q=disklrucache>) shows others who have already implemented this solution).

Here's updated example code that uses the simple DiskLruCache included in the sample application of this class:

```
private DiskLruCache mDiskCache;
private static final int DISK_CACHE_SIZE = 1024 * 1024 * 10; // 10MB
private static final String DISK_CACHE_SUBDIR = "thumbnails";

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Initialize memory cache
    ...
    File cacheDir = getCacheDir(this, DISK_CACHE_SUBDIR);
    mDiskCache = DiskLruCache.openCache(this, cacheDir, DISK_CACHE_SIZE);
    ...
}

class BitmapWorkerTask extends AsyncTask {
    ...
    // Decode image in background.
    @Override
    protected Bitmap doInBackground(Integer... params) {
        final String imageKey = String.valueOf(params[0]);
        ...
        // Check disk cache in background thread
        Bitmap bitmap = getBitmapFromDiskCache(imageKey);
    }
}
```

```
if (bitmap == null) { // Not found in disk cache
    // Process as normal
    final Bitmap bitmap = decodeSampledBitmapFromResource(
        getResources(), params[0], 100, 100));
}

// Add final bitmap to caches
addBitmapToCache(String.valueOf(imageKey, bitmap));

return bitmap;
}
...
}

public void addBitmapToCache(String key, Bitmap bitmap) {
    // Add to memory cache as before
    if (getBitmapFromMemCache(key) == null) {
        mMemoryCache.put(key, bitmap);
    }

    // Also add to disk cache
    if (!mDiskCache.containsKey(key)) {
        mDiskCache.put(key, bitmap);
    }
}

public Bitmap getBitmapFromDiskCache(String key) {
    return mDiskCache.get(key);
}

// Creates a unique subdirectory of the designated app cache directory. Tries to
// but if not mounted, falls back on internal storage.
public static File getCacheDir(Context context, String uniqueName) {
    // Check if media is mounted or storage is built-in, if so, try and use exten
    // otherwise use internal cache dir
    final String cachePath = Environment.getExternalStorageState() == Environment
        || !Environment.isExternalStorageRemovable() ?
            context.getExternalCacheDir().getPath() : context.getCacheDir

    return new File(cachePath + File.separator + uniqueName);
}
```

While the memory cache is checked in the UI thread, the disk cache is checked in the background thread. Disk operations should never take place on the UI thread. When image processing is complete, the final bitmap is added to both the memory and disk cache for future use.

## Handle Configuration Changes

Runtime configuration changes, such as a screen orientation change, cause Android to destroy and restart the running activity with the new configuration (For more information about this behavior, see [Handling Runtime Changes \(/guide/topics/resources/runtime-changes.html\)](#)). You want to avoid having to process all your images again so the user has a smooth and fast experience when a configuration change occurs.

Luckily, you have a nice memory cache of bitmaps that you built in the [Use a Memory Cache \(#memory-cache\)](#) section. This cache can be passed through to the new activity instance using a [Fragment \(/reference/android/app/Fragment.html\)](#) which is preserved by calling [setRetainInstance\(true\) \(/reference/android/app/Fragment.html#setRetainInstance\(boolean\)\)](#). After the activity has been recreated, this retained [Fragment \(/reference/android/app/Fragment.html\)](#) is reattached and you gain access to the existing cache object, allowing images to be quickly fetched and re-populated into the [ImageView \(/reference/android/widget/ImageView.html\)](#) objects.

Here's an example of retaining a [LruCache \(/reference/android/util/LruCache.html\)](#) object across configuration changes using a [Fragment \(/reference/android/app/Fragment.html\)](#):

```
private LruCache mMemoryCache;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    RetainFragment mRetainFragment =
        RetainFragment.findOrCreateRetainFragment(getFragmentManager());
    mMemoryCache = mRetainFragment.mRetainedCache;
    if (mMemoryCache == null) {
        mMemoryCache = new LruCache(cacheSize) {
            ... // Initialize cache here as usual
        };
        mRetainFragment.mRetainedCache = mMemoryCache;
    }
    ...
}

class RetainFragment extends Fragment {
    private static final String TAG = "RetainFragment";
    public LruCache mRetainedCache;

    public RetainFragment() {}

    public static RetainFragment findOrCreateRetainFragment(FragmentManager fm) {
        RetainFragment fragment = (RetainFragment) fm.findFragmentByTag(TAG);
        if (fragment == null) {
            fragment = new RetainFragment();
        }
        return fragment;
    }

    @Override
```

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setRetainInstance(true);  
}
```

To test this out, try rotating a device both with and without retaining the [Fragment](#) ([/reference/android/app/Fragment.html](#)). You should notice little to no lag as the images populate the activity almost instantly from memory when you retain the cache. Any images not found in the memory cache are hopefully available in the disk cache, if not, they are processed as usual.

# Displaying Bitmaps in Your UI

This lesson brings together everything from previous lessons, showing you how to load multiple bitmaps into [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#)) and [GridView](#) ([/reference/android/widget/GridView.html](#)) components using a background thread and bitmap cache, while dealing with concurrency and configuration changes.

## Load Bitmaps into a ViewPager Implementation

The [swipe view pattern](#) ([/design/patterns/swipe-views.html](#)) is an excellent way to navigate the detail view of an image gallery. You can implement this pattern using a [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#)) component backed by a [PagerAdapter](#) ([/reference/android/support/v4/view/PagerAdapter.html](#)).

However, a more suitable backing adapter is the subclass [FragmentStatePagerAdapter](#) ([/reference/android/support/v4/app/FragmentStatePagerAdapter.html](#)) which automatically destroys and saves state of the [Fragments](#) ([/reference/android/app/Fragment.html](#)) in the [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#)) as they disappear off-screen, keeping memory usage down.

Note: If you have a smaller number of images and are confident they all fit within the application memory limit, then using a regular [PagerAdapter](#) ([/reference/android/support/v4/view/PagerAdapter.html](#)) or [FragmentPagerAdapter](#) ([/reference/android/support/v4/app/FragmentPagerAdapter.html](#)) might be more appropriate.

Here's an implementation of a [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#)) with [ImageView](#) ([/reference/android/widget/ImageView.html](#)) children. The main activity holds the [ViewPager](#) ([/reference/android/support/v4/view/ViewPager.html](#)) and the adapter:

```
public class ImageDetailActivity extends FragmentActivity {
    public static final String EXTRA_IMAGE = "extra_image";

    private ImagePagerAdapter mAdapter;
    private ViewPager mPager;

    // A static dataset to back the ViewPager adapter
    public final static Integer[] imageResIds = new Integer[] {
        R.drawable.sample_image_1, R.drawable.sample_image_2, R.drawable.sample_image_3,
        R.drawable.sample_image_4, R.drawable.sample_image_5, R.drawable.sample_image_6,
        R.drawable.sample_image_7, R.drawable.sample_image_8, R.drawable.sample_image_9
    };
}
```

### THIS LESSON TEACHES YOU TO

1. [Load Bitmaps into a ViewPager Implementation](#)
2. [Load Bitmaps into a GridView Implementation](#)

### YOU SHOULD ALSO READ

- [Android Design: Swipe Views](#)
- [Android Design: Grid Lists](#)

### TRY IT OUT

[Download the sample](#)

BitmapFun.zip

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.image_detail_pager); // Contains just a ViewPager

    mAdapter = new ImagePagerAdapter(getSupportFragmentManager(), imageResIds
    mPager = (ViewPager) findViewById(R.id.pager);
    mPager.setAdapter(mAdapter);
}

public static class ImagePagerAdapter extends FragmentStatePagerAdapter {
    private final int mSize;

    public ImagePagerAdapter(FragmentManager fm, int size) {
        super(fm);
        mSize = size;
    }

    @Override
    public int getCount() {
        return mSize;
    }

    @Override
    public Fragment getItem(int position) {
        return ImageDetailFragment.newInstance(position);
    }
}
}
```

The details [Fragment](#) ([/reference/android/app/Fragment.html](#)) holds the [ImageView](#) ([/reference/android/widget/ImageView.html](#)) children:

```
public class ImageDetailFragment extends Fragment {
    private static final String IMAGE_DATA_EXTRA = "resId";
    private int mImageNum;
    private ImageView mImageView;

    static ImageDetailFragment newInstance(int imageNum) {
        final ImageDetailFragment f = new ImageDetailFragment();
        final Bundle args = new Bundle();
        args.putInt(IMAGE_DATA_EXTRA, imageNum);
        f.setArguments(args);
        return f;
    }

    // Empty constructor, required as per Fragment docs
```

```
public ImageDetailFragment() {}

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mImageNum = getArguments() != null ? getArguments().getInt(IMAGE_DATA_EXT)
}

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    // image_detail_fragment.xml contains just an ImageView
    final View v = inflater.inflate(R.layout.image_detail_fragment, container,
        mImageView = (ImageView) v.findViewById(R.id.imageView);
    return v;
}

@Override
public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    final int resId = ImageDetailActivity.imageResIds[mImageNum];
    mImageView.setImageResource(resId); // Load image into ImageView
}
}
```

Hopefully you noticed the issue with this implementation; The images are being read from resources on the UI thread which can lead to an application hanging and being force closed. Using an [AsyncTask](#) ([/reference/android/os/AsyncTask.html](#)) as described in the [Processing Bitmaps Off the UI Thread](#) ([process-bitmap.html](#)) lesson, it's straightforward to move image loading and processing to a background thread:

```
public class ImageDetailActivity extends FragmentActivity {
    ...

    public void loadBitmap(int resId, ImageView imageView) {
        mImageView.setImageResource(R.drawable.image_placeholder);
        BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
        task.execute(resId);
    }

    ... // include BitmapWorkerTask class
}

public class ImageDetailFragment extends Fragment {
    ...

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
```

```
super.onCreate(savedInstanceState);
if (ImageDetailActivity.class.isInstance(getActivity())) {
    final int resId = ImageDetailActivity.imageResIds[mImageNum];
    // Call out to ImageDetailActivity to load the bitmap in a background
    ((ImageDetailActivity) getActivity()).loadBitmap(resId, mImageView);
}
}
```

Any additional processing (such as resizing or fetching images from the network) can take place in the [BitmapWorkerTask \(process-bitmap.html#BitmapWorkerTask\)](#) without affecting responsiveness of the main UI. If the background thread is doing more than just loading an image directly from disk, it can also be beneficial to add a memory and/or disk cache as described in the lesson [Caching Bitmaps \(cache-bitmap.html#memory-cache\)](#). Here's the additional modifications for a memory cache:

```
public class ImageDetailActivity extends FragmentActivity {
    ...
    private LruCache mMemoryCache;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        // initialize LruCache as per Use a Memory Cache section
    }

    public void loadBitmap(int resId, ImageView imageView) {
        final String imageKey = String.valueOf(resId);

        final Bitmap bitmap = mMemoryCache.get(imageKey);
        if (bitmap != null) {
            imageView.setImageBitmap(bitmap);
        } else {
            imageView.setImageResource(R.drawable.image_placeholder);
            BitmapWorkerTask task = new BitmapWorkerTask(mImageView);
            task.execute(resId);
        }
    }

    ... // include updated BitmapWorkerTask from Use a Memory Cache section
}
```

Putting all these pieces together gives you a responsive [ViewPager \(/reference/android/support/v4/view/ViewPager.html\)](#) implementation with minimal image loading latency and the ability to do as much or as little background processing on your images as needed.

## Load Bitmaps into a GridView Implementation

The [grid list building block](#) ([/design/building-blocks/grid-lists.html](#)) is useful for showing image data sets and can be implemented using a [GridView](#) ([/reference/android/widget/GridView.html](#)) component in which many images can be on-screen at any one time and many more need to be ready to appear if the user scrolls up or down. When implementing this type of control, you must ensure the UI remains fluid, memory usage remains under control and concurrency is handled correctly (due to the way [GridView](#) ([/reference/android/widget/GridView.html](#)) recycles its children views).

To start with, here is a standard [GridView](#) ([/reference/android/widget/GridView.html](#)) implementation with [ImageView](#) ([/reference/android/widget/ImageView.html](#)) children placed inside a [Fragment](#) ([/reference/android/app/Fragment.html](#)):

```
public class ImageGridFragment extends Fragment implements AdapterView.OnItemClickListener
    private ImageAdapter mAdapter;

    // A static dataset to back the GridView adapter
    public final static Integer[] imageResIds = new Integer[] {
        R.drawable.sample_image_1, R.drawable.sample_image_2, R.drawable.sample_image_3,
        R.drawable.sample_image_4, R.drawable.sample_image_5, R.drawable.sample_image_6,
        R.drawable.sample_image_7, R.drawable.sample_image_8, R.drawable.sample_image_9
    };

    // Empty constructor as per Fragment docs
    public ImageGridFragment() {}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mAdapter = new ImageAdapter(getActivity());
    }

    @Override
    public View onCreateView(
            LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
    {
        final View v = inflater.inflate(R.layout.image_grid_fragment, container, false);
        final GridView mGridView = (GridView) v.findViewById(R.id.gridView);
        mGridView.setAdapter(mAdapter);
        mGridView.setOnItemClickListener(this);
        return v;
    }

    @Override
    public void onItemClick(AdapterView parent, View v, int position, long id) {
        final Intent i = new Intent(getActivity(), ImageDetailActivity.class);
        i.putExtra(ImageDetailActivity.EXTRA_IMAGE, position);
        startActivity(i);
    }

    private class ImageAdapter extends BaseAdapter {
        private final Context mContext;
```

```
public ImageAdapter(Context context) {
    super();
    mContext = context;
}

@Override
public int getCount() {
    return imageResIds.length;
}

@Override
public Object getItem(int position) {
    return imageResIds[position];
}

@Override
public long getItemId(int position) {
    return position;
}

@Override
public View getView(int position, View convertView, ViewGroup container)
    ImageView imageView;
    if (convertView == null) { // if it's not recycled, initialize some a
        imageView = new ImageView(mContext);
        imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
        imageView.setLayoutParams(new GridView.LayoutParams(
            LayoutParams.MATCH_PARENT, LayoutParams.MATCH_PARENT));
    } else {
        imageView = (ImageView) convertView;
    }
    imageView.setImageResource(imageResIds[position]); // Load image into
    return imageView;
}
}
```

Once again, the problem with this implementation is that the image is being set in the UI thread. While this may work for small, simple images (due to system resource loading and caching), if any additional processing needs to be done, your UI grinds to a halt.

The same asynchronous processing and caching methods from the previous section can be implemented here. However, you also need to wary of concurrency issues as the [GridView](#) ([/reference/android/widget/GridView.html](#)) recycles its children views. To handle this, use the techniques discussed in the [Processing Bitmaps Off the UI Thread](#) ([process-bitmap.html#concurrency](#)) lesson. Here is the updated solution:

```
public class ImageGridFragment extends Fragment implements AdapterView.OnItemClickListener
```

```
...  
  
private class ImageAdapter extends BaseAdapter {  
    ...  
  
    @Override  
    public View getView(int position, View convertView, ViewGroup container)  
        ...  
        loadBitmap(imageResIds[position], imageView)  
        return imageView;  
    }  
}  
  
public void loadBitmap(int resId, ImageView imageView) {  
    if (cancelPotentialWork(resId, imageView)) {  
        final BitmapWorkerTask task = new BitmapWorkerTask(imageView);  
        final AsyncDrawable asyncDrawable =  
            new AsyncDrawable(getResources(), mPlaceHolderBitmap, task);  
        imageView.setImageDrawable(asyncDrawable);  
        task.execute(resId);  
    }  
}  
  
static class AsyncDrawable extends BitmapDrawable {  
    private final WeakReference bitmapWorkerTaskReference;  
  
    public AsyncDrawable(Resources res, Bitmap bitmap,  
        BitmapWorkerTask bitmapWorkerTask) {  
        super(res, bitmap);  
        bitmapWorkerTaskReference =  
            new WeakReference(bitmapWorkerTask);  
    }  
  
    public BitmapWorkerTask getBitmapWorkerTask() {  
        return bitmapWorkerTaskReference.get();  
    }  
}  
  
public static boolean cancelPotentialWork(int data, ImageView imageView) {  
    final BitmapWorkerTask bitmapWorkerTask = getBitmapWorkerTask(imageView);  
  
    if (bitmapWorkerTask != null) {  
        final int bitmapData = bitmapWorkerTask.data;  
        if (bitmapData != data) {  
            // Cancel previous task  
            bitmapWorkerTask.cancel(true);  
        } else {  
            // The same work is already in progress  
            return false;  
        }  
    }  
}
```

```
        }
    }
    // No task associated with the ImageView, or an existing task was cancellable
    return true;
}

private static BitmapWorkerTask getBitmapWorkerTask(ImageView imageView) {
    if (imageView != null) {
        final Drawable drawable = imageView.getDrawable();
        if (drawable instanceof AsyncDrawable) {
            final AsyncDrawable asyncDrawable = (AsyncDrawable) drawable;
            return asyncDrawable.getBitmapWorkerTask();
        }
    }
    return null;
}

... // include updated BitmapWorkerTask class
```

Note: The same code can easily be adapted to work with [ListView](#) ([/reference/android/widget/ListView.html](#)) as well.

This implementation allows for flexibility in how the images are processed and loaded without impeding the smoothness of the UI. In the background task you can load images from the network or resize large digital camera photos and the images appear as the tasks finish processing.

For a full example of this and other concepts discussed in this lesson, please see the included sample application.

# Implementing Accessibility

When it comes to reaching as wide a userbase as possible, it's important to pay attention to accessibility in your Android application. Cues in your user interface that may work for a majority of users, such as a visible change in state when a button is pressed, can be less optimal if the user is visually impaired.

This class shows you how to make the most of the accessibility features built into the Android framework. It covers how to optimize your app for accessibility, leveraging platform features like focus navigation and content descriptions. It also covers how to build accessibility services, that can facilitate user interaction with any Android application, not just your own.

## Lessons

---

### **Developing Accessible Applications**

Learn to make your Android application accessible. Allow for easy navigation with a keyboard or directional pad, set labels and fire events that can be interpreted by an accessibility service to facilitate a smooth user experience.

### **Developing Accessibility Services**

Develop an accessibility service that listens for accessibility events, mines those events for information like event type and content descriptions, and uses that information to communicate with the user. The example will use a text-to-speech engine to speak to the user.

---

### **DEPENDENCIES AND PREREQUISITES**

---

- Android 2.0 (API Level 5) or higher

---

### **YOU SHOULD ALSO READ**

---

- [Accessibility](#)

# Developing Accessible Applications

Android has several accessibility-focused features baked into the platform, which make it easy to optimize your application for those with visual or physical disabilities. However, it's not always obvious what the correct optimizations are, or the easiest way to leverage the framework toward this purpose. This lesson shows you how to implement the strategies and platform features that make for a great accessibility-enabled Android application.

## THIS LESSON TEACHES YOU TO

1. [Add Content Descriptions](#)
2. [Design for Focus Navigation](#)
3. [Fire Accessibility Events](#)
4. [Test Your Application](#)

## YOU SHOULD ALSO READ

- [Making Applications Accessible](#)

## Add Content Descriptions

A well-designed user interface (UI) often has elements that don't require an explicit label to indicate their purpose to the user. A checkbox next to an item in a task list application has a fairly obvious purpose, as does a trash can in a file manager application. However, to your users with vision impairment, other UI cues are needed.

Fortunately, it's easy to add labels to UI elements in your application that can be read out loud to your user by a speech-based accessibility service like [TalkBack](https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback) (<https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback>) . If you have a label that's likely not to change during the lifecycle of the application (such as "Pause" or "Purchase"), you can add it via the XML layout, by setting a UI element's [android:contentDescription](#) ([/reference/android/view/View.html#attr\\_android:contentDescription](#)) attribute, like in this example:

```
<Button  
    android:id="@+id/pause_button"  
    android:src="@drawable/pause"  
    android:contentDescription="@string/pause"/>
```

However, there are plenty of situations where it's desirable to base the content description on some context, such as the state of a toggle button, or a piece selectable data like a list item. To edit the content description at runtime, use the [setContentDescription\(\)](#) ([/reference/android/view/View.html#setContentDescription\(java.lang.CharSequence\)](#)) method, like this:

```
String contentDescription = "Select " + strValues[position];  
label.setContentDescription(contentDescription);
```

This addition to your code is the simplest accessibility improvement you can make to your application, but one of the most useful. Try to add content descriptions wherever there's useful information, but avoid the web-developer pitfall of labelling *everything* with useless information. For instance, don't set an application icon's content description to "app icon". That just increases the noise a user needs to navigate in order to pull useful information from your interface.

Try it out! Download [TalkBack](https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback) (<https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback>) (an accessibility service published by Google) and enable it in Settings > Accessibility > TalkBack. Then navigate around your own application and listen for the audible cues provided by TalkBack.

## Design for Focus Navigation

---

Your application should support more methods of navigation than the touch screen alone. Many Android devices come with navigation hardware other than the touchscreen, like a D-Pad, arrow keys, or a trackball. In addition, later Android releases also support connecting external devices like keyboards via USB or bluetooth.

In order to enable this form of navigation, all navigational elements that the user should be able to navigate to need to be set as focusable. This modification can be done at runtime using the [View.setFocusable\(\)](#) ([/reference/android/view/View.html#setFocusable\(boolean\)](#)) method on that UI control, or by setting the [android:focusable](#) ([/reference/android/view/View.html#attr\\_android:focusable](#)) attribute in your XML layout files.

Also, each UI control has 4 attributes, [android:nextFocusUp](#) ([/reference/android/view/View.html#attr\\_android:nextFocusUp](#)), [android:nextFocusDown](#) ([/reference/android/view/View.html#attr\\_android:nextFocusDown](#)), [android:nextFocusLeft](#) ([/reference/android/view/View.html#attr\\_android:nextFocusLeft](#)), and [android:nextFocusRight](#) ([/reference/android/view/View.html#attr\\_android:nextFocusRight](#)), which you can use to designate the next view to receive focus when the user navigates in that direction. While the platform determines navigation sequences automatically based on layout proximity, you can use these attributes to override that sequence if it isn't appropriate in your application.

For instance, here's how you represent a button and label, both focusable, such that pressing down takes you from the button to the text view, and pressing up would take you back to the button.

```
<Button android:id="@+id/doSomething"
    android:focusable="true"
    android:nextFocusDown="@+id/label"
    ... />
<TextView android:id="@+id/label"
    android:focusable="true"
    android:text="@string/labelText"
    android:nextFocusUp="@+id/doSomething"
    ... />
```

Verify that your application works intuitively in these situations. The easiest way is to simply run your application in the Android emulator, and navigate around the UI with the emulator's arrow keys, using the OK button as a replacement for touch to select UI controls.

## Fire Accessibility Events

---

If you're using the view components in the Android framework, an [AccessibilityEvent](#) ([/reference/android/view/accessibility/AccessibilityEvent.html](#)) is created whenever you select an

item or change focus in your UI. These events are examined by the accessibility service, enabling it to provide features like text-to-speech to the user.

If you write a custom view, make sure it fires events at the appropriate times. Generate events by calling [sendAccessibilityEvent\(int\)](#) ([/reference/android/view/View.html#sendAccessibilityEvent\(int\)](#)), with a parameter representing the type of event that occurred. A complete list of the event types currently supported can be found in the [AccessibilityEvent](#) ([/reference/android/view/accessibility/AccessibilityEvent.html](#)) reference documentation.

As an example, if you want to extend an image view such that you can write captions by typing on the keyboard when it has focus, it makes sense to fire an [TYPE\\_VIEW\\_TEXT\\_CHANGED](#) ([/reference/android/view/accessibility/AccessibilityEvent.html#TYPE\\_VIEW\\_TEXT\\_CHANGED](#)) event, even though that's not normally built into image views. The code to generate that event would look like this:

```
public void onTextChanged(String before, String after) {  
    ...  
    if (AccessibilityManager.getInstance(mContext).isEnabled()) {  
        sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED);  
    }  
    ...  
}
```

## Test Your Application

---

Be sure to test the accessibility functionality as you add it to your application. In order to test the content descriptions and Accessibility events, install and enable an accessibility service. One option is [Talkback](#) (<https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback>), a free, open source screen reader available on Google Play. With the service enabled, test all the navigation flows through your application and listen to the spoken feedback.

Also, attempt to navigate your application using a directional controller, instead of the touch screen. You can use a physical device with a d-pad or trackball if one is available. If not, use the Android emulator and its simulated keyboard controls.

Between the service providing feedback and the directional navigation through your application, you should get a sense of what your application is like to navigate without any visual cues. Fix problem areas as they appear, and you'll end up with a more accessible Android application.

# Developing an Accessibility Service

Accessibility services are a feature of the Android framework designed to provide alternative navigation feedback to the user on behalf of applications installed on Android devices. An accessibility service can communicate to the user on the application's behalf, such as converting text to speech, or haptic feedback when a user is hovering on an important area of the screen. This lesson covers how to create an accessibility service, process information received from the application, and report that information back to the user.

## Create Your Accessibility Service

An accessibility service can be bundled with a normal application, or created as a standalone Android project. The steps to creating the service are the same in either situation. Within your project, create a class that extends [AccessibilityService](#) ([/reference/android/accessibilityservice/AccessibilityService.html](#)).

```
package com.example.android.apis.accessibility;

import android.accessibilityservice.AccessibilityService;

public class MyAccessibilityService extends AccessibilityService {
    ...
    @Override
    public void onAccessibilityEvent(AccessibilityEvent event) {
    }

    @Override
    public void onInterrupt() {
    }

    ...
}
```

Like any other service, you also declare it in the manifest file. Remember to specify that it handles the `android.accessibilityservice` intent, so that the service is called when applications fire an [AccessibilityEvent](#) ([/reference/android/view/accessibility/AccessibilityEvent.html](#)).

```
<application ...>
    ...

```

### THIS LESSON TEACHES YOU TO

1. [Create Your Accessibility Service](#)
2. [Configure Your Accessibility Service](#)
3. [Respond to AccessibilityEvents](#)
4. [Query the View Hierarchy for More Context](#)

### YOU SHOULD ALSO READ

- [Building Accessibility Services](#)

```
<service android:name=".MyAccessibilityService">
    <intent-filter>
        <action android:name="android.accessibilityservice.AccessibilityService" />
    </intent-filter>
    ...
</service>
...
</application>
```

If you created a new project for this service, and don't plan on having an application, you can remove the starter Activity class (usually called `MainActivity.java`) from your source. Remember to also remove the corresponding activity element from your manifest.

## Configure Your Accessibility Service

---

Setting the configuration variables for your accessibility service tells the system how and when you want it to run. Which event types would you like to respond to? Should the service be active for all applications, or only specific package names? What different feedback types does it use?

You have two options for how to set these variables. The backwards-compatible option is to set them in code, using

[setServiceInfo\(android.accessibilityservice.AccessibilityServiceInfo\)](#) ([/reference/android/accessibilityservice](#)  
[/AccessibilityService.html#setServiceInfo\(android.accessibilityservice.AccessibilityServiceInfo\)](#)). To do that, override the [onServiceConnected\(\)](#) ([/reference/android/accessibilityservice](#)  
[/AccessibilityService.html#onServiceConnected\(\)](#)) method and configure your service in there.

```
@Override
public void onServiceConnected() {
    // Set the type of events that this service wants to listen to. Others
    // won't be passed to this service.
    info.eventTypes = AccessibilityEvent.TYPE_VIEW_CLICKED |
        AccessibilityEvent.TYPE_VIEW_FOCUSED;

    // If you only want this service to work with specific applications, set their
    // package names here. Otherwise, when the service is activated, it will list
    // to events from all applications.
    info.packageNames = new String[]
        {"com.example.android.myFirstApp", "com.example.android.mySecondApp"};

    // Set the type of feedback your service will provide.
    info.feedbackType = AccessibilityServiceInfo.FEEDBACK_SPOKEN;

    // Default services are invoked only if no package-specific ones are present
    // for the type of AccessibilityEvent generated. This service *is*
    // application-specific, so the flag isn't necessary. If this was a
    // general-purpose service, it would be worth considering setting the
```

```
// DEFAULT flag.  
  
// info.flags = AccessibilityServiceInfo.DEFAULT;  
  
info.notificationTimeout = 100;  
  
this.setServiceInfo(info);  
  
}
```

Starting with Android 4.0, there is a second option available: configure the service using an XML file. Certain configuration options like [canRetrieveWindowContent](#) ([/reference/android/R.attr.html#canRetrieveWindowContent](#)) are only available if you configure your service using XML. The same configuration options above, defined using XML, would look like this:

```
<accessibility-service  
    android:accessibilityEventTypes="typeViewClicked|typeViewFocused"  
    android:packageName="com.example.android.myFirstApp, com.example.android.my  
    android:accessibilityFeedbackType="feedbackSpoken"  
    android:notificationTimeout="100"  
    android:settingsActivity="com.example.android.apis.accessibility.TestBackAct  
    android:canRetrieveWindowContent="true"  
/>
```

If you go the XML route, be sure to reference it in your manifest, by adding a [<meta-data>](#) ([/guide/topics/manifest/meta-data-element.html](#)) tag to your service declaration, pointing at the XML file. If you stored your XML file in res/xml/serviceconfig.xml, the new tag would look like this:

```
<service android:name=".MyAccessibilityService">  
    <intent-filter>  
        <action android:name="android.accessibilityservice.AccessibilityService" />  
    </intent-filter>  
    <meta-data android:name="android.accessibilityservice"  
        android:resource="@xml/serviceconfig" />  
/</service>
```

## Respond to AccessibilityEvents

Now that your service is set up to run and listen for events, write some code so it knows what to do when an [AccessibilityEvent](#) ([/reference/android/view/accessibility/AccessibilityEvent.html](#)) actually arrives! Start by overriding the [onAccessibilityEvent\(AccessibilityEvent\)](#) ([/reference/android/accessibilityservice/AccessibilityService.html#onAccessibilityEvent\(android.view.accessibility.AccessibilityEvent\)](#)) method. In that method, use [getEventType\(\)](#) ([/reference/android/view/accessibility/AccessibilityEvent.html#getEventType\(\)](#)) to determine the type of event, and

[getContentDescription\(\)](#) ([/reference/android/view/accessibility/AccessibilityRecord.html#getContentDescription\(\)](#)) to extract any label text associated with the view that fired the event.

```
@Override  
public void onAccessibilityEvent(AccessibilityEvent event) {  
    final int eventType = event.getEventType();  
    String eventText = null;  
    switch(eventType) {  
        case AccessibilityEvent.TYPE_VIEW_CLICKED:  
            eventText = "Focused: ";  
            break;  
        case AccessibilityEvent.TYPE_VIEW_FOCUSED:  
            eventText = "Focused: ";  
            break;  
    }  
  
    eventText = eventText + event.getContentDescription();  
  
    // Do something nifty with this text, like speak the composed string  
    // back to the user.  
    speakToUser(eventText);  
    ...  
}
```

## Query the View Hierarchy for More Context

This step is optional, but highly useful. One of the new features in Android 4.0 (API Level 14) is the ability for an [AccessibilityService](#) ([/reference/android/accessibilityservice/AccessibilityService.html](#)) to query the view hierarchy, collecting information about the UI component that generated an event, and its parent and children. In order to do this, make sure that you set the following line in your XML configuration:

```
android:canRetrieveWindowContent="true"
```

Once that's done, get an [AccessibilityNodeInfo](#) ([/reference/android/view/accessibility/AccessibilityNodeInfo.html](#)) object using [getSource\(\)](#) ([/reference/android/view/accessibility/AccessibilityRecord.html#getSource\(\)](#)). This call only returns an object if the window where the event originated is still the active window. If not, it will return null, so *behave accordingly*. The following example is a snippet of code that, when it receives an event, does the following:

1. Immediately grab the parent of the view where the event originated
2. In that view, look for a label and a check box as children views
3. If it finds them, create a string to report to the user, indicating the label and whether it was checked or not.
4. If at any point a null value is returned while traversing the view hierarchy, the method quietly

gives up.

```
// Alternative onAccessibilityEvent, that uses AccessibilityNodeInfo

@Override
public void onAccessibilityEvent(AccessibilityEvent event) {

    AccessibilityNodeInfo source = event.getSource();
    if (source == null) {
        return;
    }

    // Grab the parent of the view that fired the event.
    AccessibilityNodeInfo rowNode = getListItemNodeInfo(source);
    if (rowNode == null) {
        return;
    }

    // Using this parent, get references to both child nodes, the label and the c
    AccessibilityNodeInfo labelNode = rowNode.getChild(0);
    if (labelNode == null) {
        rowNode.recycle();
        return;
    }

    AccessibilityNodeInfo completeNode = rowNode.getChild(1);
    if (completeNode == null) {
        rowNode.recycle();
        return;
    }

    // Determine what the task is and whether or not it's complete, based on
    // the text inside the label, and the state of the check-box.
    if (rowNode.getChildCount() < 2 || !rowNode.getChild(1).isCheckable()) {
        rowNode.recycle();
        return;
    }

    CharSequence taskLabel = labelNode.getText();
    final boolean isComplete = completeNode.isChecked();
    String completeStr = null;

    if (isComplete) {
        completeStr = getString(R.string.checked);
    } else {
        completeStr = getString(R.string.not_checked);
    }
    String reportStr = taskLabel + completeStr;
```

```
    speakToUser(reportStr);  
}
```

Now you have a complete, functioning accessibility service. Try configuring how it interacts with the user, by adding Android's [text-to-speech engine](http://android-developers.blogspot.com/2009/09/introduction-to-text-to-speech-in.html) (<http://android-developers.blogspot.com/2009/09/introduction-to-text-to-speech-in.html>), or using a [Vibrator](#) (</reference/android/os/Vibrator.html>) to provide haptic feedback!

# Displaying Graphics with OpenGL ES

The Android framework provides plenty of standard tools for creating attractive, functional graphical user interfaces. However, if you want more control of what your application draws on screen, or are venturing into three dimensional graphics, you need to use a different tool. The OpenGL ES APIs provided by the Android framework offers a set of tools for displaying high-end, animated graphics that are limited only by your imagination and can also benefit from the acceleration of graphics processing units (GPUs) provided on many Android devices.

This class walks you through the basics of developing applications that use OpenGL, including setup, drawing objects, moving drawn elements and responding to touch input.

The example code in this class uses the OpenGL ES 2.0 APIs, which is the recommended API version to use with current Android devices. For more information about versions of OpenGL ES, see the [OpenGL \(/guide/topics/graphics/opengl.html#choosing-version\)](#) developer guide.

**Note:** Be careful not to mix OpenGL ES 1.x API calls with OpenGL ES 2.0 methods! The two APIs are not interchangeable and trying to use them together only results in frustration and sadness.

## Lessons

---

### **Building an OpenGL ES Environment**

Learn how to set up an Android application to be able to draw OpenGL graphics.

### **Defining Shapes**

Learn how to define shapes and why you need to know about faces and winding.

### **Drawing Shapes**

Learn how to draw OpenGL shapes in your application.

### **Applying Projection and Camera Views**

Learn how to use projection and camera views to get a new perspective on your drawn objects.

### **Adding Motion**

Learn how to do basic movement and animation of drawn objects with OpenGL.

### **Responding to Touch Events**

Learn how to do basic interaction with OpenGL graphics.

### **DEPENDENCIES AND PREREQUISITES**

- Android 2.2 (API Level 8) or higher
- Experience building an [Android app](#)

### **YOU SHOULD ALSO READ**

- [OpenGL](#)

### **TRY IT OUT**

[Download the sample](#)

OpenGLES.zip

# Building an OpenGL ES Environment

In order to draw graphics with OpenGL ES in your Android application, you must create a view container for them. One of the more straight-forward ways to do this is to implement both a [GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#) and a [GLSurfaceView.Renderer \(/reference/android/opengl/GLSurfaceView.Renderer.html\)](#). A [GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#) is a view container for graphics drawn with OpenGL and [GLSurfaceView.Renderer \(/reference/android/opengl/GLSurfaceView.Renderer.html\)](#) controls what is drawn within that view. For more information about these classes, see the [OpenGL ES \(/guide/topics/graphics/opengl.html\)](#) developer guide.

[GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#) is just one way to incorporate OpenGL ES graphics into your application. For a full-screen or near-full screen graphics view, it is a reasonable choice. Developers who want to incorporate OpenGL ES graphics in a small portion of their layouts should take a look at [TextureView \(/reference/android/view/TextureView.html\)](#). For real, do-it-yourself developers, it is also possible to build up an OpenGL ES view using [SurfaceView \(/reference/android/view/SurfaceView.html\)](#), but this requires writing quite a bit of additional code.

This lesson explains how to complete a minimal implementation of [GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#) and [GLSurfaceView.Renderer \(/reference/android/opengl/GLSurfaceView.Renderer.html\)](#) in a simple application activity.

## Declare OpenGL ES Use in the Manifest

In order for your application to use the OpenGL ES 2.0 API, you must add the following declaration to your manifest:

```
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

If your application uses texture compression, you must also declare which compression formats you support so that devices that do not support these formats do not try to run your application:

```
<supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />
<supports-gl-texture android:name="GL_OES_compressed_paletted_texture" />
```

### THIS LESSON TEACHES YOU TO

1. [Declare OpenGL ES Use in the Manifest](#)
2. [Create an Activity for OpenGL ES Graphics](#)
3. [Build a GLSurfaceView Object](#)
4. [Build a Renderer Class](#)

### YOU SHOULD ALSO READ

- [OpenGL](#)

### TRY IT OUT

[Download the sample](#)

OpenGLES.zip

For more information about texture compression formats, see the [OpenGL \(/guide/topics/graphics/opengl.html#textures\)](#) developer guide.

## Create an Activity for OpenGL ES Graphics

Android applications that use OpenGL ES have activities just like any other application that has a user interface. The main difference from other applications is what you put in the layout for your activity. While in many applications you might use [TextView \(/reference/android/widget/TextView.html\)](#), [Button \(/reference/android/widget/Button.html\)](#) and [ListView \(/reference/android/widget/ListView.html\)](#), in an app that uses OpenGL ES, you can also add a [GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#).

The following code example shows a minimal implementation of an activity that uses a [GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#) as its primary view:

```
public class OpenGLES20 extends Activity {  
  
    private GLSurfaceView mGLView;  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        // Create a GLSurfaceView instance and set it  
        // as the ContentView for this Activity.  
        mGLView = new MyGLSurfaceView(this);  
        setContentView(mGLView);  
    }  
}
```

Note: OpenGL ES 2.0 requires Android 2.2 (API Level 8) or higher, so make sure your Android project targets that API or higher.

## Build a GLSurfaceView Object

A [GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#) is a specialized view where you can draw OpenGL ES graphics. It does not do much by itself. The actual drawing of objects is controlled in the [GLSurfaceView.Renderer \(/reference/android/opengl/GLSurfaceView.Renderer.html\)](#) that you set on this view. In fact, the code for this object is so thin, you may be tempted to skip extending it and just create an unmodified [GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#) instance, but don't do that. You need to extend this class in order to capture touch events, which is covered in the [Responding to Touch Events \(#touch.html\)](#) lesson.

The essential code for a [GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#) is minimal, so for a quick implementation, it is common to just create an inner class in the activity that uses it:

```

class MyGLSurfaceView extends GLSurfaceView {

    public MyGLSurfaceView(Context context) {
        super(context);

        // Set the Renderer for drawing on the GLSurfaceView
        setRenderer(new MyRenderer());
    }
}

```

When using OpenGL ES 2.0, you must add another call to your [GLSurfaceView](#) ([/reference/android/opengl/GLSurfaceView.html](#)) constructor, specifying that you want to use the 2.0 API:

```

// Create an OpenGL ES 2.0 context
setEGLContextClientVersion(2);

```

Note: If you are using the OpenGL ES 2.0 API, make sure you declare this in your application manifest. For more information, see [Declare OpenGL ES Use in the Manifest](#) (#manifest).

One other optional addition to your [GLSurfaceView](#) ([/reference/android/opengl/GLSurfaceView.html](#)) implementation is to set the render mode to only draw the view when there is a change to your drawing data using the [GLSurfaceView.RENDERMODE\\_WHEN\\_DIRTY](#) ([/reference/android/opengl/GLSurfaceView.html#RENDERMODE\\_WHEN\\_DIRTY](#)) setting:

```

// Render the view only when there is a change in the drawing data
setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);

```

This setting prevents the [GLSurfaceView](#) ([/reference/android/opengl/GLSurfaceView.html](#)) frame from being redrawn until you call [requestRender\(\)](#) ([/reference/android/opengl/GLSurfaceView.html#requestRender\(\)](#)), which is more efficient for this sample app.

## Build a Renderer Class

The implementation of the [GLSurfaceView.Renderer](#) ([/reference/android/opengl/GLSurfaceView.Renderer.html](#)) class, or renderer, within an application that uses OpenGL ES is where things start to get interesting. This class controls what gets drawn on the [GLSurfaceView](#) ([/reference/android/opengl/GLSurfaceView.html](#)) with which it is associated. There are three methods in a renderer that are called by the Android system in order to figure out what and how to draw on a [GLSurfaceView](#) ([/reference/android/opengl/GLSurfaceView.html](#)):

- [onSurfaceCreated\(\)](#) - Called once to set up the view's OpenGL ES environment.
- [onDrawFrame\(\)](#) - Called for each redraw of the view.
- [onSurfaceChanged\(\)](#) - Called if the geometry of the view changes, for example when the device's screen orientation changes.

Here is a very basic implementation of an OpenGL ES renderer, that does nothing more than draw a gray background in the [GLSurfaceView](#) ([/reference/android/opengl/GLSurfaceView.html](#)):

```
public class MyGL20Renderer implements GLSurfaceView.Renderer {  
  
    public void onSurfaceCreated(GL10 unused, EGLConfig config) {  
        // Set the background frame color  
        GLES20.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);  
    }  
  
    public void onDrawFrame(GL10 unused) {  
        // Redraw background color  
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);  
    }  
  
    public void onSurfaceChanged(GL10 unused, int width, int height) {  
        GLES20.glViewport(0, 0, width, height);  
    }  
}
```

That's all there is to it! The code examples above create a simple Android application that displays a gray screen using OpenGL. While this code does not do anything very interesting, by creating these classes, you have laid the foundation you need to start drawing graphic elements with OpenGL.

Note: You may wonder why these methods have a [GL10](#) ([/reference/javax/microedition/khronos/opengles/GL10.html](#)) parameter, when you are using the OpenGL ES 2.0 APIs. These method signatures are simply reused for the 2.0 APIs to keep the Android framework code simpler.

If you are familiar with the OpenGL ES APIs, you should now be able to set up a OpenGL ES environment in your app and start drawing graphics. However, if you need a bit more help getting started with OpenGL, head on to the next lessons for a few more hints.

# Defining Shapes

Being able to define shapes to be drawn in the context of an OpenGL ES view is the first step in creating your high-end graphics masterpiece. Drawing with OpenGL ES can be a little tricky without knowing a few basic things about how OpenGL ES expects you to define graphic objects.

This lesson explains the OpenGL ES coordinate system relative to an Android device screen, the basics of defining a shape, shape faces, as well as defining a triangle and a square.

## THIS LESSON TEACHES YOU TO

1. [Define a Triangle](#)
2. [Define a Square](#)

## YOU SHOULD ALSO READ

- [OpenGL](#)

[Download the sample](#)

OpenGLES.zip

## Define a Triangle

OpenGL ES allows you to define drawn objects using coordinates in three-dimensional space. So, before you can draw a triangle, you must define its coordinates. In OpenGL, the typical way to do this is to define a vertex array of floating point numbers for the coordinates. For maximum efficiency, you write these coordinates into a [ByteBuffer](#) (</reference/java/nio/ByteBuffer.html>), that is passed into the OpenGL ES graphics pipeline for processing.

```
class Triangle {  
  
    private FloatBuffer vertexBuffer;  
  
    // number of coordinates per vertex in this array  
    static final int COORDS_PER_VERTEX = 3;  
    static float triangleCoords[] = { // in counterclockwise order:  
        0.0f,  0.622008459f, 0.0f,      // top  
       -0.5f, -0.311004243f, 0.0f,      // bottom left  
        0.5f, -0.311004243f, 0.0f      // bottom right  
    };  
  
    // Set color with red, green, blue and alpha (opacity) values  
    float color[] = { 0.63671875f, 0.76953125f, 0.22265625f, 1.0f };  
  
    public Triangle() {  
        // initialize vertex byte buffer for shape coordinates  
        ByteBuffer bb = ByteBuffer.allocateDirect(  
            // (number of coordinate values * 4 bytes per float)  
            triangleCoords.length * 4);  
        // use the device hardware's native byte order  
        bb.order(ByteOrder.nativeOrder());  
    }  
}
```

```
// create a floating point buffer from the ByteBuffer  
vertexBuffer = bb.asFloatBuffer();  
// add the coordinates to the FloatBuffer  
vertexBuffer.put(triangleCoords);  
// set the buffer to read the first coordinate  
vertexBuffer.position(0);  
}  
}
```

By default, OpenGL ES assumes a coordinate system where [0,0,0] (X,Y,Z) specifies the center of the [GLSurfaceView](#) ([/reference/android/opengl/GLSurfaceView.html](#)) frame, [1,1,0] is the top right corner of the frame and [-1,-1,0] is bottom left corner of the frame. For an illustration of this coordinate system, see the [OpenGL ES](#) ([/guide/topics/graphics/opengl.html#coordinate-mapping](#)) developer guide.

Note that the coordinates of this shape are defined in a counterclockwise order. The drawing order is important because it defines which side is the front face of the shape, which you typically want to have drawn, and the back face, which you can choose to not draw using the OpenGL ES cull face feature. For more information about faces and culling, see the [OpenGL ES](#) ([/guide/topics/graphics/opengl.html#faces-winding](#)) developer guide.

## Define a Square

Defining triangles is pretty easy in OpenGL, but what if you want to get a just a little more complex? Say, a square? There are a number of ways to do this, but a typical path to drawing such a shape in OpenGL ES is to use two triangles drawn together:

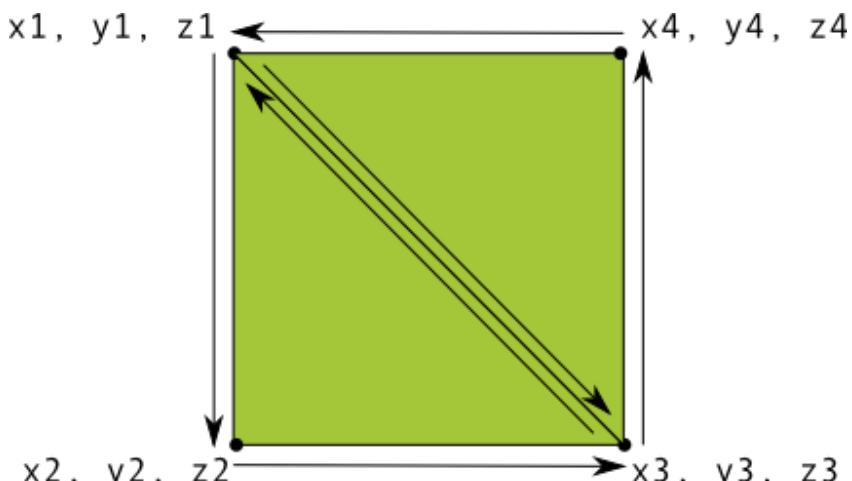


Figure 1. Drawing a square using two triangles.

Again, you should define the vertices in a counterclockwise order for both triangles that represent this shape, and put the values in a [ByteBuffer](#) ([/reference/java/nio/ByteBuffer.html](#)). In order to avoid defining the two coordinates shared by each triangle twice, use a drawing list to tell the OpenGL ES graphics pipeline how to draw these vertices. Here's the code for this shape:

```
class Square {
```

```
private FloatBuffer vertexBuffer;
private ShortBuffer drawListBuffer;

// number of coordinates per vertex in this array
static final int COORDS_PER_VERTEX = 3;
static float squareCoords[] = { -0.5f,  0.5f, 0.0f,    // top left
                               -0.5f, -0.5f, 0.0f,    // bottom left
                                0.5f, -0.5f, 0.0f,    // bottom right
                                0.5f,  0.5f, 0.0f }; // top right

private short drawOrder[] = { 0, 1, 2, 0, 2, 3 }; // order to draw vertices

public Square() {
    // initialize vertex byte buffer for shape coordinates
    ByteBuffer bb = ByteBuffer.allocateDirect(
        // (# of coordinate values * 4 bytes per float)
        squareCoords.length * 4);
    bb.order(ByteOrder.nativeOrder());
    vertexBuffer = bb.asFloatBuffer();
    vertexBuffer.put(squareCoords);
    vertexBuffer.position(0);

    // initialize byte buffer for the draw list
    ByteBuffer dbl = ByteBuffer.allocateDirect(
        // (# of coordinate values * 2 bytes per short)
        drawOrder.length * 2);
    dbl.order(ByteOrder.nativeOrder());
    drawListBuffer = dbl.asShortBuffer();
    drawListBuffer.put(drawOrder);
    drawListBuffer.position(0);
}
}
```

This example gives you a peek at what it takes to create more complex shapes with OpenGL. In general, you use collections of triangles to draw objects. In the next lesson, you learn how to draw these shapes on screen.

# Drawing Shapes

After you define shapes to be drawn with OpenGL, you probably want to draw them. Drawing shapes with the OpenGL ES 2.0 takes a bit more code than you might imagine, because the API provides a great deal of control over the graphics rendering pipeline.

This lesson explains how to draw the shapes you defined in the previous lesson using the OpenGL ES 2.0 API.

## Initialize Shapes

Before you do any drawing, you must initialize and load the shapes you plan to draw. Unless the structure (the original coordinates) of the shapes you use in your program change during the course of execution, you should initialize them in the `onSurfaceCreated()` ([/reference/android/opengl/GLSurfaceView.Renderer.html#onSurfaceCreated\(javax.microedition.khronos.opengles.GL10, javax.microedition.khronos.egl.EGLConfig\)](#)) method of your renderer for memory and processing efficiency.

```
public void onSurfaceCreated(GL10 unused, EGLConfig config) {  
    ...  
  
    // initialize a triangle  
    mTriangle = new Triangle();  
    // initialize a square  
    mSquare = new Square();  
}
```

### THIS LESSON TEACHES YOU TO

1. [Initialize Shapes](#)
2. [Draw a Shape](#)

### YOU SHOULD ALSO READ

- [OpenGL](#)

[Download the sample](#)

OpenGLES.zip

## Draw a Shape

Drawing a defined shape using OpenGL ES 2.0 requires a significant amount of code, because you must provide a lot of details to the graphics rendering pipeline. Specifically, you must define the following:

- *Vertex Shader* - OpenGL ES graphics code for rendering the vertices of a shape.
- *Fragment Shader* - OpenGL ES code for rendering the face of a shape with colors or textures.
- *Program* - An OpenGL ES object that contains the shaders you want to use for drawing one or more shapes.

You need at least one vertex shader to draw a shape and one fragment shader to color that shape. These shaders must be compiled and then added to an OpenGL ES program, which is then used to

draw the shape. Here is an example of how to define basic shaders you can use to draw a shape:

```
private final String vertexShaderCode =  
    "attribute vec4 vPosition;" +  
    "void main() {" +  
    "    gl_Position = vPosition;" +  
    "}";  
  
private final String fragmentShaderCode =  
    "precision mediump float;" +  
    "uniform vec4 vColor;" +  
    "void main() {" +  
    "    gl_FragColor = vColor;" +  
    "}";
```

Shaders contain OpenGL Shading Language (GLSL) code that must be compiled prior to using it in the OpenGL ES environment. To compile this code, create a utility method in your renderer class:

```
public static int loadShader(int type, String shaderCode){  
  
    // create a vertex shader type (GLES20.GL_VERTEX_SHADER)  
    // or a fragment shader type (GLES20.GL_FRAGMENT_SHADER)  
    int shader = GLES20.glCreateShader(type);  
  
    // add the source code to the shader and compile it  
    GLES20.glShaderSource(shader, shaderCode);  
    GLES20.glCompileShader(shader);  
  
    return shader;  
}
```

In order to draw your shape, you must compile the shader code, add them to a OpenGL ES program object and then link the program. Do this in your drawn object's constructor, so it is only done once.

Note: Compiling OpenGL ES shaders and linking programs is expensive in terms of CPU cycles and processing time, so you should avoid doing this more than once. If you do not know the content of your shaders at runtime, you should build your code such that they only get created once and then cached for later use.

```
public Triangle() {  
    ...  
  
    int vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vertexShaderCode);  
    int fragmentShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fragmentShaderCode)  
  
    mProgram = GLES20.glCreateProgram();  
    // create empty OpenGL ES Program
```

```
GLES20.glAttachShader(mProgram, vertexShader); // add the vertex shader to  
GLES20.glAttachShader(mProgram, fragmentShader); // add the fragment shader to  
GLES20.glLinkProgram(mProgram); // creates OpenGL ES program  
}
```

At this point, you are ready to add the actual calls that draw your shape. Drawing shapes with OpenGL ES requires that you specify several parameters to tell the rendering pipeline what you want to draw and how to draw it. Since drawing options can vary by shape, it's a good idea to have your shape classes contain their own drawing logic.

Create a draw() method for drawing the shape. This code sets the position and color values to the shape's vertex shader and fragment shader, and then executes the drawing function.

```
public void draw() {  
    // Add program to OpenGL ES environment  
    GLES20.glUseProgram(mProgram);  
  
    // get handle to vertex shader's vPosition member  
    mPositionHandle = GLES20.glGetAttribLocation(mProgram, "vPosition");  
  
    // Enable a handle to the triangle vertices  
    GLES20.glEnableVertexAttribArray(mPositionHandle);  
  
    // Prepare the triangle coordinate data  
    GLES20.glVertexAttribPointer(mPositionHandle, COORDS_PER_VERTEX,  
                                 GLES20.GL_FLOAT, false,  
                                 vertexStride, vertexBuffer);  
  
    // get handle to fragment shader's vColor member  
    mColorHandle = GLES20.glGetUniformLocation(mProgram, "vColor");  
  
    // Set color for drawing the triangle  
    GLES20 glUniform4fv(mColorHandle, 1, color, 0);  
  
    // Draw the triangle  
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);  
  
    // Disable vertex array  
    GLES20.glDisableVertexAttribArray(mPositionHandle);  
}
```

Once you have all this code in place, drawing this object just requires a call to the draw() method from within your renderer's [onDrawFrame\(\)](#) ([/reference/android/opengl/GLSurfaceView.Renderer.html#onDrawFrame\(javafx.microedition.khronos.opengles.GL10\)](#)) method. When you run the application, it should look something like this:

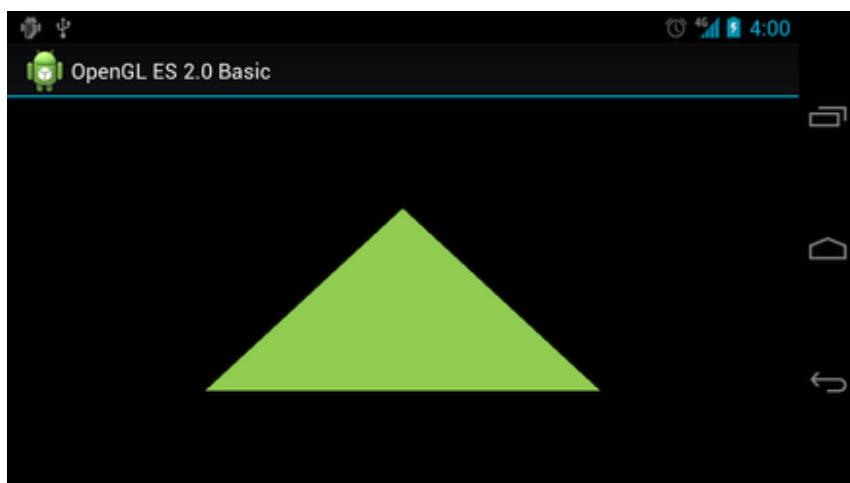


Figure 1. Triangle drawn without a projection or camera view.

There are a few problems with this code example. First of all, it is not going to impress your friends. Secondly, the triangle is a bit squashed and changes shape when you change the screen orientation of the device. The reason the shape is skewed is due to the fact that the object's vertices have not been corrected for the proportions of the screen area where the [GLSurfaceView](#) ([/reference/android/opengl/GLSurfaceView.html](#)) is displayed. You can fix that problem using a projection and camera view in the next lesson.

Lastly, the triangle is stationary, which is a bit boring. In the [Adding Motion \(motion.html\)](#) lesson, you make this shape rotate and make more interesting use of the OpenGL ES graphics pipeline.

# Applying Projection and Camera Views

In the OpenGL ES environment, projection and camera views allow you to display drawn objects in a way that more closely resembles how you see physical objects with your eyes. This simulation of physical viewing is done with mathematical transformations of drawn object coordinates:

- **Projection** - This transformation adjusts the coordinates of drawn objects based on the width and height of the `GLSurfaceView` where they are displayed. Without this calculation, objects drawn by OpenGL ES are skewed by the unequal proportions of the view window. A projection transformation typically only has to be calculated when the proportions of the OpenGL view are established or changed in the `onSurfaceChanged()` method of your renderer. For more information about OpenGL ES projections and coordinate mapping, see [Mapping Coordinates for Drawn Objects](#).
- **Camera View** - This transformation adjusts the coordinates of drawn objects based on a virtual camera position. It's important to note that OpenGL ES does not define an actual camera object, but instead provides utility methods that simulate a camera by transforming the display of drawn objects. A camera view transformation might be calculated only once when you establish your `GLSurfaceView`, or might change dynamically based on user actions or your application's function.

This lesson describes how to create a projection and camera view and apply it to shapes drawn in your `GLSurfaceView` ([/reference/android/opengl/GLSurfaceView.html](#)).

## Define a Projection

The data for a projection transformation is calculated in the `onSurfaceChanged()` ([/reference/android/opengl/GLSurfaceView.Renderer.html#onSurfaceChanged\(javax.microedition.khronos.opengles.GL10, int, int\)](#)) method of your `GLSurfaceView.Renderer` ([/reference/android/opengl/GLSurfaceView.Renderer.html](#)) class. The following example code takes the height and width of the `GLSurfaceView` ([/reference/android/opengl/GLSurfaceView.html](#)) and uses it to populate a projection transformation `Matrix` ([/reference/android/opengl/Matrix.html](#)) using the `Matrix.frustumM()` ([/reference/android/opengl/Matrix.html#frustumM\(float\[\], int, float, float, float, float, float\)](#)) method:

```
@Override
public void onSurfaceChanged(GL10 unused, int width, int height) {
    GLES20.glViewport(0, 0, width, height);

    float ratio = (float) width / height;
```

### THIS LESSON TEACHES YOU TO

1. [Define a Projection](#)
2. [Define a Camera View](#)
3. [Apply Projection and Camera Transformations](#)

### YOU SHOULD ALSO READ

- [OpenGL](#)

[Download the sample](#)

OpenGLES.zip

```
// this projection matrix is applied to object coordinates
// in the onDrawFrame() method
Matrix.frustumM(mProjMatrix, 0, -ratio, ratio, -1, 1, 3, 7);
}
```

This code populates a projection matrix, `mProjMatrix` which you can then combine with a camera view transformation in the [onDrawFrame\(\)](#) (`/reference/android/opengl/GLSurfaceView.Renderer.html#onDrawFrame(javax.microedition.khronos.opengles.GL10)`) method, which is shown in the next section.

Note: Just applying a projection transformation to your drawing objects typically results in a very empty display. In general, you must also apply a camera view transformation in order for anything to show up on screen.

## Define a Camera View

Complete the process of transforming your drawn objects by adding a camera view transformation as part of the drawing process. In the following example code, the camera view transformation is calculated using the [Matrix.setLookAtM\(\)](#) (`/reference/android/opengl/Matrix.html#setLookAtM(float[], int, float, float, float, float, float, float, float, float, float, float)`) method and then combined with the previously calculated projection matrix. The combined transformation matrices are then passed to the drawn shape.

```
@Override
public void onDrawFrame(GL10 unused) {
    ...
    // Set the camera position (View matrix)
    Matrix.setLookAtM(mVMatrix, 0, 0, 0, -3, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

    // Calculate the projection and view transformation
    Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mVMatrix, 0);

    // Draw shape
    mTriangle.draw(mMVPMatrix);
}
```

## Apply Projection and Camera Transformations

In order to use the combined projection and camera view transformation matrix shown in the previous sections, modify the `draw()` method of your graphic objects to accept the combined transformation matrix and apply it to the shape:

```
public void draw(float[] mvpMatrix) { // pass in the calculated transformation ma
```

```
...
// get handle to shape's transformation matrix
mMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");

// Apply the projection and view transformation
GLES20.glUniformMatrix4fv(mMVPMatrixHandle, 1, false, mvpMatrix, 0);

// Draw the triangle
GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);
...
}
```

Once you have correctly calculated and applied the projection and camera view transformations, your graphic objects are drawn in correct proportions and should look like this:

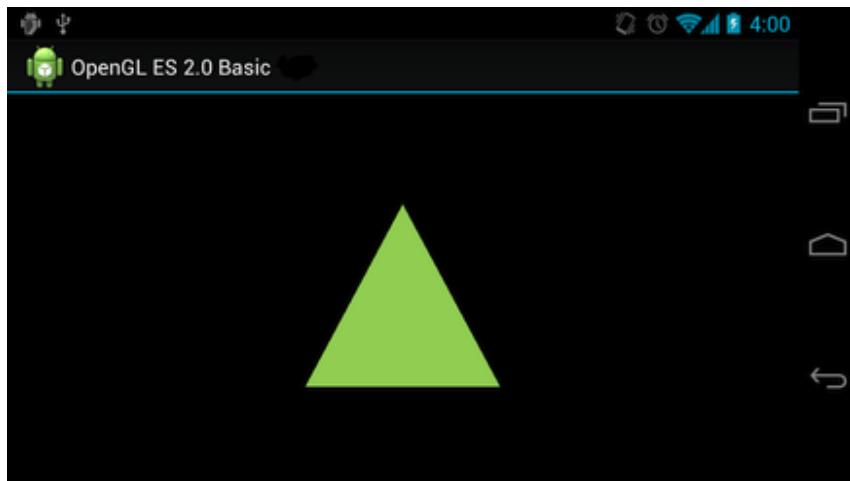


Figure 1. Triangle drawn with a projection and camera view applied.

Now that you have an application that displays your shapes in correct proportions, it's time to add motion to your shapes.

# Adding Motion

Drawing objects on screen is a pretty basic feature of OpenGL, but you can do this with other Android graphics framework classes, including [Canvas](#) (/reference/android/graphics/Canvas.html) and [Drawable](#) (/reference/android/graphics/drawable/Drawable.html) objects. OpenGL ES provides additional capabilities for moving and transforming drawn objects in three dimensions or in other unique ways to create compelling user experiences.

In this lesson, you take another step forward into using OpenGL ES by learning how to add motion to a shape with rotation.

## THIS LESSON TEACHES YOU TO

1. [Rotate a Shape](#)
2. [Enable Continuous Rendering](#)

## YOU SHOULD ALSO READ

- [OpenGL](#)

[Download the sample](#)

OpenGL.zip

## Rotate a Shape

Rotating a drawing object with OpenGL ES 2.0 is relatively simple. You create another transformation matrix (a rotation matrix) and then combine it with your projection and camera view transformation matrices:

```
private float[] mRotationMatrix = new float[16];
public void onDrawFrame(GL10 gl) {
    ...
    // Create a rotation transformation for the triangle
    long time = SystemClock.uptimeMillis() % 4000L;
    float angle = 0.090f * ((int) time);
    Matrix.setRotateM(mRotationMatrix, 0, mAngle, 0, 0, -1.0f);

    // Combine the rotation matrix with the projection and camera view
    Matrix.multiplyMM(mMVPMatrix, 0, mRotationMatrix, 0, mMVPMatrix, 0);

    // Draw triangle
    mTriangle.draw(mMVPMatrix);
}
```

If your triangle does not rotate after making these changes, make sure you have commented out the [GLSurfaceView.RENDERMODE\\_WHEN\\_DIRTY](#) (/reference/android/opengl/GLSurfaceView.html#RENDERMODE\_WHEN\_DIRTY) setting, as described in the next section.

## Enable Continuous Rendering

If you have diligently followed along with the example code in this class to this point, make sure

you comment out the line that sets the render mode only draw when dirty, otherwise OpenGL rotates the shape only one increment and then waits for a call to [requestRender\(\)](#) ([/reference/android/opengl/GLSurfaceView.html#requestRender\(\)](#)) from the [GLSurfaceView](#) ([/reference/android/opengl/GLSurfaceView.html](#)) container:

```
public MyGLSurfaceView(Context context) {  
    ...  
    // Render the view only when there is a change in the drawing data  
    //setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY); // comment out for auto  
}
```

Unless you have objects changing without any user interaction, it's usually a good idea have this flag turned on. Be ready to uncomment this code, because the next lesson makes this call applicable once again.

# Responding to Touch Events

Making objects move according to a preset program like the rotating triangle is useful for getting some attention, but what if you want to have users interact with your OpenGL ES graphics? The key to making your OpenGL ES application touch interactive is expanding your implementation of [GLSurfaceView \(/reference/android.opengl/GLSurfaceView.html\)](#) to override the [onTouchEvent \(\) \(/reference/android/view/View.html#onTouchEvent\(android.view.MotionEvent\)\)](#) to listen for touch events.

This lesson shows you how to listen for touch events to let users rotate an OpenGL ES object.

## THIS LESSON TEACHES YOU TO

1. [Setup a Touch Listener](#)
2. [Expose the Rotation Angle](#)
3. [Apply Rotation](#)

## YOU SHOULD ALSO READ

- [OpenGL](#)

[Download the sample](#)

OpenGLES.zip

## Setup a Touch Listener

In order to make your OpenGL ES application respond to touch events, you must implement the [onTouchEvent \(\) \(/reference/android/view/View.html#onTouchEvent\(android.view.MotionEvent\)\)](#) method in your [GLSurfaceView \(/reference/android/opengl/GLSurfaceView.html\)](#) class. The example implementation below shows how to listen for [MotionEvent.ACTION\\_MOVE \(/reference/android/view/MotionEvent.html#ACTION\\_MOVE\)](#) events and translate them to an angle of rotation for a shape.

```
@Override  
public boolean onTouchEvent(MotionEvent e) {  
    // MotionEvent reports input details from the touch screen  
    // and other input controls. In this case, you are only  
    // interested in events where the touch position changed.  
  
    float x = e.getX();  
    float y = e.getY();  
  
    switch (e.getAction()) {  
        case MotionEvent.ACTION_MOVE:  
  
            float dx = x - mPreviousX;  
            float dy = y - mPreviousY;  
  
            // reverse direction of rotation above the mid-line  
            if (y > getHeight() / 2) {  
                dx = dx * -1 ;  
            }  
  
            // reverse direction of rotation to left of the mid-line
```

```
        if (x < getWidth() / 2) {
            dy = dy * -1 ;
        }

        mRenderer.mAngle += (dx + dy) * TOUCH_SCALE_FACTOR; // = 180.0f / 32
        requestRender();
    }

    mPreviousX = x;
    mPreviousY = y;
    return true;
}
```

Notice that after calculating the rotation angle, this method calls [requestRender\(\)](#) to tell the renderer that it is time to render the frame. This approach is the most efficient in this example because the frame does not need to be redrawn unless there is a change in the rotation. However, it does not have any impact on efficiency unless you also request that the renderer only redraw when the data changes using the [setRenderMode\(\)](#) method, so make sure this line is uncommented in the renderer:

```
public MyGLSurfaceView(Context context) {
    ...
    // Render the view only when there is a change in the drawing data
    setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
}
```

## Expose the Rotation Angle

The example code above requires that you expose the rotation angle through your renderer by adding a public member. Since the renderer code is running on a separate thread from the main user interface thread of your application, you must declare this public variable as `volatile`. Here is the code to do that:

```
public class MyGLRenderer implements GLSurfaceView.Renderer {
    ...
    public volatile float mAngle;
```

## Apply Rotation

To apply the rotation generated by touch input, comment out the code that generates an angle and add `mAngle`, which contains the touch input generated angle:

```
public void onDrawFrame(GL10 gl) {  
    ...  
    // Create a rotation for the triangle  
    // long time = SystemClock.uptimeMillis() % 4000L;  
    // float angle = 0.090f * ((int) time);  
    Matrix.setRotateM(mRotationMatrix, 0, mAngle, 0, 0, -1.0f);  
  
    // Combine the rotation matrix with the projection and camera view  
    Matrix.multiplyMM(mMVPMatrix, 0, mRotationMatrix, 0, mMVPMatrix, 0);  
  
    // Draw triangle  
    mTriangle.draw(mMVPMatrix);  
}
```

When you have completed the steps described above, run the program and drag your finger over the screen to rotate the triangle:

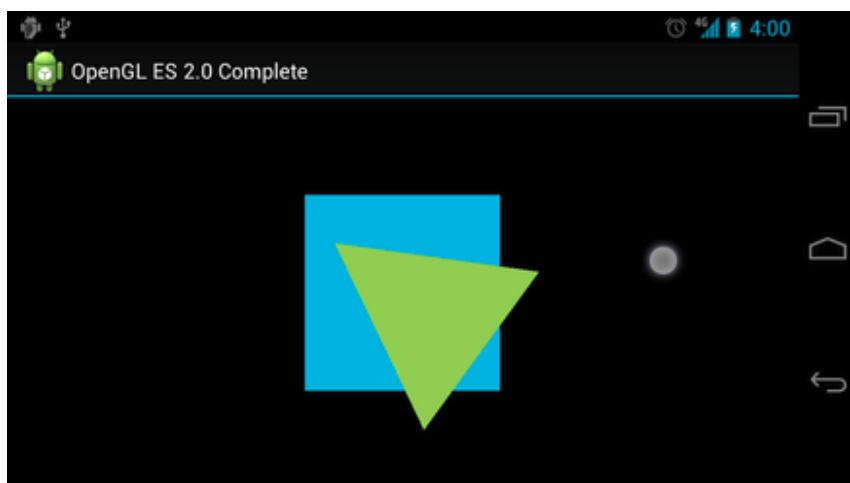


Figure 1. Triangle being rotated with touch input (circle shows touch location).

# Connecting Devices Wirelessly

Sharing Content

Besides enabling communication between devices on the same network, Android's wireless APIs also let you connect to other devices on the same network or to nearby devices which are not on the same network. The addition of Network Service Discovery (NSD) takes this further by letting your application seek out a nearby device that it can communicate with. Integrating NSD with your application helps you add new features, such as playing content from another device in the room, pulling images from a nearby webcam, or remotely logging into a device.

This class describes the key concepts involved in NSD. Specifically, it describes the process for doing peer-to-peer wireless connections using Wi-Fi Direct in combination to discover nearby devices. Neither device is connected to the same network.

## Lessons

### Using Network Service Discovery

Learn how to broadcast services on the local network, and discover services to connect to.

### Connecting with Wi-Fi Direct

Learn how to fetch a list of nearby peer devices, create an access point for legacy devices, and connect to other devices capable of Wi-Fi Direct connections.

### Using Wi-Fi Direct for Service Discovery

Learn how to discover services published by nearby devices without being on the same network, using Wi-Fi Direct.

Capturing Photos

Maintaining Multiple APKs

Creating Backward-Compatible UIs

Developing for Enterprise

Monetizing Your App

Designing Effective Navigation

Implementing Effective Navigation

Designing for TV

Displaying Bitmaps Efficiently

Implementing Accessibility

Displaying Graphics with OpenGL ES

[Connecting Devices Wirelessly](#)

Using Network Service Discovery

Connecting with Wi-Fi Direct

Using Wi-Fi Direct for Service Discovery

## DEPENDENCIES AND PREREQUISITES

- Android 4.1 or higher

## YOU SHOULD ALSO READ

- [Wi-Fi Direct](#)

same network.

ing to other devices from your application. You can use the Network Service Discovery API and the Wi-Fi Direct™ API to do this. This guide shows you how to use NSD and Wi-Fi Direct to find a nearby device and connect to the device when they are both on the same network.

In this guide, you learn how to publish services from your application, discover services offered on the network, and get connection details for the service you wish to connect to.

# Using Network Service Discovery

Adding Network Service Discovery (NSD) to your app allows your users to identify other devices on the local network that support the services your app requests. This is useful for a variety of peer-to-peer applications such as file sharing or multi-player gaming. Android's NSD APIs simplify the effort required for you to implement such features.

This lesson shows you how to build an application that can broadcast its name and connection information to the local network and scan for information from other applications doing the same. Finally, this lesson shows you how to connect to the same application running on another device.

## Register Your Service on the Network

Note: This step is optional. If you don't care about broadcasting your app's services over the local network, you can skip forward to the next section, [Discover Services on the Network \(#discover\)](#).

To register your service on the local network, first create a [NsdServiceInfo \(/reference/android/net/nsd/NsdServiceInfo.html\)](#) object. This object provides the information that other devices on the network use when they're deciding whether to connect to your service.

```
public void registerService(int port) {  
    // Create the NsdServiceInfo object, and populate it.  
    NsdServiceInfo serviceInfo = new NsdServiceInfo();  
  
    // The name is subject to change based on conflicts  
    // with other services advertised on the same network.  
    serviceInfo.setServiceName("NsdChat");  
    serviceInfo.setServiceType("_http._tcp.");  
    serviceInfo.setPort(port);  
    ....  
}
```

This code snippet sets the service name to "NsdChat". The name is visible to any device on the network that is using NSD to look for local services. Keep in mind that the name must be unique for any service on the network, and Android automatically handles conflict resolution. If two devices on the network both have the NsdChat application installed, one of them changes the service name automatically, to something like "NsdChat (1)".

The second parameter sets the service type, specifies which protocol and transport layer the

### THIS LESSON TEACHES YOU HOW TO

1. [Register Your Service on the Network](#)
2. [Discover Services on the Network](#)
3. [Connect to Services on the Network](#)
4. [Unregister Your Service on Application Close](#)

### TRY IT OUT

[Download the sample app](#)

nsdchat.zip

application uses. The syntax is "`_<protocol>._<transportlayer>`". In the code snippet, the service uses HTTP protocol running over TCP. An application offering a printer service (for instance, a network printer) would set the service type to "`_ipp._tcp`".

Note: The International Assigned Numbers Authority (IANA) manages a centralized, authoritative list of service types used by service discovery protocols such as NSD and Bonjour. You can download the list from [the IANA list of service names and port numbers \(http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml\)](http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml). If you intend to use a new service type, you should reserve it by filling out the [IANA Ports and Service registration form \(http://www.iana.org/form/ports-services\)](http://www.iana.org/form/ports-services).

When setting the port for your service, avoid hardcoding it as this conflicts with other applications. For instance, assuming that your application always uses port 1337 puts it in potential conflict with other installed applications that use the same port. Instead, use the device's next available port. Because this information is provided to other apps by a service broadcast, there's no need for the port your application uses to be known by other applications at compile-time. Instead, the applications can get this information from your service broadcast, right before connecting to your service.

If you're working with sockets, here's how you can initialize a socket to any available port simply by setting it to 0.

```
public void initializeServerSocket() {
    // Initialize a server socket on the next available port.
    mServerSocket = new ServerSocket(0);

    // Store the chosen port.
    mLocalPort = mServerSocket.getLocalPort();
    ...
}
```

Now that you've defined the [NsdServiceInfo](#) ([/reference/android/net/nsd/NsdServiceInfo.html](#)) object, you need to implement the [RegistrationListener](#) ([/reference/android/net/nsd/NsdManager.RegistrationListener.html](#)) interface. This interface contains callbacks used by Android to alert your application of the success or failure of service registration and unregistration.

```
public void initializeRegistrationListener() {
    mRegistrationListener = new NsdManager.RegistrationListener() {

        @Override
        public void onServiceRegistered(NsdServiceInfo nsdServiceInfo) {
            // Save the service name. Android may have changed it in order to
            // resolve a conflict, so update the name you initially requested
            // with the name Android actually used.
            mServiceName = nsdServiceInfo.getServiceName();
        }

        @Override
        ...
    }
}
```

```
public void onRegistrationFailed(NsdServiceInfo serviceInfo, int errorCode
        // Registration failed! Put debugging code here to determine why.
    }

@Override
public void onServiceUnregistered(NsdServiceInfo arg0) {
    // Service has been unregistered. This only happens when you call
    // NsdManager.unregisterService() and pass in this listener.
}

@Override
public void onUnregistrationFailed(NsdServiceInfo serviceInfo, int errorCode
        // Unregistration failed. Put debugging code here to determine why.
    }
};

}
```

Now you have all the pieces to register your service. Call the method [registerService\(\)](#) ([/reference/android/net/nsd/NsdManager.html#registerService\(android.net.nsd.NsdServiceInfo, int, android.net.nsd.NsdManager.RegistrationListener\)](#)).

Note that this method is asynchronous, so any code that needs to run after the service has been registered must go in the [onServiceRegistered\(\)](#) ([/reference/android/net/nsd/NsdManager.RegistrationListener.html#onServiceRegistered\(android.net.nsd.NsdServiceInfo\)](#)) method.

```
public void registerService(int port) {
    NsdServiceInfo serviceInfo = new NsdServiceInfo();
    serviceInfo.setServiceName("NsdChat");
    serviceInfo.setServiceType("_http._tcp.");
    serviceInfo.setPort(port);

    mNsdManager = Context.getSystemService(Context.NSD_SERVICE);

    mNsdManager.registerService(
        serviceInfo, NsdManager.PROTOCOL_DNS_SD, mRegistrationListener);
}
```

## Discover Services on the Network

The network is teeming with life, from the beastly network printers to the docile network webcams, to the brutal, fiery battles of nearby tic-tac-toe players. The key to letting your application see this vibrant ecosystem of functionality is service discovery. Your application needs to listen to service broadcasts on the network to see what services are available, and filter out anything the application can't work with.

Service discovery, like service registration, has two steps: setting up a discovery listener with the relevant callbacks, and making a single asynchronous API call to [discoverServices\(\)](#) ([/reference](#)

```
/android/net/nsd/NsdManager.html#discoverServices(java.lang.String, int,  
        android.net.nsd.NsdManager.DiscoveryListener)).
```

First, instantiate an anonymous class that implements [NsdManager.DiscoveryListener](#) ([/reference/android/net/nsd/NsdManager.DiscoveryListener.html](#)). The following snippet shows a simple example:

```
public void initializeDiscoveryListener() {  
  
    // Instantiate a new DiscoveryListener  
    mDiscoveryListener = new NsdManager.DiscoveryListener() {  
  
        // Called as soon as service discovery begins.  
        @Override  
        public void onDiscoveryStarted(String regType) {  
            Log.d(TAG, "Service discovery started");  
        }  
  
        @Override  
        public void onServiceFound(NsdServiceInfo service) {  
            // A service was found! Do something with it.  
            Log.d(TAG, "Service discovery success" + service);  
            if (!service.getServiceType().equals(SERVICE_TYPE)) {  
                // Service type is the string containing the protocol and  
                // transport layer for this service.  
                Log.d(TAG, "Unknown Service Type: " + service.getServiceType());  
            } else if (service.getServiceName().equals(mServiceName)) {  
                // The name of the service tells the user what they'd be  
                // connecting to. It could be "Bob's Chat App".  
                Log.d(TAG, "Same machine: " + mServiceName);  
            } else if (service.getServiceName().contains("NsdChat")){  
                mNsdManager.resolveService(service, mResolveListener);  
            }  
        }  
  
        @Override  
        public void onServiceLost(NsdServiceInfo service) {  
            // When the network service is no longer available.  
            // Internal bookkeeping code goes here.  
            Log.e(TAG, "service lost" + service);  
        }  
  
        @Override  
        public void onDiscoveryStopped(String serviceType) {  
            Log.i(TAG, "Discovery stopped: " + serviceType);  
        }  
  
        @Override  
        public void onStartDiscoveryFailed(String serviceType, int errorCode) {  
    }
```

```
        Log.e(TAG, "Discovery failed: Error code:" + errorCode);
        mNsdManager.stopServiceDiscovery(this);
    }

    @Override
    public void onStopDiscoveryFailed(String serviceType, int errorCode) {
        Log.e(TAG, "Discovery failed: Error code:" + errorCode);
        mNsdManager.stopServiceDiscovery(this);
    }
}
```

The NSD API uses the methods in this interface to inform your application when discovery is started, when it fails, and when services are found and lost (lost means "is no longer available"). Notice that this snippet does several checks when a service is found.

1. The service name of the found service is compared to the service name of the local service to determine if the device just picked up its own broadcast (which is valid).
2. The service type is checked, to verify it's a type of service your application can connect to.
3. The service name is checked to verify connection to the correct application.

Checking the service name isn't always necessary, and is only relevant if you want to connect to a specific application. For instance, the application might only want to connect to instances of itself running on other devices. However, if the application wants to connect to a network printer, it's enough to see that the service type is "\_ipp.\_tcp".

After setting up the listener, call [discoverServices\(\) \(/reference/android/net/nsd/NsdManager.html#discoverServices\(java.lang.String, int, android.net.nsd.NsdManager.DiscoveryListener\)\)](#), passing in the service type your application should look for, the discovery protocol to use, and the listener you just created.

```
mNsdManager.discoverServices(
    SERVICE_TYPE, NsdManager.PROTOCOL_DNS_SD, mDiscoveryListener);
```

## Connect to Services on the Network

When your application finds a service on the network to connect to, it must first determine the connection information for that service, using the [resolveService\(\) \(/reference/android/net/nsd/NsdManager.html#resolveService\(android.net.nsd.NsdServiceInfo, android.net.nsd.NsdManager.ResolveListener\)\)](#) method. Implement a [NsdManager.ResolveListener \(/reference/android/net/nsd/NsdManager.ResolveListener.html\)](#) to pass into this method, and use it to get a [NsdServiceInfo \(/reference/android/net/nsd/NsdServiceInfo.html\)](#) containing the connection information.

```
public void initializeResolveListener() {
    mResolveListener = new NsdManager.ResolveListener() {
```

```
@Override  
public void onResolveFailed(NsdServiceInfo serviceInfo, int errorCode) {  
    // Called when the resolve fails. Use the error code to debug.  
    Log.e(TAG, "Resolve failed" + errorCode);  
}  
  
@Override  
public void onServiceResolved(NsdServiceInfo serviceInfo) {  
    Log.e(TAG, "Resolve Succeeded. " + serviceInfo);  
  
    if (serviceInfo.getServiceName().equals(mServiceName)) {  
        Log.d(TAG, "Same IP.");  
        return;  
    }  
    mService = serviceInfo;  
    int port = mService.getPort();  
    InetAddress host = mService.getHost();  
}  
};  
}
```

Once the service is resolved, your application receives detailed service information including an IP address and port number. This is everything you need to create your own network connection to the service.

## Unregister Your Service on Application Close

It's important to enable and disable NSD functionality as appropriate during the application's lifecycle. Unregistering your application when it closes down helps prevent other applications from thinking it's still active and attempting to connect to it. Also, service discovery is an expensive operation, and should be stopped when the parent Activity is paused, and re-enabled when the Activity is resumed. Override the lifecycle methods of your main Activity and insert code to start and stop service broadcast and discovery as appropriate.

```
//In your application's Activity  
  
@Override  
protected void onPause() {  
    if (mNsdHelper != null) {  
        mNsdHelper.tearDown();  
    }  
    super.onPause();  
}  
  
@Override  
protected void onResume() {
```

```
super.onResume();
if (mNsdHelper != null) {
    mNsdHelper.registerService(mConnection.getLocalPort());
    mNsdHelper.discoverServices();
}
}

@Override
protected void onDestroy() {
    mNsdHelper.tearDown();
    mConnection.tearDown();
    super.onDestroy();
}

// NsdHelper's tearDown method
public void tearDown() {
    mNsdManager.unregisterService(mRegistrationListener);
    mNsdManager.stopServiceDiscovery(mDiscoveryListener);
}
```

# Connecting with Wi-Fi Direct

The Wi-Fi Direct™ APIs allow applications to connect to nearby devices without needing to connect to a network or hotspot. This allows your application to quickly find and interact with nearby devices, at a range beyond the capabilities of Bluetooth.

This lesson shows you how to find and connect to nearby devices using Wi-Fi Direct.

## Set Up Application Permissions

In order to use Wi-Fi Direct, add the [CHANGE\\_WIFI\\_STATE](#) ([/reference/android/Manifest.permission.html#CHANGE\\_WIFI\\_STATE](#)), [ACCESS\\_WIFI\\_STATE](#) ([/reference/android/Manifest.permission.html#ACCESS\\_WIFI\\_STATE](#)), and [INTERNET](#) ([/reference/android/Manifest.permission.html#INTERNET](#)) permissions to your manifest. Wi-Fi Direct doesn't require an internet connection, but it does use standard Java sockets, which require the [INTERNET](#) ([/reference/android/Manifest.permission.html#INTERNET](#)) permission. So you need the following permissions to use Wi-Fi Direct.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.nsdchat"
    ...
    <uses-permission
        android:required="true"
        android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.INTERNET"/>
    ...

```

### THIS LESSON TEACHES YOU HOW TO

1. [Set Up Application Permissions](#)
2. [Set Up the Broadcast Receiver and Peer-to-Peer Manager](#)
3. [Initiate Peer Discovery](#)
4. [Fetch the List of Peers](#)
5. [Connect to a Peer](#)

## Set Up a Broadcast Receiver and Peer-to-Peer Manager

To use Wi-Fi Direct, you need to listen for broadcast intents that tell your application when certain events have occurred. In your application, instantiate an [IntentFilter](#) ([/reference/android/content/IntentFilter.html](#)) and set it to listen for the following:

### WIFI\_P2P\_STATE\_CHANGED\_ACTION

Indicates whether Wi-Fi Peer-To-Peer (P2P) is enabled

### WIFI\_P2P\_PEERS\_CHANGED\_ACTION

Indicates that the available peer list has changed.

### WIFI\_P2P\_CONNECTION\_CHANGED\_ACTION

Indicates the state of Wi-Fi P2P connectivity has changed.

### WIFI\_P2P\_THIS\_DEVICE\_CHANGED\_ACTION

Indicates this device's configuration details have changed.

```
private final IntentFilter intentFilter = new IntentFilter();  
...  
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
  
    // Indicates a change in the Wi-Fi Peer-to-Peer status.  
    intentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION);  
  
    // Indicates a change in the list of available peers.  
    intentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION);  
  
    // Indicates the state of Wi-Fi P2P connectivity has changed.  
    intentFilter.addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION);  
  
    // Indicates this device's details have changed.  
    intentFilter.addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION);  
  
    ...  
}
```

At the end of the `onCreate()` ([/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](#)) method, get an instance of the `WifiP2pManager` ([/reference/android/net/wifi/p2p/WifiP2pManager.html](#)), and call its `initialize()` ([/reference/android/net/wifi/p2p/WifiP2pManager.html#initialize\(android.content.Context, android.os.Looper, android.net.wifi.p2p.WifiP2pManager.ChannelListener\)](#)) method. This method returns a `WifiP2pManager.Channel` ([/reference/android/net/wifi/p2p/WifiP2pManager.Channel.html](#)) object, which you'll use later to connect your app to the Wi-Fi Direct Framework.

```
@Override  
  
Channel mChannel;  
  
public void onCreate(Bundle savedInstanceState) {  
    ...  
    mManager = (WifiP2pManager) getSystemService(Context.WIFI_P2P_SERVICE);  
    mChannel = mManager.initialize(this, getMainLooper(), null);  
}
```

Now create a new `BroadcastReceiver` ([/reference/android/content/BroadcastReceiver.html](#)) class that

you'll use to listen for changes to the System's Wi-Fi P2P state. In the [onReceive\(\) \(/reference/android/content/BroadcastReceiver.html#onReceive\(android.content.Context, android.content.Intent\)\)](#) method, add a condition to handle each P2P state change listed above.

```
@Override  
public void onReceive(Context context, Intent intent) {  
    String action = intent.getAction();  
    if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {  
        // Determine if Wifi Direct mode is enabled or not, alert  
        // the Activity.  
        int state = intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE, -1);  
        if (state == WifiP2pManager.WIFI_P2P_STATE_ENABLED) {  
            activity.setIsWifiP2pEnabled(true);  
        } else {  
            activity.setIsWifiP2pEnabled(false);  
        }  
    } else if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {  
        // The peer list has changed! We should probably do something about  
        // that.  
    } else if (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {  
        // Connection state changed! We should probably do something about  
        // that.  
    } else if (WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION.equals(action)) {  
        DeviceListFragment fragment = (DeviceListFragment) activity.getFragmentManager().findFragmentById(R.id.frag_list);  
        fragment.updateThisDevice((WifiP2pDevice) intent.getParcelableExtra(WifiP2pManager.EXTRA_WIFI_P2P_DEVICE));  
    }  
}
```

Finally, add code to register the intent filter and broadcast receiver when your main activity is active, and unregister them when the activity is paused. The best place to do this is the [onResume\(\) \(/reference/android/app/Activity.html#onResume\(\)\)](#) and [onPause\(\) \(/reference/android/app/Activity.html#onPause\(\)\)](#) methods.

```
/** register the BroadcastReceiver with the intent values to be matched */  
@Override  
public void onResume() {  
    super.onResume();  
    receiver = new WiFiDirectBroadcastReceiver(mManager, mChannel, this);  
    registerReceiver(receiver, intentFilter);
```

```
    }

    @Override
    public void onPause() {
        super.onPause();
        unregisterReceiver(receiver);
    }
}
```

## Initiate Peer Discovery

To start searching for nearby devices with Wi-Fi Direct, call [discoverPeers\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#discoverPeers\(android.net.wifi.p2p.WifiP2pManager.Channel, android.net.wifi.p2p.WifiP2pManager.ActionListener\)](#)). This method takes the following arguments:

- The [WifiP2pManager.Channel](#) you received back when you initialized the peer-to-peer mManager
- An implementation of [WifiP2pManager.ActionListener](#) with methods the system invokes for successful and unsuccessful discovery.

```
mManager.discoverPeers(mChannel, new WifiP2pManager.ActionListener() {

    @Override
    public void onSuccess() {
        // Code for when the discovery initiation is successful goes here.
        // No services have actually been discovered yet, so this method
        // can often be left blank. Code for peer discovery goes in the
        // onReceive method, detailed below.
    }

    @Override
    public void onFailure(int reasonCode) {
        // Code for when the discovery initiation fails goes here.
        // Alert the user that something went wrong.
    }
});
```

Keep in mind that this only *initiates* peer discovery. The [discoverPeers\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#discoverPeers\(android.net.wifi.p2p.WifiP2pManager.Channel, android.net.wifi.p2p.WifiP2pManager.ActionListener\)](#)) method starts the discovery process and then immediately returns. The system notifies you if the peer discovery process is successfully initiated by calling methods in the provided action listener. Also, discovery will remain active until a connection is initiated or a P2P group is formed.

## Fetch the List of Peers

Now write the code that fetches and processes the list of peers. First implement the [WifiP2pManager.PeerListListener](#) ([/reference/android/net/wifi](#)

[/p2p/WifiP2pManager.PeerListListener.html](#) interface, which provides information about the peers that Wi-Fi Direct has detected. The following code snippet illustrates this.

```
private List peers = new ArrayList();
...
private PeerListListener peerListListener = new PeerListListener() {
    @Override
    public void onPeersAvailable(WifiP2pDeviceList peerList) {

        // Out with the old, in with the new.
        peers.clear();
        peers.addAll(peerList.getDeviceList());

        // If an AdapterView is backed by this data, notify it
        // of the change. For instance, if you have a ListView of available
        // peers, trigger an update.
        ((WiFiPeerListAdapter) getListAdapter()).notifyDataSetChanged();
        if (peers.size() == 0) {
            Log.d(WiFiDirectActivity.TAG, "No devices found");
            return;
        }
    }
}
```

Now modify your broadcast receiver's [onReceive\(\)](#) ([/reference/android/content/BroadcastReceiver.html#onReceive\(android.content.Context, android.content.Intent\)](#)) method to call [requestPeers\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#requestPeers\(android.net.wifi.p2p.WifiP2pManager.Channel, android.net.wifi.p2p.WifiP2pManager.PeerListListener\)](#)) when an intent with the action [WIFI\\_P2P\\_PEERS\\_CHANGED\\_ACTION](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#WIFI\\_P2P\\_PEERS\\_CHANGED\\_ACTION](#)) is received. You need to pass this listener into the receiver somehow. One way is to send it as an argument to the broadcast receiver's constructor.

```
public void onReceive(Context context, Intent intent) {
    ...
    else 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, peerListener);
        }
        Log.d(WiFiDirectActivity.TAG, "P2P peers changed");
    }...
```

}

Now, an intent with the action [WIFI\\_P2P\\_PEERS\\_CHANGED\\_ACTION](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#WIFI\\_P2P\\_PEERS\\_CHANGED\\_ACTION](#)) intent will trigger a request for an updated peer list.

## Connect to a Peer

In order to connect to a peer, create a new [WifiP2pConfig](#) ([/reference/android/net/wifi/p2p/WifiP2pConfig.html](#)) object, and copy data into it from the [WifiP2pDevice](#) ([/reference/android/net/wifi/p2p/WifiP2pDevice.html](#)) representing the device you want to connect to. Then call the [connect\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#connect\(android.net.wifi.p2p.WifiP2pManager.Channel, android.net.wifi.p2p.WifiP2pConfig, android.net.wifi.p2p.WifiP2pManager.ActionListener\)](#)) method.

```
@Override  
public void connect() {  
    // Picking the first device found on the network.  
    WifiP2pDevice device = peers.get(0);  
  
    WifiP2pConfig config = new WifiP2pConfig();  
    config.deviceAddress = device.deviceAddress;  
    config.wps.setup = WpsInfo.PBC;  
  
    mManager.connect(mChannel, config, new ActionListener() {  
  
        @Override  
        public void onSuccess() {  
            // WiFiDirectBroadcastReceiver will notify us. Ignore for now.  
        }  
  
        @Override  
        public void onFailure(int reason) {  
            Toast.makeText(WiFiDirectActivity.this, "Connect failed. Retry.",  
                           Toast.LENGTH_SHORT).show();  
        }  
    });  
}
```

The [WifiP2pManager.ActionListener](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.ActionListener.html](#)) implemented in this snippet only notifies you when the initiation succeeds or fails. To listen for changes in connection state, implement the [WifiP2pManager.ConnectionInfoListener](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.ConnectionInfoListener.html](#)) interface. Its [onConnectionInfoAvailable\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.ConnectionInfoListener.html#onConnectionInfoAvailable\(android.net.wifi.p2p.WifiP2pInfo\)](#)) callback will notify you when the state of the connection changes. In cases where multiple

devices are going to be connected to a single device (like a game with 3 or more players, or a chat app), one device will be designated the "group owner".

```
@Override  
public void onConnectionInfoAvailable(final WifiP2pInfo info) {  
  
    // InetAddress from WifiP2pInfo struct.  
    InetAddress groupOwnerAddress = info.groupOwnerAddress.getHostAddress();  
  
    // After the group negotiation, we can determine the group owner.  
    if (info.groupFormed && info.isGroupOwner) {  
        // Do whatever tasks are specific to the group owner.  
        // One common case is creating a server thread and accepting  
        // incoming connections.  
    } else if (info.groupFormed) {  
        // The other device acts as the client. In this case,  
        // you'll want to create a client thread that connects to the group  
        // owner.  
    }  
}
```

Now go back to the [onReceive\(\)](#) ([/reference/android/content/BroadcastReceiver.html#onReceive\(android.content.Context, android.content.Intent\)](#)) method of the broadcast receiver, and modify the section that listens for a [WIFI\\_P2P\\_CONNECTION\\_CHANGED\\_ACTION](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#WIFI\\_P2P\\_CONNECTION\\_CHANGED\\_ACTION](#)) intent. When this intent is received, call [requestConnectionInfo\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#requestConnectionInfo\(android.net.wifi.p2p.WifiP2pManager.Channel, android.net.wifi.p2p.WifiP2pManager.ConnectionInfoListener\)](#)). This is an asynchronous call, so results will be received by the connection info listener you provide as a parameter.

```
...  
} else if (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {  
  
    if (mManager == null) {  
        return;  
    }  
  
    NetworkInfo networkInfo = (NetworkInfo) intent  
        .getParcelableExtra(WifiP2pManager.EXTRA_NETWORK_INFO);  
  
    if (networkInfo.isConnected()) {  
  
        // We are connected with the other device, request connection  
        // info to find group owner IP  
  
        mManager.requestConnectionInfo(mChannel, connectionListener);  
    }  
}
```

}  
...

# Using Wi-Fi Direct for Service Discovery

The first lesson in this class, [Using Network Service Discovery \(nsd.html\)](#), showed you how to discover services that are connected to a local network. However, using Wi-Fi Direct™; Service Discovery allows you to discover the services of nearby devices directly, without being connected to a network. You can also advertise the services running on your device.

These capabilities help you communicate between apps, even when no local network or hotspot is available.

While this set of APIs is similar in purpose to the Network Service Discovery APIs outlined in a previous lesson, implementing them in code is very different. This lesson shows you how to discover services available from other devices, using Wi-Fi Direct™. The lesson assumes that you're already familiar with the [Wi-Fi Direct \(/guide/topics/connectivity/wifip2p.html\)](#) API.

## THIS LESSON TEACHES YOU TO

1. [Set Up the Manifest](#)
2. [Add a Local Service](#)
3. [Discover Nearby Services](#)

## Set Up the Manifest

In order to use Wi-Fi Direct, add the [CHANGE\\_WIFI\\_STATE \(/reference/android/Manifest.permission.html#CHANGE\\_WIFI\\_STATE\)](#), [ACCESS\\_WIFI\\_STATE \(/reference/android/Manifest.permission.html#ACCESS\\_WIFI\\_STATE\)](#), and [INTERNET \(/reference/android/Manifest.permission.html#INTERNET\)](#) permissions to your manifest. Even though Wi-Fi Direct doesn't require an Internet connection, it uses standard Java sockets, and using these in Android requires the requested permissions.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.nsdchat"

    ...

    <uses-permission
        android:required="true"
        android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission
        android:required="true"
        android:name="android.permission.INTERNET"/>
    ...
}
```

## Add a Local Service

If you're providing a local service, you need to register it for service discovery. Once your local

service is registered, the framework automatically responds to service discovery requests from peers.

To create a local service:

1. Create a `WifiP2pServiceInfo` object.
2. Populate it with information about your service.
3. Call `addLocalService()` to register the local service for service discovery.

```
private void startRegistration() {
    // Create a string map containing information about your service.
    Map record = new HashMap();
    record.put("listenport", String.valueOf(SERVER_PORT));
    record.put("buddyname", "John Doe" + (int) (Math.random() * 1000));
    record.put("available", "visible");

    // Service information. Pass it an instance name, service type
    // _protocol._transportlayer , and the map containing
    // information other devices will want once they connect to this one.
    WifiP2pDnsSdServiceInfo serviceInfo =
        WifiP2pDnsSdServiceInfo.newInstance("_test", "_presence._tcp", re

    // Add the local service, sending the service info, network channel,
    // and listener that will be used to indicate success or failure of
    // the request.
    mManager.addLocalService(channel, serviceInfo, new ActionListener() {
        @Override
        public void onSuccess() {
            // Command successful! Code isn't necessarily needed here,
            // Unless you want to update the UI or add logging statements.
        }

        @Override
        public void onFailure(int arg0) {
            // Command failed. Check for P2P_UNSUPPORTED, ERROR, or BUSY
        }
    });
}
```

## Discover Nearby Services

Android uses callback methods to notify your application of available services, so the first thing to do is set those up. Create a `WifiP2pManager.DnsSdTxtRecordListener` (</reference/android/net/wifi/p2p/WifiP2pManager.DnsSdTxtRecordListener.html>) to listen for incoming records. This record can optionally be broadcast by other devices. When one comes in, copy the device address and any other relevant information you want into a data structure external to the current method, so you can access it later. The following example assumes that the record contains a "buddyname" field, populated with the user's identity.

```

final HashMap<String, String> buddies = new HashMap<String, String>();
...
private void discoverService() {
    DnsSdTxtRecordListener txtListener = new DnsSdTxtRecordListener() {
        @Override
        /* Callback includes:
         * fullDomain: full domain name: e.g "printer._ipp._tcp.local."
         * record: TXT record data as a map of key/value pairs.
         * device: The device running the advertised service.
         */
        public void onDnsSdTxtRecordAvailable(
            String fullDomain, Map record, WifiP2pDevice device) {
            Log.d(TAG, "DnsSdTxtRecord available - " + record.toString());
            buddies.put(device.deviceAddress, record.get("buddynname"));
        }
    };
    ...
}

```

To get the service information, create a [WifiP2pManager.DnsSdServiceResponseListener](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.DnsSdServiceResponseListener.html](#)). This receives the actual description and connection information. The previous code snippet implemented a [Map](#) ([/reference/java/util/Map.html](#)) object to pair a device address with the buddy name. The service response listener uses this to link the DNS record with the corresponding service information. Once both listeners are implemented, add them to the [WifiP2pManager](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html](#)) using the [setDnsSdResponseListeners\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#setDnsSdResponseListeners\(android.net.wifi.p2p.WifiP2pManager.Channel, android.net.wifi.p2p.WifiP2pManager.DnsSdServiceResponseListener, android.net.wifi.p2p.WifiP2pManager.DnsSdTxtRecordListener\)](#) method.

```

private void discoverService() {
    ...

    DnsSdServiceResponseListener servListener = new DnsSdServiceResponseListener(
        @Override
        public void onDnsSdServiceAvailable(String instanceName, String registrat
            WifiP2pDevice resourceType) {

            // Update the device name with the human-friendly version from
            // the DnsTxtRecord, assuming one arrived.
            resourceType.deviceName = buddies
                .containsKey(resourceType.deviceAddress) ? buddies
                .get(resourceType.deviceAddress) : resourceType.deviceNam

            // Add to the custom adapter defined specifically for showing
            // wifi devices.
        }
    );
}

```

```

        WiFiDirectServicesList fragment = (WiFiDirectServicesList) getFra
            .findFragmentById(R.id.frag_peerlist);
        WiFiDevicesAdapter adapter = ((WiFiDevicesAdapter) fragment
            .getListAdapter());

        adapter.add(resourceType);
        adapter.notifyDataSetChanged();
        Log.d(TAG, "onBonjourServiceAvailable " + instanceName);
    }
};

mManager.setDnsSdResponseListeners(channel, servListener, txtListener);
...
}

```

Now create a service request and call [addServiceRequest\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#addServiceRequest\(android.net.wifi.p2p.WifiP2pManager.Channel, android.net.wifi.p2p.nsd.WifiP2pServiceRequest, android.net.wifi.p2p.WifiP2pManager.ActionListener\)](#)). This method also takes a listener to report success or failure.

```

serviceRequest = WifiP2pDnsSdServiceRequest.newInstance();
mManager.addServiceRequest(channel,
    serviceRequest,
    new ActionListener() {
        @Override
        public void onSuccess() {
            // Success!
        }

        @Override
        public void onFailure(int code) {
            // Command failed. Check for P2P_UNSUPPORTED, ERROR, or ...
        }
    });

```

Finally, make the call to [discoverServices\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.html#discoverServices\(android.net.wifi.p2p.WifiP2pManager.Channel, android.net.wifi.p2p.WifiP2pManager.ActionListener\)](#)).

```

mManager.discoverServices(channel, new ActionListener() {

    @Override
    public void onSuccess() {
        // Success!
    }

    @Override

```

```
public void onFailure(int code) {  
    // Command failed. Check for P2P_UNSUPPORTED, ERROR, or BUSY  
    if (code == WifiP2pManager.P2P_UNSUPPORTED) {  
        Log.d(TAG, "P2P isn't supported on this device.");  
    } else if (...) {  
        ...  
    }  
};
```

If all goes well, hooray, you're done! If you encounter problems, remember that the asynchronous calls you've made take an [WifiP2pManager.ActionListener](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.ActionListener.html](#)) as an argument, and this provides you with callbacks indicating success or failure. To diagnose problems, put debugging code in [onFailure\(\)](#) ([/reference/android/net/wifi/p2p/WifiP2pManager.ActionListener.html#onFailure\(int\)](#)). The error code provided by the method hints at the problem. Here are the possible error values and what they mean

#### P2P\_UNSUPPORTED

Wi-Fi Direct isn't supported on the device running the app.

#### BUSY

The system is too busy to process the request.

#### ERROR

The operation failed due to an internal error.