# Our First Android Application

This lecture covers:

* Create a new project in Android Studio
* The MainActivity UI and code
* Intents and starting another activity, namely, the DisplayMessageActivity
* The DisplayMessageActivity UI and code
* Application logging

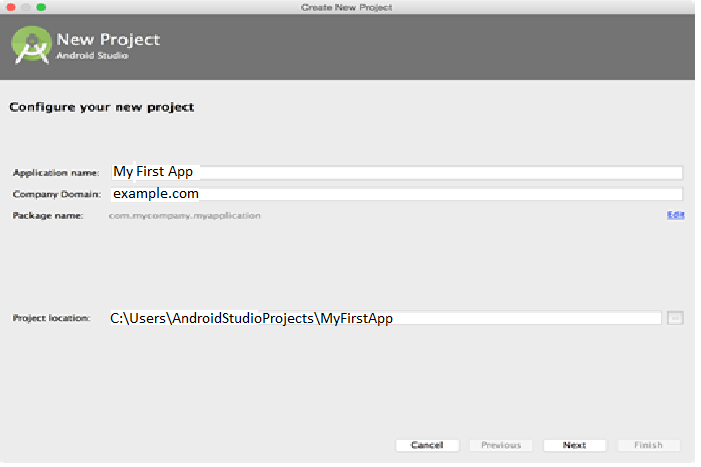
## Create a Project

### Step 1.

If you didn't have a project opened, Android Studio shows the Welcome screen. To create a new project, click **Start a New Android Studio project**.

If you had a project opened, Android Studio shows the development environment. To create a new project, click **File** > **New** > **New Project**.

The next window lets you configure the name of your app, the package name, and the location of your project.



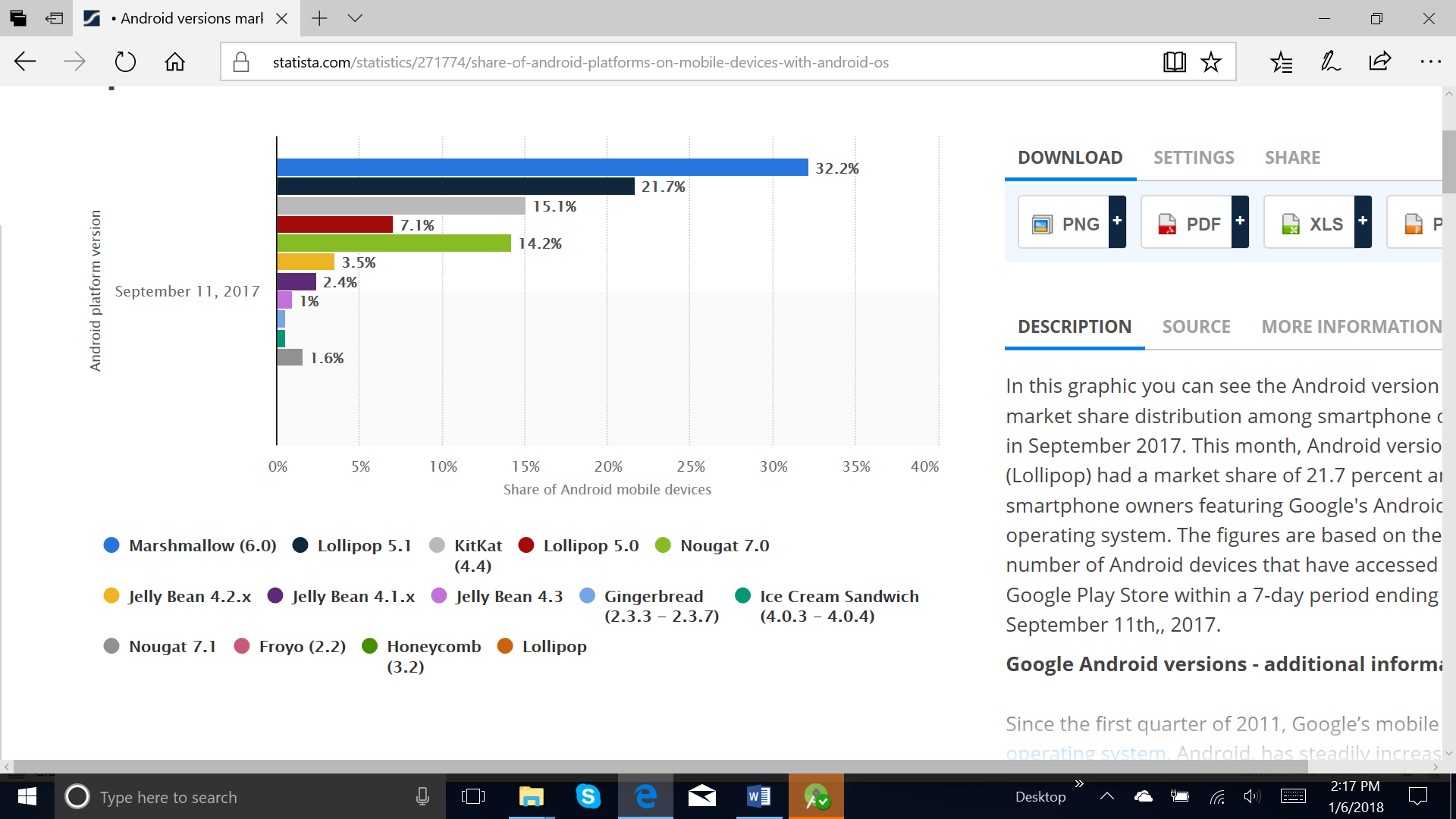
**Figure 1.** The **Configure your new project** screen.

In the **New Project** screen, enter as **Application Name**: "My First App" and as **Company Domain**: "example.com" then click **Next**. (The domain name is used by Android Studio to generate a package name. It can be anything you want as long as it is unique. Generally, we use reverse domain names like com.something. or org.something. structure for naming java packages. With programmers worldwide writing classes and interfaces using the Java programming language, it is likely that many programmers will use the same name for different types. Package names are written in all lower case to avoid conflict with the names of classes or interfaces.)

### Step 2: Select form factors and API level

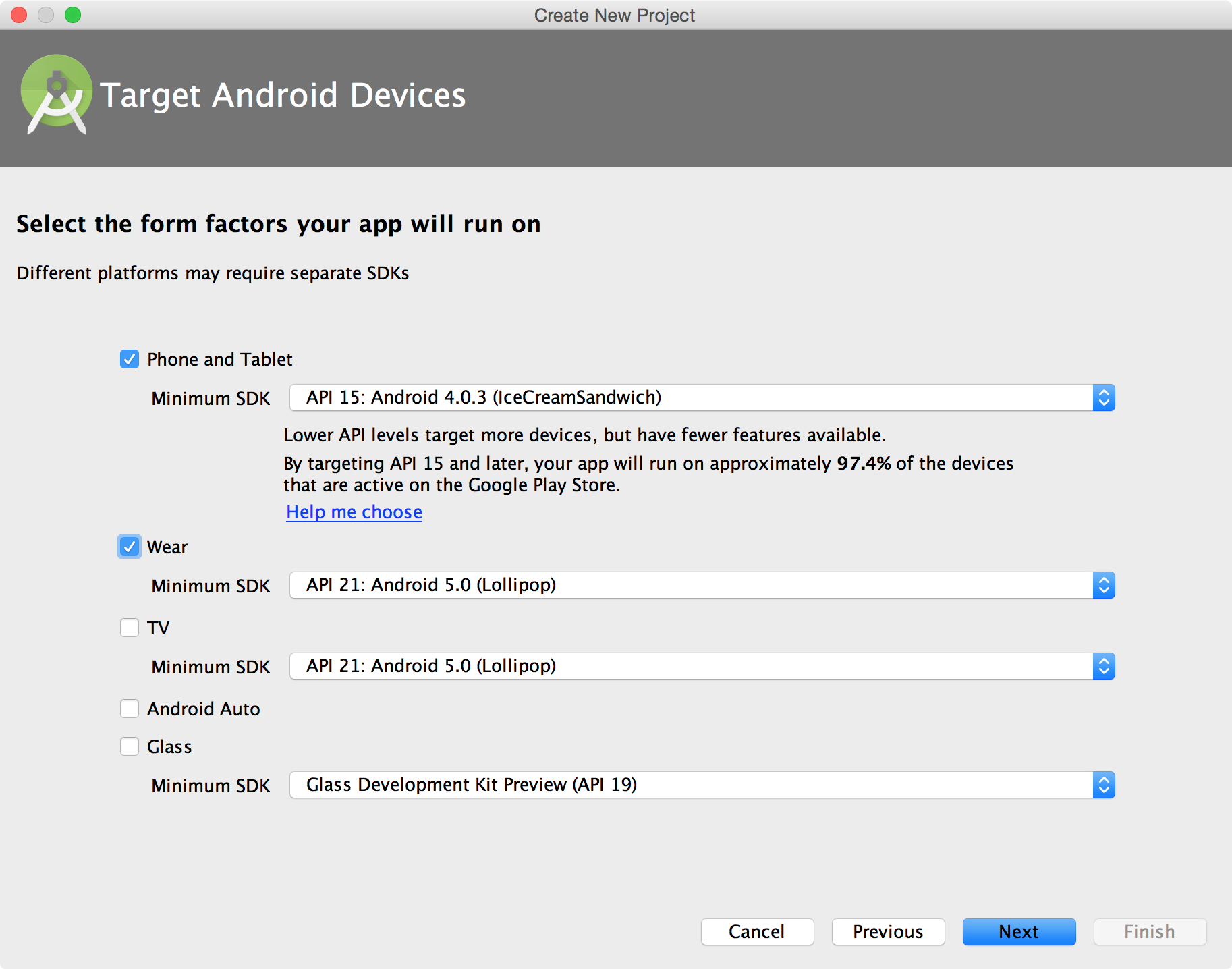
Android Studio makes it easy to create Android apps for various form factors[[1]](#footnote-1), such as phone, tablet, TV, Wear, and Google Glass. The New Project wizard lets you choose the form factors for your app and populates the project structure with everything you need to get started. Use the following steps to create a new project.

The selected form factors become the app modules within the project. For each form factor, you can also select the API Level for that app.



**Figure 2.** Chart of the current Android version distributions, shown when you click **Help me choose**.

The Android Platform Distribution window shows the distribution of mobile devices running each version of Android, as shown in figure 2. Click on an API level to see a list of features introduced in the corresponding version of Android. This helps you choose the minimum API Level that has all the features that your apps needs, so you can reach as many devices as possible. Then click **OK**.

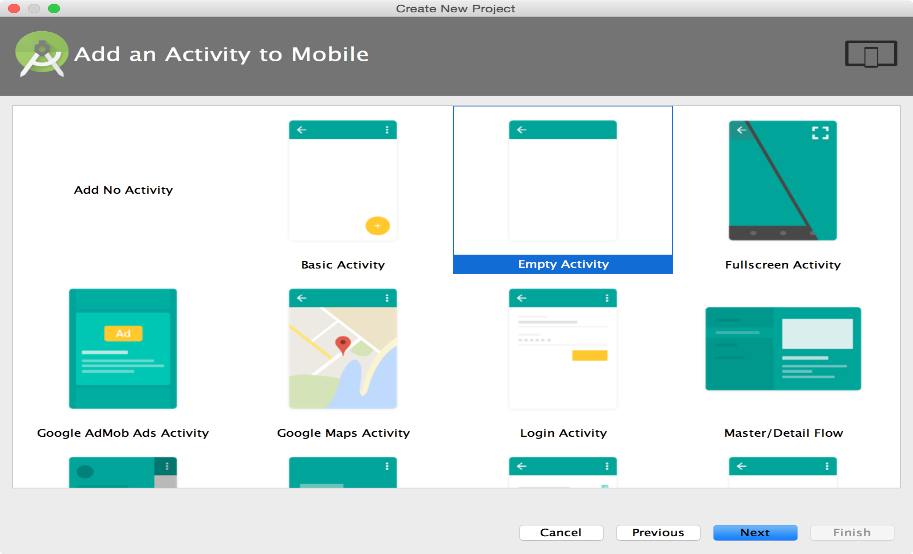


**Figure 3.** The **Target Android Devices** screen.

Then, on the Target Android Devices window, once you've selected your form factors and API versions, click **Next**.

### Step 3: Add an activity

The next screen lets you select an activity type to add to your app, as shown in figure 4. This screen displays a different set of activities for each of the form factors you selected earlier.

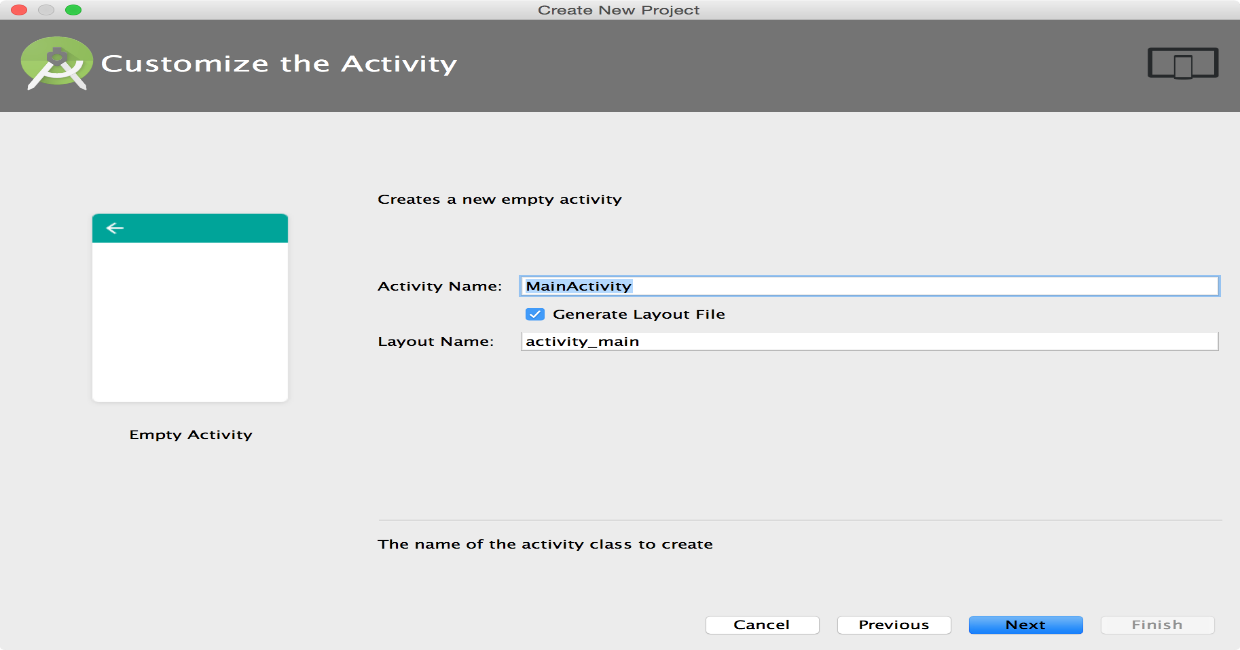


**Figure 4.** The **Add an Activity** screen for a mobile form factor.

For our project choose Empty Activity then click **Next**.

### Step 4: Configure your activity

The next screen lets you configure the activity to add to your app, as shown in figure 5.



**Figure 5.** The **Customize the Activity** screen.

Enter the activity name, the layout name, and the activity title. Then click **Finish**.

### Step 5: Develop your app

First, be sure the **Project** window is open (select **View > Tool Windows > Project**) and the **Android** view is selected from the drop-down list at the top of that window. You can then see the following files:

**app > java > com.example.myfirstapp > MainActivity.java**

**package** . . .;  
  
**import** android.support.v7.app.AppCompatActivity;  
**import** android.os.Bundle;  
  
**public class** MainActivity **extends** AppCompatActivity {  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 }  
}

**Notes:**

* AppCompatActivity

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| java.lang.Object | | | | | | |
| ↳ | android.content.Context | | | | | |
|  | ↳ | android.content.ContextWrapper | | | | |
|  |  | ↳ | android.view.ContextThemeWrapper | | | |
|  |  |  | ↳ | android.app.Activity | | |
|  |  |  |  | ↳ | android.support.v4.app.FragmentActivity | |
|  |  |  |  |  | ↳ | android.support.v7.app.AppCompatActivity |

Base class for activities that use the support library action bar features.

* The savedInstanceState is a reference to a [Bundle](http://developer.android.com/reference/android/os/Bundle.html) object that is passed into the [onCreate](http://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) method of every Android [Activity](http://developer.android.com/reference/android/app/Activity.html). Activities have the ability, under special circumstances, to restore themselves to a previous state using the data stored in this bundle. If there is no available instance data, the savedInstanceState will be null. For example, the savedInstanceState will always be null the first time an Activity is started, but may be non-null if an Activity is destroyed during rotation.
* setContentView(R.layout.***activity\_main***);It means to call the method setContentView, which is in both Activity and Fragment, with the parameter R.layout.activity\_main.

R.layout.activity\_main is an integer in Android R.java, which is is an auto-generated file that contains resource IDs for all the resources of res/ directory defined at build time, that tells the Android framework where to find the layout. Specifically, this would locate a file named main.xml in your res/layouts folder.

**app > res > layout > activity\_main.xml:** This XML file defines the layout for the activity's UI. It contains a TextView element with the text "Hello world!".

*<?***xml version="1.0" encoding="utf-8"***?>*<**RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:id="@+id/activity\_main"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:paddingBottom="@dimen/activity\_vertical\_margin"  
 android:paddingLeft="@dimen/activity\_horizontal\_margin"  
 android:paddingRight="@dimen/activity\_horizontal\_margin"  
 android:paddingTop="@dimen/activity\_vertical\_margin"  
 tools:context="com.example.user.myapplicationtest.MainActivity"**>  
  
 <**TextView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Hello World!"** />  
</**RelativeLayout**>

**Notes**

* [RelativeLayout](https://developer.android.com/reference/android/widget/RelativeLayout.html) is a view group that displays child views in relative positions. The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent [RelativeLayout](https://developer.android.com/reference/android/widget/RelativeLayout.html) area (such as aligned to the bottom, left or center).
* <*. ..* xmlns:android="http://schemas.android.com/apk/res/android"  
      **xmlns:tools="http://schemas.android.com/tools"** > In XML, xmlns declares a Namespace. The namespace has pretty much the same uses as the package name in a Java application.

Android Studio supports a variety of XML attributes in the tools namespace that enable design-time features (such as which layout to show in a fragment) or compile-time. When you build your app, the build tools remove these attributes so there is no effect on your APK size or runtime behavior.

* **android:id="@+id/activity\_main"** Resource ID. A unique resource name for the element, which you can use to obtain a reference to the [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) from your application
* **android:layout\_width="match\_parent"** Sets the dimension to match that of the parent element

which means that the view wants to be as big as its parent container (minus padding) **android:layout\_height="wrap\_parent"** Sets the dimension only to the size required to fit the content of the element

content of this element

* **android:paddingBottom="@dimen/activity\_vertical\_margin"  
  android:paddingLeft="@dimen/activity\_horizontal\_margin"  
  android:paddingRight="@dimen/activity\_horizontal\_margin"  
  android:paddingTop="@dimen/activity\_vertical\_margin"**

These are the dimensions defined in dimens.xml which is used for providing padding, in our case

<**resources**>  
 *<!-- Default screen margins, per the Android Design guidelines. -->* <**dimen name="activity\_horizontal\_margin"**>16dp</**dimen**>  
 <**dimen name="activity\_vertical\_margin"**>16dp</**dimen**>  
</**resources**>

Dp is for - Density independent pixels- See for the Units and Measurements [https://material.io/guidelines/layout/units-measurements.html#](https://material.io/guidelines/layout/units-measurements.html)

**app > manifests > AndroidManifest.xml**

*<?***xml version="1.0" encoding="utf-8"***?>*<**manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="com.example.user.myapplicationtest"**>  
  
 <**application  
 android:allowBackup="true"  
 android:icon="@mipmap/ic\_launcher"  
 android:label="@string/app\_name"  
 android:supportsRtl="true"  
 android:theme="@style/AppTheme"**>  
 <**activity android:name=".MainActivity"**>  
 <**intent-filter**>  
 <**action android:name="android.intent.action.MAIN"** />  
  
 <**category android:name="android.intent.category.LAUNCHER"** />  
 </**intent-filter**>  
 </**activity**>  
 </**application**>  
  
</**manifest**>

The manifest file describes the fundamental characteristics of the app and defines each of its components.

**Notes:**

* **package="com.example.user.myapplicationtest"** The package name of the application
* **android:allowBackup="true"** Whether to allow the application to participate in the backup and restore infrastructure

**android:icon="@mipmap/ic\_launcher"** This attribute must be set as a reference to a drawable resource containing the image (for example "@drawable/icon"). There is no default icon.

* **android:label="@string/app\_name"** The label should be set as a reference to a string resource, so that it can be localized like other strings in the user interface

<**resources**>  
 <**string name="app\_name"**>My ApplicationTest</**string**>  
</**resources**>

**android:supportsRtl="true"** Declares whether your application is willing to support right-to-left (RTL) layouts. If set to true and [targetSdkVersion](https://developer.android.com/guide/topics/manifest/uses-sdk-element.html#target) is set to 17 or higher

* **android:theme="@style/AppTheme"** A reference to a style resource defining a default theme for all activities in the application

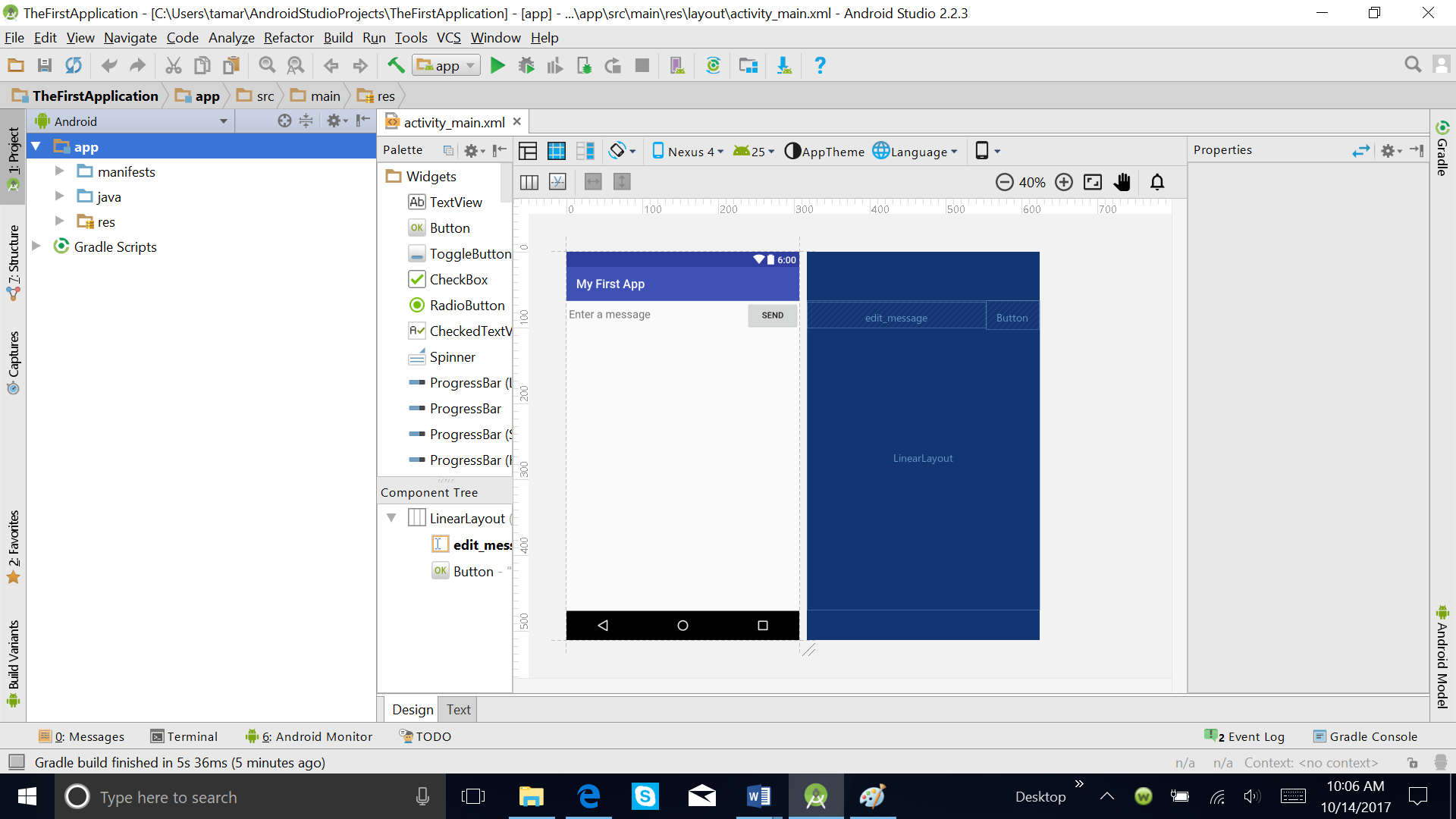
<**resources**>  
  
 *<!-- Base application theme. -->* <**style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar"**>  
 *<!-- Customize your theme here. -->* <**item name="colorPrimary"**>@color/colorPrimary</**item**>  
 <**item name="colorPrimaryDark"**>@color/colorPrimaryDark</**item**>  
 <**item name="colorAccent"**>@color/colorAccent</**item**>  
 </**style**>  
  
</**resources**>

* <**activity android:name=".MainActivity"**> The name if the Main Activity
* <**intent-filter**> Specifies the types of intents that the activity can respond to  
   <**action android:name="android.intent.action.MAIN"** /> android.intent.action.**MAIN** means that this activity is the entry point of the application  
   <**category android:name="android.intent.category.LAUNCHER"** /> Gives additional information about the action to execute. For example, **CATEGORY**\_**LAUNCHER** means it should appear in the **Launcher** as a top-level application. Launcher is the name given to the part of the [Android](http://www.webopedia.com/TERM/A/Android_platform.html) user interface that lets users customize the home screen (e.g. the phone's desktop)  
  </**intent-filter**>

**Gradle Scripts > build.gradle**

Android Studio uses Gradle, an advanced build toolkit, to automate and manage the build process, while allowing you to define flexible custom build configurations

You'll see two files with this name: one for the project and one for the "app" module. Each module has its own build.gradle file, but this project currently has just one module.



**Figure 6.** Project structure for a newly created app.

Now you are ready to develop your app.

## MyFirstApp

This application moves things forward. First it includes two activities. The main activity starts the second activity using an **intent**. When the user inputs a message on the UI of the first activity and clicks SEND, the message is bundled up in an intent and the display activity started -- which displays the message on its own UI. Take a look at the screens below: the first one is the UI rendered by the MainActivity UI, next the user enters a message and clicks the SENX button -- and -- the DisplayMessageActivity is started to display the message in a larger -- font. Note, the menus for each UI is different. You can navigate back using the Up button instead of the back button from the DisplayMessageActivity UI.

|  |  |  |
| --- | --- | --- |
|  |  |  |

This program -- myFirstApp -- is taken from the Android developers with minor modifications.

### The MainActivity's UI

Open the app/java folder and look at the MainActivity.java code.

**Create a linear layout**

The MainActivity's layout file is at **app\src\main\res\layout**

<?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>

* Linear Layout is a view group that aligns all children in a single direction, vertically or horizontally. You can specify the layout direction with the [android:orientation](https://developer.android.com/reference/android/widget/LinearLayout.html#attr_android:orientation) attribute.

**Add a text field**

<EditText android:id="@+id/edit\_message"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:hint="@string/edit\_message" />

**Add String Resources**

By default, your Android project includes a string resource file at src\main\res\values\strings.xml. Add a new string named "edit\_message" and set the value to "Enter a message." (You can delete the "hello\_world" string.)

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

*<?***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\_main\_activity"**>My Main Activity</**string**>  
 <**string name="title\_activity\_display\_message"**>My Message</**string**>  
  
</**resources**>

**Add a button**

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

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

**Format the Input box**

<EditText

android:layout\_width="0dp"  
 android:layout\_weight="1"  
 ... />

* To create a linear layout in which each child uses the same amount of space on the screen, set the [android:layout\_height](https://developer.android.com/reference/android/view/ViewGroup.LayoutParams.html#attr_android:layout_height) of each view to "0dp" (for a vertical layout) or the [android:layout\_width](https://developer.android.com/reference/android/view/ViewGroup.LayoutParams.html#attr_android:layout_width) of each view to "0dp" (for a horizontal layout). Then set the [android:layout\_weight](https://developer.android.com/reference/android/widget/LinearLayout.LayoutParams.html#attr_android:layout_weight) of each view to "1".

**Final XML**

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

<?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>

**Respond to the Send Button**

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

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

android:onClick="sendMessage" />

**Building an Intent**

Open the MainActivity class (located in the project's src/ directory) and add the corresponding method:

**package** com.example.thefirstapplication;  
  
**import** android.app.Activity;  
**import** android.content.Intent;  
**import** android.os.Bundle;  
**import** android.util.Log;  
**import** android.view.View;  
**import** android.widget.EditText;  
  
  
**public class** MainActivity **extends** Activity {  
  
 **private static final** String ***TAG*** = **"MyFirstApp"**;  
  
 **public final static** String ***EXTRA\_MESSAGE*** = **"com.example.myfirstapp.MESSAGE"**;  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
 }  
 */\*\* Called when the user clicks the Send button \*/* **public void** sendMessage(View view) {  
  
  }  
}

**Starting another 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.

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

.....

**public void** sendMessage(View view) {  
  
 *// Send the input string to the DisplayMessageActivity using an intent* Log.*d*(***TAG***, **"sendMessage"**);  
  
 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);  
 }

}

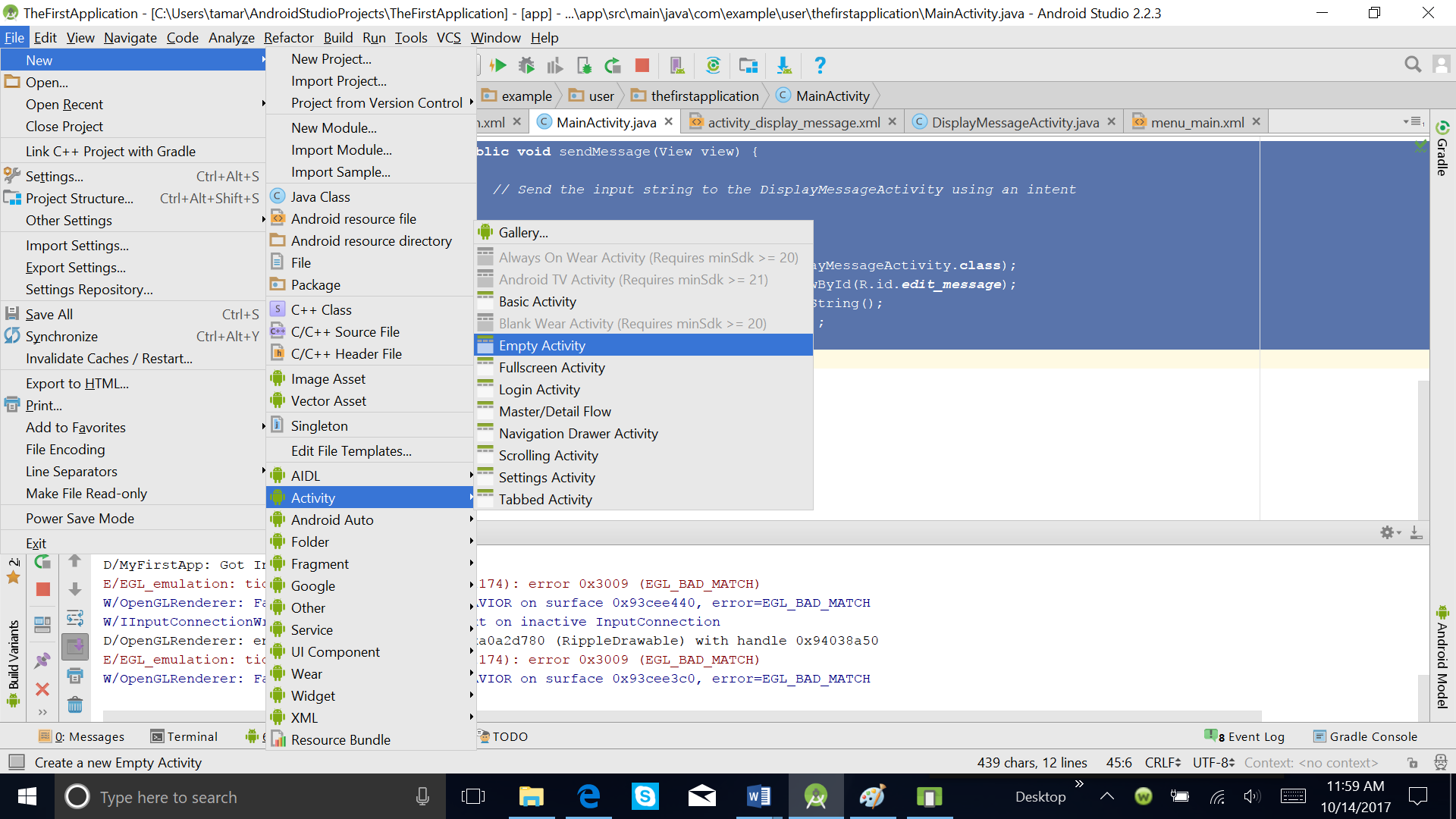
...

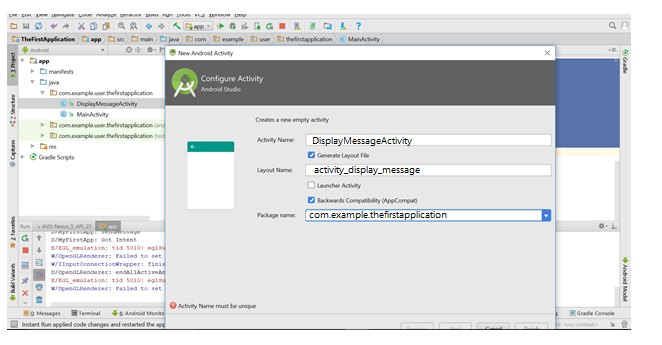
Now you need to create the DisplayMessageActivity class in order for this to work.

**Create another activity -- DisplayMessageActivity**

To create a new activity using Android Studio:

Right click **app**, find **New->Activity->Blank Activity** and click to open the create activity dialog. Fill in the activity details:





Finally click *Finish*.

The DisplayMessageActivity class should now look like this:

**package** . . .;  
  
**import** android.app.Activity;  
**import** android.content.Intent;  
**import** android.os.Bundle;  
**import** android.util.Log;  
**import** android.widget.TextView;  
  
**public class** DisplayMessageActivity **extends** Activity {  
  
 **private static final** String ***TAG*** = **"MyFirstApp"**;  
  
 @Override  
 **public void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);

setContentView(R.layout.***activity\_display\_message***);

}  
}

....

**Add the title string**

<resources>

...

<string name="title\_activity\_display\_message">My Message</string>

</resources>

This is included in the Manifest where the DisplayMessageActivity is specified.

**Add it to the manifest**

Take a look at the XML for the new activity .DisplayMessageActivity in the manifest. The code is self explanatory.

***<?*xml version="1.0" encoding="utf-8"*?>*<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package=". . .">  
  
 <application  
 android:allowBackup="true"  
 android:icon="@mipmap/ic\_launcher"  
 android:label="@string/app\_name"  
 android:supportsRtl="true"  
 android:theme="@style/AppTheme">  
 <activity android:name=".MainActivity">  
 <intent-filter>  
 <action android:name="android.intent.action.MAIN" />  
  
 <category android:name="android.intent.category.LAUNCHER" />  
 </intent-filter>  
 </activity>  
  
 <activity  
 android:name=".DisplayMessageActivity"  
 android:label="@string/title\_activity\_display\_message"  
 android:parentActivityName=".MainActivity" >  
 <meta-data  
 android:name="android.support.PARENT\_ACTIVITY"  
 android:value=".MainActivity" />  
 </activity>  
 </application>  
  
 </manifest>**

**Running the app**

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.

|  |  |  |
| --- | --- | --- |
|  |  |  |

**Receive the Intent**

In the DisplayMessageActivity class’s onCreate() method, get the intent and extract the message delivered by MainActivity:

import android.app.Activity;

import android.content.Intent;

import android.os.Bundle;

import android.support.v4.app.NavUtils;

import android.view.MenuItem;

public class DisplayMessageActivity extends Activity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_display\_message);

// Get the message from the intent

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().

The complete onCreate() method for DisplayMessageActivity now looks like this:

import android.app.Activity;

import android.content.Intent;

import android.os.Bundle;

import android.support.v4.app.NavUtils;

import android.view.MenuItem;

import android.widget.TextView;

public class DisplayMessageActivity extends Activity {

@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 are done, now run your Android firstapp!**

|  |  |
| --- | --- |
|  |  |

You should see the above screens in Android emulator. But to see the Toolbars in both Activities, the activity extends AppCompatActivity.

https://developer.android.com/training/appbar/setting-up.html

**Application Logging using Log.d()**

You can put print or log statements your code. You might want to confirm for example that an intent fired by one activity is received by another. You can use Log.d() to do this. You have to first create a TAG in your code and then call Log.d() with the tag.

In the code snippet below that you need to import android.util.Log, set up the TAG, and then call Log.d()

package edu.dartmouth.cs.myfirstapp;

import android.app.Activity;

import android.content.Intent;

import android.os.Bundle;

import android.support.v4.app.NavUtils;

import android.util.Log;

import android.view.MenuItem;

import android.widget.TextView;

public class DisplayMessageActivity extends AppCompatActivity {

private static final String TAG = "MyFirstApp";

@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);

Log.d(TAG, "Got Intent");

// Create the text view

TextView textView = new TextView(this);

textView.setTextSize(40);

textView.setText(message);

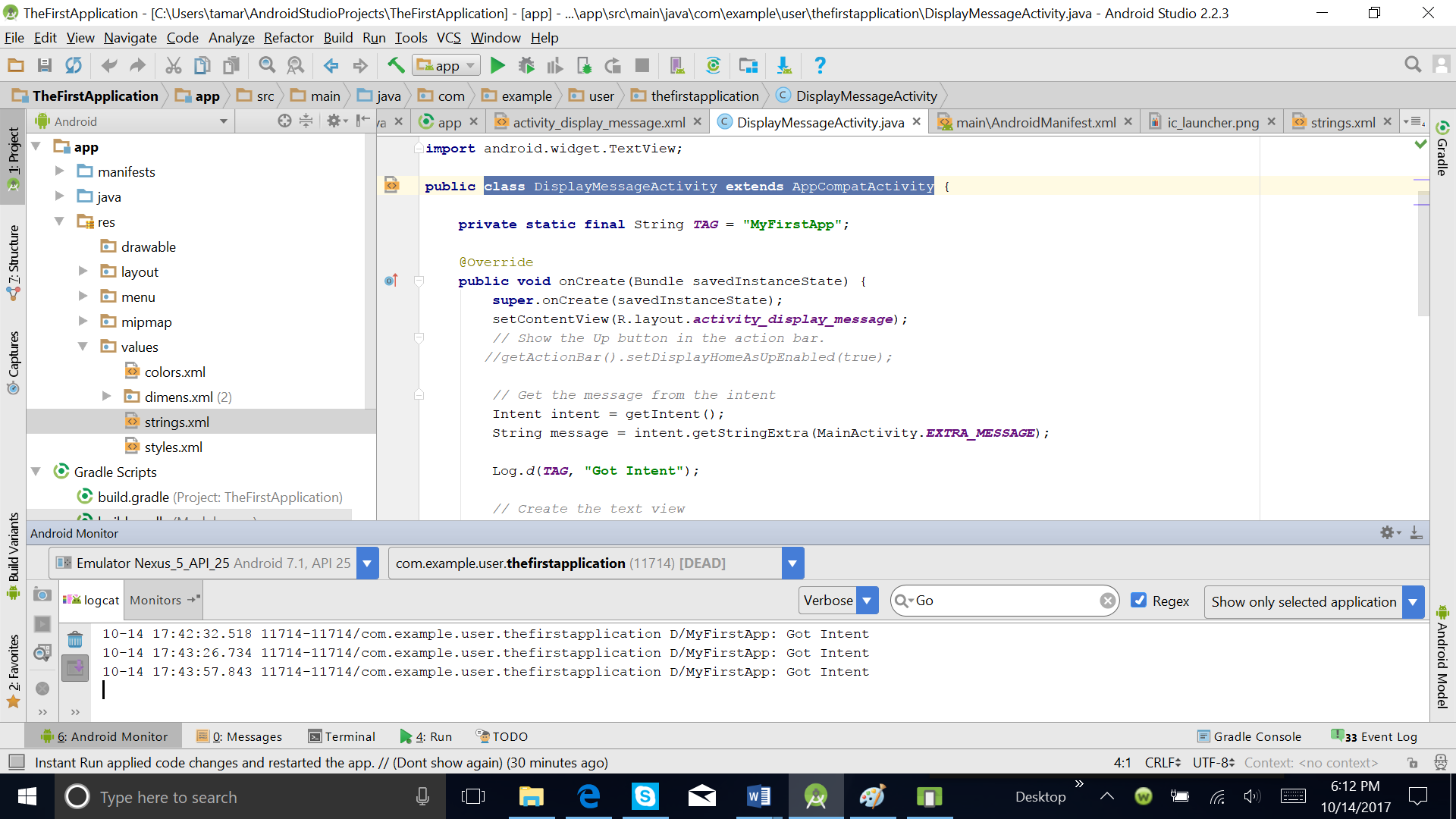
// Set the text view as the activity layout

setContentView(textView);

}

......

**Using LogCat to view application output**

An easy way to monitor the output from log.d is to use the LogCat utility. LogCat is integrated into Android Studio. You’ll find the LogCat by clicking the **Android** tab at the bottom of Android Studio. The following screenshot shows the code to print logs in LogCat and viewing logs from the LogCat window.

Once you click on run and enter input you will see to logs printed out from the application: one when the user clicks send at the UI; and one when DisplayMessageActivity receives the *Intent* and prints the message, as shown above.

## References

1. https://developer.android.com/studio/write/tool-attributes.html
2. <https://developer.android.com/guide/topics/resources/layout-resource.html>
3. [https://material.io/guidelines/layout/units-measurements.html#](https://material.io/guidelines/layout/units-measurements.html)
4. <https://content.pivotal.io/blog/android-savedinstancestate-bundle-faq>

## Assignment

Create a project where an Activity calls another Activity. In the first activity you enter some data, and on the other activity you do some calculation with the data that we get from the first activity and display all the information.

* First you finish the project of the class in your computer.
* Make a copy of this project, and create a new one with the name **FirstApplicationAssignment.** For this you have to follow the following steps

1. Using Windows Explorer, navigate to the Android Studio projects directory select the project folder to be cloned;
2. Copy the selected folder;
3. Paste the folder into the Android Studio (AS) projects directory;
4. Rename the pasted folder to be the new name of the cloned project, say 'ViewPager\_Shell';
5. Launch Android Studio;
6. Select 'Open an existing Android Studio project';
7. Navigate to the AS projects directory;
8. Select the folder 'ViewPager\_Shell';
9. Ensure that AS's explorer pane is visible on the left-hand-side of the AS main frame;
10. Select the 'Android' view for the explorer pane;
11. Open app -> java; the old package name should now be visible, e.g. 'com.hulme.owm';
12. Right-click on the old package name and select 'Refactor -> Rename';
13. A dialog should now appear; left-click 'Rename Package';
14. In the pop-up dialog, replace the old package (e.g. 'owm') with the new package name (e.g. viewpager\_shell);
15. Left-click on the 'Refactor' button; BEWARE!! ... A new 'Find Factoring Preview' pane should open in the lower-left of the AS main frame window;
16. Select 'Do Refactor';
17. Using the AS explorer pane, navigate to 'Gradle Scripts -> build.gradle(Module:app);
18. Double-click build.gradle(Module:app) to view its contents in the AS editor pane;
19. Change the defaultConfig 'applicationID' string to your new full package path (e.g. "com.hulme.viewpager\_shell");
20. Using the AS explorer pane, navigate to 'app -> manifests -> AndroidManifest.xml';
21. Double-click on 'AndroidManifest.xml' to view it in the AS editor pane;
22. Under the **manifest** tag, you should observe that the new package name (e.g. 'com.hulme.viewpager\_shell') now appears, as a result of the previous refactoring step;
23. Under **application** tag, change android:label to a new name (e.g. "ViewPager Shell Demo") - this is the name that appears in the list of installed apps provided by Android's 'Settings -> Application manager;
24. Under **activity** tag, change android:label to a new name (e.g. "VP Shell") - this is the name that appears under the app's icon and it is also the activity's default ActionBar title;

You should now be able to build and run the newly cloned project.

<https://stackoverflow.com/questions/29045964/android-studio-how-to-copy-a-project>

* Update now this project to fulfill the request.

1. The **form factor** of a mobile phone is its size, shape, and style, as well as the layout and position of its major components. There are three major form factors – bar phones, flip phones, and sliders – as well as sub-categories of these forms and some atypical forms. [↑](#footnote-ref-1)