Application Name: The name by which application will be visible to the Android Users.

Project Name: Name of the project. This will be the folder name on Hard disk and the name by which the project will be visible in Eclipse.

Package Name: General package name

Minimum required SDK: Lowest level of Android supported by the application. It is defined by API Level.

Target SDK: Highest version of Android supported by the Application.

Compile with: The platform the application is compiled on.

Theme: Specifies the UI design for the Application.

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.

One of the most important elements your manifest should include is the <uses-sdk> element. This declares your app's compatibility with different Android versions using the android:minSdkVersion and android:targetSdkVersion attributes. For your first app, it should look like this:

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

<uses-sdk android:minSdkVersion="8" android:targetSdkVersion="17" />

...

</manifest>

You should always set the android:targetSdkVersion as high as possible and test your app on the corresponding platform version.

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 class starts and loads a layout file that says "Hello World."

Building a Simple User Interface

The graphical user interface for an Android app is built using a hierarchy of View and ViewGroup objects. View objects are usually UI widgets such as buttons or text fields and ViewGroup 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 and ViewGroup so you can define your UI in XML using a hierarchy of UI elements.



Create a Linear Layout

*Open the activity\_main.xml file from the res/layout/ directory.*

The BlankActivity template you chose when you created this project includes the activity\_main.xml file with a RelativeLayout root view and a TextView child view.

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

The other two attributes, android:layout\_width and android:layout\_height, are required for all views in order to specify their size.

Because the LinearLayout 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.

Add a Text Field

To create a user-editable text field, add an <EditText> element inside the <LinearLayout>.

Like every View object, you must define certain XML attributes to specify the EditText object's properties.

<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.

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.

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.

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.

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.

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.

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.

Add a Button

Now add a <Button> to the layout, immediately following the <EditText> 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 attribute, because it won't be referenced from the activity code.

Make the Input Box Fill in the Screen WidthThe layout is currently designed so that both the EditText and Button widgets are only as big as necessary to fit their 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.

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.

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.

Starting Another Activity

Respond to the Send Button

To respond to the button's on-click event, open the activity\_main.xml layout file and add the android:onClick attribute to the <Button> element:

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="@string/button\_send"

android:onClick="sendMessage" />

The 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 (located in the project's src/ directory) and add the corresponding method.

In order for the system to match this method to the method name given to 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)

Build an Intent

An Intent is an object that provides runtime binding between separate components (such as two activities). The Intent 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 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)

An intent not only allows you to start another activity, but it can carry a bundle of data to the activity as well. Inside the sendMessage() method, use findViewById() to get the EditText element and add its text value to the intent:

EditText editText = (EditText) findViewById(R.id.edit\_message);

String message = editText.getText().toString();

intent.putExtra(EXTRA\_MESSAGE, message);

An Intent can carry a collection of various data types as key-value pairs called extras. The putExtra() 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 the key for your intent's extra using a public constant. So add the EXTRA\_MESSAGE definition to the top of the MainActivity class:

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, call startActivity() and pass it your Intent. The system receives this call and starts an instance of the Activity specified by the Intent.

All subclasses of Activity must implement the onCreate() method. The system calls this when creating a new instance of the activity. This method is where you must define the activity layout with the setContentView() method and is where you should perform initial setup for the activity components.

Add it to the manifest

All activities must be declared in your manifest file, AndroidManifest.xml, using an <activity> element.

The android:parentActivityName attribute declares the name of this activity's parent activity within the app's logical hierarchy. The system uses this value to implement default navigation behaviors, such as Up navigation on Android 4.1 (API level 16) and higher.

Receive the Intent

Every Activity is invoked by an Intent, regardless of how the user navigated there. You can get the Intent that started your activity by calling getIntent() and retrieve the data contained within it.

In the DisplayMessageActivity class’s onCreate() 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 widget and set the text using setText(). Then add the TextView as the root view of the activity’s layout by passing it to setContentView().