# Building Your First App

Get started

Dependencies

Android Studio

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.

## Set Up Your Environment

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

1. Download Android Studio.
2. Download the latest SDK tools and platforms using the SDK Manager.

> **Note:** Although most of this training class expects that you're using Android Studio, some procedures include alternative instructions for using the SDK tools from the command line instead.

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

Get started

# Creating an Android Project

An Android project contains all the files that comprise the source code for your Android app.

This lesson shows how to create a new project either using Android Studio or using the SDK tools from a command line.

**Note:** You should already have the Android SDK installed, and if you're using Android Studio, you should also have Android Studio installed. If you don't have these, follow the guide to Installing the Android SDK before you start this lesson.

## Create a Project with Android Studio

1. In Android Studio, create a new project:
   - If you don't have a project opened, in the **Welcome** screen, click **New Project**.
   - If you have a project opened, from the **File** menu, select **New Project**. The *Create New Project* screen appears.

2. Fill out the fields on the screen, and click **Next**.
   It is easier to follow these lessons if you use the same values as shown.

   - **Application Name** is the app name that appears to users. For this project, use "My First App."
   - **Company domain** provides a qualifier that will be appended to the package name; Android Studio will remember this qualifier for each new project you create.
   - **Package name** is the fully qualified name for the project (following the same rules as those for naming packages in the Java programming language). Your package name must be unique across all packages installed on the Android system. You can **Edit** this value independently from the application name or the company domain.
   - **Project location** is the directory on your system that holds the project files.

3. Under **Select the form factors your app will run on**, check the box for **Phone and Tablet**.
4. For **Minimum SDK**, select **API 8: Android 2.2 (Froyo)**.
   The Minimum Required SDK is the earliest version of Android that your app supports, indicated using the API level. 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 (as discussed in Supporting Different Platform Versions).

5. Leave all of the other options (TV, Wear, and Glass) unchecked and click **Next.**

   ### Activities

An activity is one of the distinguishing features of the Android framework. Activities provide the user with access to your app, and there may be many activities. An application will usually have a main activity for when the user launches the application, another activity for when she selects some content to view, for example, and other activities for when she performs other tasks within the app. See Activities for more information.

6. Under **Add an activity to <*template*>**, select **Blank Activity** and click **Next**.
7. Under **Customize the Activity**, change the **Activity Name** to *MyActivity*. The **Layout Name** changes to *activity_my*, and the **Title** to *MyActivity*. The **Menu Resource Name** is *menu_my*.
8. Click the **Finish** button to create the project.

Your Android project is now a basic "Hello World" app that contains some default files. Take a moment to review the most important of these:

`app/src/main/res/layout/activity_my.xml`
This XML layout file is for the activity you added when you created the project with Android Studio. Following the New Project workflow, Android Studio presents this file with both a text view and a preview of the screen UI. The file contains some default interface elements from the material design library, including the app bar and a floating action button. It also includes a separate layout file with the main content.

`app/src/main/res/layout/content_my.xml`
This XML layout file resides in `activity_my.xml`, and contains some settings and a `TextView` element that displays the message, "Hello world!".

`app/src/main/java/com.mycompany.myfirstapp/MyActivity.java`
A tab for this file appears in Android Studio when the New Project workflow finishes. When you select the file you see the class definition for the activity you created. When you build and run the app, the `Activity` class starts the activity and loads the layout file that says "Hello World!"

`app/src/main/AndroidManifest.xml`
The manifest file describes the fundamental characteristics of the app and defines each of its components. You'll revisit this file as you follow these lessons and add more components to your app.

`app/build.gradle`
Android Studio uses Gradle to compile and build your app. There is a `build.gradle` file for each module of your project, as well as a `build.gradle` file for the entire project. Usually, you're only interested in the `build.gradle` file for the module, in this case the `app` or application module. This is where your app's build dependencies are set, including the `defaultConfig` settings:

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

- `applicationId` is the fully qualified package name for your application that you specified during the New Project workflow.
- `minSdkVersion` is the Minimum SDK version you specified during the New Project workflow. This is the earliest version of the Android SDK that your app supports.
- `targetSdkVersion` indicates the highest version of Android with which you have tested your application. As new versions of Android become available, you should test your app on the new version and update this value to match the latest API level and thereby take advantage of new platform features. For more information, read Supporting Different Platform Versions.

Note also the `/res` subdirectories that contain the resources for your application:

`drawable-<density>/`
Directories for drawable resources, other than launcher icons, designed for various densities.

`layout/`
Directory for files that define your app's user interface like `activity_my.xml`, discussed above, which describes a basic layout for the `MyActivity` class.

`menu/`
Directory for files that define your app's menu items.

`mipmap/`
Launcher icons reside in the `mipmap/` folder rather than the `drawable/` folders. This folder contains the `ic_launcher.png` image that appears when you run the default app.

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

To run the app, continue to the next lesson.

# Running Your App

If you followed the previous lesson 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 device running Android and whether you're using Android Studio. 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 Android Studio or the command line tools.

## Run on a Real Device

If you have a device running Android, here's how to install and run your app.

### Set up your device

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.
2. Enable **USB debugging** on your device.
    - On most devices running Android 3.2 or older, you can find the option under **Settings > Applications > Development**.
    - On Android 4.0 and newer, it's in **Settings > Developer options**.
      **Note:** On Android 4.2 and newer, **Developer options** is hidden by default. To make it available, go to **Settings > About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options**.

### Run the app from Android Studio

1. Select one of your project's files and click **Run**

   from the toolbar.
2. In the **Choose Device** window that appears, select the **Choose a running device** radio button, select your device, and click **OK** .

Android Studio installs the app on your connected device and starts it.

### Run the app from a command line

Open a command-line and navigate to the root of your project directory. Use Gradle to build your project in debug mode, invoke the `assembleDebug` build task using the Gradle wrapper script (`gradlew assembleRelease`).

This creates your debug `.apk` file inside the module `build/` directory, named `app-debug.apk`.

On Windows platforms, type this command:

```
> gradlew.bat assembleDebug
```

On Mac OS and Linux platforms, type these commands:

```
$ chmod +x gradlew
$ ./gradlew assembleDebug
```

After you build the project, the output APK for the app module is located in `app/build/outputs/apk/`

**Note:** The first command (`chmod`) adds the execution permission to the Gradle wrapper script and is only necessary the first time you build this project from the command line.

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

```
$ adb install app/build/outputs/apk/app-debug.apk
```

On your device, locate *MyFirstApp* and open it.

That's how you build and run your Android app on a device! To start developing, continue to the next lesson.

## Run on the Emulator

Whether you're using Android Studio or the command line, to run your app on the emulator you need to first create an Android Virtual Device (AVD). An AVD is a device configuration for the Android emulator that allows you to model a specific device.

### Create an AVD

1. Launch the Android Virtual Device Manager:
   - In Android Studio, select **Tools > Android > AVD Manager**, or click the AVD Manager icon

     in the toolbar. The *AVD Manager* screen appears.
   - Or, from the command line, change directories to `sdk/` and execute:

     ```
     tools/android avd
     ```

> **Note:** The AVD Manager that appears when launched from the command line is different from the version in Android Studio, so the following instructions may not all apply.

2. On the AVD Manager main screen, click **Create Virtual Device**.
3. In the Select Hardware window, select a device configuration, such as Nexus 6, then click **Next**.
4. Select the desired system version for the AVD and click **Next**.
5. Verify the configuration settings, then click **Finish**.

For more information about using AVDs, see Managing AVDs with AVD Manager.

## Run the app from Android Studio

1. In **Android Studio**, select your project and click **Run**

   from the toolbar.
2. In the **Choose Device** window, click the **Launch emulator** radio button.
3. From the **Android virtual device** pull-down menu, select the emulator you created, and click **OK**.

It can take a few minutes for the emulator to load itself. You may have to unlock the screen. When you do, *My First App* appears on the emulator screen.

## Run your app from the command line

1. Build the project from the command line. The output APK for the app module is located in `app/build/outputs/apk/`.
2. Make sure the Android SDK `platform-tools/` directory is included in your `PATH` environment variable.
3. Execute this command:

   ```
   $ adb install app/build/outputs/apk/apk-debug.apk
   ```

4. On the emulator, locate *MyFirstApp* 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 a Simple User Interface

In this lesson, you create a layout in XML that includes a text field and a button. In the next lesson, your app responds when the button is pressed by sending the content of the text field to another activity.

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. `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.

Layouts are subclasses of the `ViewGroup`. In this exercise, you'll work with a `LinearLayout`.

## 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.

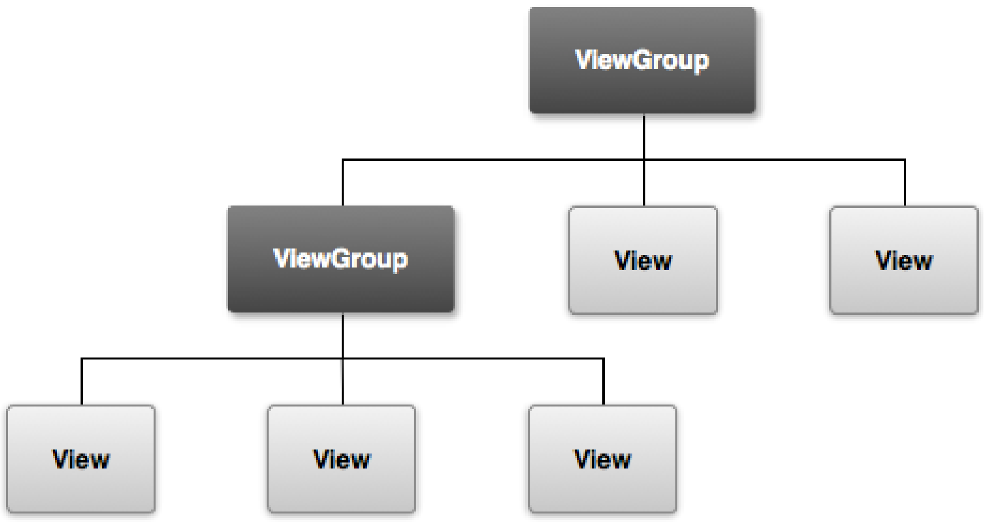

**Figure 1.** Illustration of how `ViewGroup` objects form branches in the layout and contain other `View` objects.

## Create a Linear Layout

1. In Android Studio, from the `res/layout` directory, open the `content_my.xml` file.
   The BlankActivity template you chose when you created this project includes the `content_my.xml` file with a `RelativeLayout` root view and a `TextView` child view.

2. In the **Preview** pane, click the Hide icon

to close the Preview pane.
In Android Studio, when you open a layout file, you’re first shown the Preview pane. Clicking elements in this pane opens the WYSIWYG tools in the Design pane. For this lesson, you’re going to work directly with the XML.

3. Delete the `<TextView>` element.
4. Change the `<RelativeLayout>` element to `<LinearLayout>`.
5. Add the `android:orientation` attribute and set it to `"horizontal"`.
6. Remove the `android:padding` attributes and the `tools:context` attribute.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_my">
```

`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.

Two other 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.

For more information about layout properties, see the Layout guide.

## Add a Text Field

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

1. In the `content_my.xml` file, within the `<LinearLayout>` element, define an `<EditText>` element with the `id` attribute set to `@+id/edit_message`.
2. Define the `layout_width` and `layout_height` attributes as `wrap_content`.
3. Define a `hint` attribute as a string object named `edit_message`.

The `<EditText>` element should read as follows:

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

Here are the `<EditText>` attributes you added:

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`).

## Resource Objects

A resource object is 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` file each time you compile your app. You should never modify this file by hand.

For more information, read the guide to Providing Resources.

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. With the resource ID 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 sidebox for more information about resource objects.

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.

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.

## Add String Resources

By default, your Android project includes a string resource file at `res/values/strings.xml`. Here, you'll add a new string named `"edit_message"` and set the value to "Enter a message."

1. In Android Studio, from the `res/values` directory, open `strings.xml`.
2. Add a line for a string named `"edit_message"` with the value, "Enter a message".
3. Add a line for a string named `"button_send"` with the value, "Send".
   You'll create the button that uses this string in the next section.

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="action_settings">Settings</string>
</resources>
```

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

For more information about using string resources to localize your app for other languages, see the Supporting Different Devices class.

## Add a Button

1. In Android Studio, from the `res/layout` directory, edit the `content_my.xml` file.
2. Within the `<LinearLayout>` element, define a `<Button>` element immediately following the `<EditText>` element.
3. Set the button's width and height attributes to `"wrap_content"` so the button is only as big as necessary to fit the button's text label.
4. Define the button's text label with the `android:text` attribute; set its value to the `button_send` string resource you defined in the previous section.

Your `<LinearLayout>` should look like this:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_my">
        <EditText android:id="@+id/edit_message"
          android:layout_width="wrap_content"
          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>
```

**Note:** This button doesn't need the `android:id` attribute, because it won't be referenced from the activity code.

The layout is currently designed so that both the `EditText` and `Button` widgets are only as big as necessary to fit their content, as Figure 2 shows.

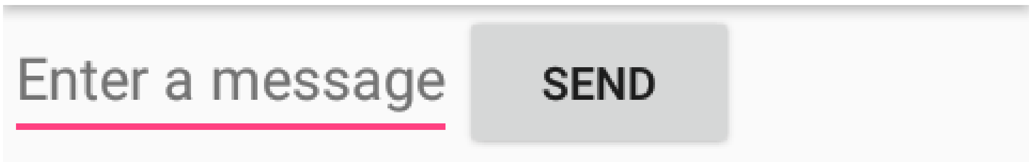

**Figure 2.** The `EditText` and `Button` widgets have their widths set to `"wrap_content"`.

This works fine for the button, but not as well for the text field, because the user might type something longer. 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 soda, 1 part syrup" means two-thirds of the drink is soda. 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.

## Make the Input Box Fill in the Screen Width

To fill the remaining space in your layout with the `EditText` element, do the following:

1. In the `content_my.xml` file, assign the `<EditText>` element's `layout_weight` attribute a value of `1`.
2. Also, assign `<EditText>` element's `layout_width` attribute a value of `0dp`.

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

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.

Figure 3 shows the result when you assign all weight to the `EditText` element.

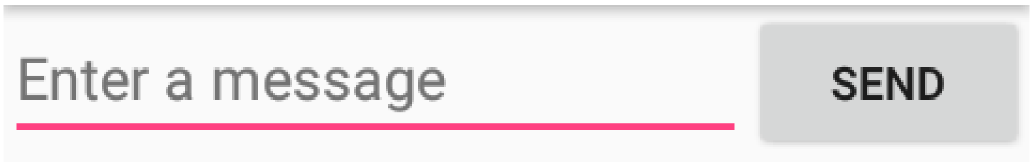

**Figure 3.** The `EditText` widget is given all the layout weight, so it fills the remaining space in the `LinearLayout`.

Here's how your complete `content_my.xml` layout file should now look:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:orientation="horizontal"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   app:layout_behavior="@string/appbar_scrolling_view_behavior"
   tools:showIn="@layout/activity_my">
    <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>
```

## Run Your App

This layout is applied by the default `Activity` class that the SDK tools generated when you created the project. Run the app to see the results:

- In Android Studio, from the toolbar, click **Run**

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

  ```
  $ ant debug
  adb install -r app/build/outputs/apk/app-debug.apk
  ```

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

The result looks like this:

# Starting Another Activity

After completing the previous lesson, 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 `MyActivity` that starts a new activity when the user clicks the Send button.

## Respond to the Send Button

1. In Android Studio, from the `res/layout` directory, edit the `content_my.xml` file.
2. Add the `android:onClick` attribute to the `<Button>` element.
   res/layout/content_my.xml

   ```
   <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.

3. In the `java/com.mycompany.myfirstapp` directory, open the `MyActivity.java` file.
4. Within the `MyActivity` class, add the `sendMessage()` method stub shown below.
   java/com.mycompany.myfirstapp/MyActivity.java

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

   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)

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

## Build an Intent

1. In `MyActivity.java`, inside the `sendMessage()` method, create an `Intent` to start an activity called `DisplayMessageActivity` with the following code:
   java/com.mycompany.myfirstapp/MyActivity.java

   ```
   public void sendMessage(View view) {
     Intent intent = new Intent(this, DisplayMessageActivity.class);
   }
   ```

   ### Intents

   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. For more information, see Intents and Intent Filters.

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

   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)

   Android Studio indicates that you must import the `Intent` class.

2. At the top of the file, import the `Intent` class:
   java/com.mycompany.myfirstapp/MyActivity.java

   ```
   import android.content.Intent;
   ```

   **Tip:** In Android Studio, press Alt + Enter (option + return on Mac) to import missing classes.

3. Inside the `sendMessage()` method, use `findViewById()` to get the `EditText` element.
   java/com.mycompany.myfirstapp/MyActivity.java

   ```
   public void sendMessage(View view) {
     Intent intent = new Intent(this, DisplayMessageActivity.class);
     EditText editText = (EditText) findViewById(R.id.edit_message);
   }
   ```

4. At the top of the file, import the `EditText` class.

In Android Studio, press Alt + Enter (option + return on Mac) to import missing classes.

5. Assign the text to a local `message` variable, and use the `putExtra()` method to add its text value to the intent.
   java/com.mycompany.myfirstapp/MyActivity.java

   ```
   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);
   }
   ```

   An `Intent` can carry 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.

6. At the top of the `MyActivity` class, add the `EXTRA_MESSAGE` definition as follows:
   java/com.mycompany.myfirstapp/MyActivity.java

   ```
   public class MyActivity extends AppCompatActivity {
       public final static String EXTRA_MESSAGE =
   "com.mycompany.myfirstapp.MESSAGE";
       ...
   }
   ```

   For the next activity to query the extra data, you should define the key for your intent's extra using a public constant. It's generally a good practice to define keys for intent extras using your app's package name as a prefix. This ensures the keys are unique, in case your app interacts with other apps.

7. In the `sendMessage()` method, to finish the intent, call the `startActivity()` method, passing it the `Intent` object created in step 1.

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

java/com.mycompany.myfirstapp/MyActivity.java

```
/** 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);
}
```

The system receives this call and starts an instance of the `Activity` specified by the `Intent`. Now you need to create the `DisplayMessageActivity` class in order for this to work.

## Create the Second Activity

All subclasses of `Activity` must implement the `onCreate()` method. This method is where the activity receives the intent with the message, then renders the message. Also, the `onCreate()` method must define the activity layout with the `setContentView()` method. This is where the activity performs the initial setup of the activity components.

### Create a new activity using Android Studio

Android Studio includes a stub for the `onCreate()` method when you create a new activity. The *New Android Activity* window appears.

1. In Android Studio, in the `java` directory, select the package, **com.mycompany.myfirstapp**, right-click, and select **New > Activity > Blank Activity**.
2. In the **Choose options** window, fill in the activity details:
   - **Activity Name**: DisplayMessageActivity
   - **Layout Name**: activity_display_message
   - **Title**: My Message
   - **Hierarchical Parent**: com.mycompany.myfirstapp.MyActivity
   - **Package name**: com.mycompany.myfirstapp

   Click **Finish**.

3. Open the `DisplayMessageActivity.java` file.
   The class already includes an implementation of the required `onCreate()` method. You update the implementation of this method later.

If you're developing with Android Studio, you can run the app now, but not much happens. Clicking the Send button starts the second activity, but it uses a default "Hello world" layout provided by the template. You'll soon update the activity to instead display a custom text view.

### Create the activity without Android Studio

If you're using a different IDE or the command line tools, do the following:

1. Create a new file named `DisplayMessageActivity.java` in the project's `src/` directory, next to the original `MyActivity.java` file.
2. Add the following code to the file:

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

        if (savedInstanceState == null) {
            getSupportFragmentManager().beginTransaction()
                .add(R.id.container, new
PlaceholderFragment()).commit();
        }
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle app bar item clicks here. The app bar
        // automatically handles clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();
        if (id == R.id.action_settings) {
            return true;
        }
        return super.onOptionsItemSelected(item);
    }

    /**
     * A placeholder fragment containing a simple view.
     */
    public static class PlaceholderFragment extends Fragment {

        public PlaceholderFragment() { }

        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup
container,
                  Bundle savedInstanceState) {
              View rootView =
inflater.inflate(R.layout.fragment_display_message,
                      container, false);
              return rootView;
        }
    }
}
```

**Note:** If you are using an IDE other than Android Studio, your project does not contain the `activity_display_message` layout that's requested by `setContentView()`. That's OK because you will update this method later and won't be using that layout.

3. To your `strings.xml` file, add the new activity's title as follows:

```
<resources>
    ...
    <string name="title_activity_display_message">My Message</string>
</resources>
```

4. In your manifest file, `AndroidManifest.xml`, within the `Application` element, add the `<activity>` element for your `DisplayMessageActivity` class, as follows:

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

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. You can provide the same navigation behaviors for older versions of Android by using the Support Library and adding the `<meta-data>` element as shown here.

**Note:** Your Android SDK should already include the latest Android Support Library, which you installed during the Adding SDK Packages step. When using the templates in Android Studio, the Support Library is automatically added to your app project (you can see the library's JAR file listed under *Android Dependencies*). If you're not using Android Studio, you need to manually add the library to your project—follow the guide for setting up the Support Library then return here.

If you're using a different IDE than Android Studio, don't worry that the app won't yet compile. You'll soon update the activity to display a custom text view.

## 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 the intent.

1. In the `java/com.mycompany.myfirstapp` directory, edit the `DisplayMessageActivity.java` file.
2. Get the intent and assign it to a local variable.

```
Intent intent = getIntent();
```