

# Android Developer Fundamentals

*Learn to develop Android applications*

University Curriculum, Draft 1

Developed by Google Developer Training Team

*August 2016*



# Table of Contents

<a href="#">Introduction</a>	1.1
<a href="#">Unit 1. Get started</a>	1.2
<a href="#">    Lesson 1: Build your first app</a>	1.2.1
<a href="#">        1.1 P: Install Android Studio, Hello World, Logging</a>	1.2.1.1
<a href="#">        1.2 P: Make Your First Interactive UI</a>	1.2.1.2
<a href="#">        1.3 P: Working with TextView Elements</a>	1.2.1.3
<a href="#">        1.4 P: Learning Resources</a>	1.2.1.4
<a href="#">    Lesson 2: Activities</a>	1.2.2
<a href="#">        2.1 P: Create and Start Activities</a>	1.2.2.1
<a href="#">        2.2 P: Activity Lifecycle and State</a>	1.2.2.2
<a href="#">        2.3 P: Activities and Implicit Intents</a>	1.2.2.3
<a href="#">    Lesson 3: Testing, debugging, and using support libraries</a>	1.2.3
<a href="#">        3.1 P: Using the Debugger</a>	1.2.3.1
<a href="#">        3.2 P: Testing your App</a>	1.2.3.2
<a href="#">        3.3 P: Using Support Libraries</a>	1.2.3.3
<a href="#">Unit 2. User experience</a>	1.3
<a href="#">    Lesson 4: User interaction</a>	1.3.1
<a href="#">        4.1 P: Use Keyboards, Input Controls, Alerts, and Pickers</a>	1.3.1.1
<a href="#">        4.2 P: Use an Options Menu and Radio Buttons</a>	1.3.1.2
<a href="#">        4.3 PC: Tab Navigation</a>	1.3.1.3
<a href="#">        4.4 P: Create a Recycler View</a>	1.3.1.4
<a href="#">    Lesson 5: Delightful user experience</a>	1.3.2
<a href="#">        5.1 P: Themes, Custom Styles, Drawables</a>	1.3.2.1
<a href="#">        5.2 P: Material Design: Cards and the FAB</a>	1.3.2.2
<a href="#">        5.3 PC: Transitions and Animations</a>	1.3.2.3
<a href="#">    Lesson 6: Testing your UI</a>	1.3.3
<a href="#">        6.1 P: Use Espresso to test your UI</a>	1.3.3.1
<a href="#">Unit 3. Working in the background</a>	1.4
<a href="#">    Lesson 7: Background Tasks</a>	1.4.1
<a href="#">        7.1 P: Create an AsyncTask</a>	1.4.1.1

<a href="#">7.2 P: Broadcast Receivers</a>	1.4.1.2
<a href="#">7.3 P: Notifications</a>	1.4.1.3
<a href="#">Lesson 8: Triggering, scheduling and optimizing background tasks</a>	1.4.2
<a href="#">8.1 P: Alarm Manager</a>	1.4.2.1
<a href="#">8.2 P: Job Scheduler</a>	1.4.2.2
<a href="#">Lesson 9: Connect to the Internet</a>	1.4.3
<a href="#">9.1 P: Connect to the Internet using an AsyncTask</a>	1.4.3.1
<a href="#">9.2 P: Using an AsyncTaskLoader</a>	1.4.3.2
<a href="#">Unit 4. All about data</a>	1.5
<a href="#">Lesson 10: Saving, retrieving, loading data</a>	1.5.1
<a href="#">10.0 P: Shared Preferences</a>	1.5.1.1
<a href="#">Lesson 11: Storing data using SQLite</a>	1.5.2
<a href="#">11.1 P: SQLite Data Storage</a>	1.5.2.1
<a href="#">11.2 P: Searching an SQLite Database</a>	1.5.2.2
<a href="#">Lesson 12: Sharing data with content providers</a>	1.5.3
<a href="#">12.1 P: Implement a Minimalist Content Provider</a>	1.5.3.1
<a href="#">12.2 P: Add a Content Provider to WordListSQL</a>	1.5.3.2
<a href="#">12.3 P: Sharing content with other apps</a>	1.5.3.3
<a href="#">Lesson 13: Loading data using loaders</a>	1.5.4
<a href="#">13.1 P: Load and display data fetched from a content provider</a>	1.5.4.1
<a href="#">Appendix Utilities</a>	1.6

# Android Developer Fundamentals

**Learn to develop Android applications**

**University Curriculum, Beta 1**

**IMPORTANT This is only a draft! It is subject to change.**

***Developed by Google Developer Training Team***

*September 2016*



*This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License*



# 1.1 P: Install Android Studio and Run Hello World

## Contents:

- What you should already KNOW
- What you will NEED
- What you will LEARN
- What you will DO
- App overview
- Task 1. Install Android Studio
- Task 2: Create “Hello World” app
- Task 3: Explore the project structure and layout
- Task 4: Create a virtual device
- Task 5: Run your app on an emulator
- Task 6. Add log statements to your app
- Task 7: Explore the AndroidManifest.xml file
- Task 8. Explore the build.gradle file
- Task 9. Run your app on a device
- Coding challenge
- Summary
- Resources

Welcome to the practical exercises. In this series of exercises you will:

- Install Android Studio, the Android development environment.
- Learn about the Android development process.
- Create and run your first Android Hello World app on an emulator and on a physical device.
- Add logging to your app for debugging.

## What you should already KNOW

For this practical you should be familiar with:

- The general software development process for object-oriented applications using an IDE (Integrated Development Environment).
- At least 1-3 years of experience with object-oriented programming and the Java programming language. (These practicals will not explain object-oriented programming)

or the Java language.)

## What you will NEED

- A Mac, Windows, or Linux computer. See the bottom of the [Android Studio download page](#) for up-to-date [system requirements](#).
- Internet access or an alternative way of loading the latest Android Studio and Java installations onto your computer.

## What you will LEARN

- How to install and use the Android IDE.
- The development process for building Android apps.
- How to create an Android project from a basic app template.

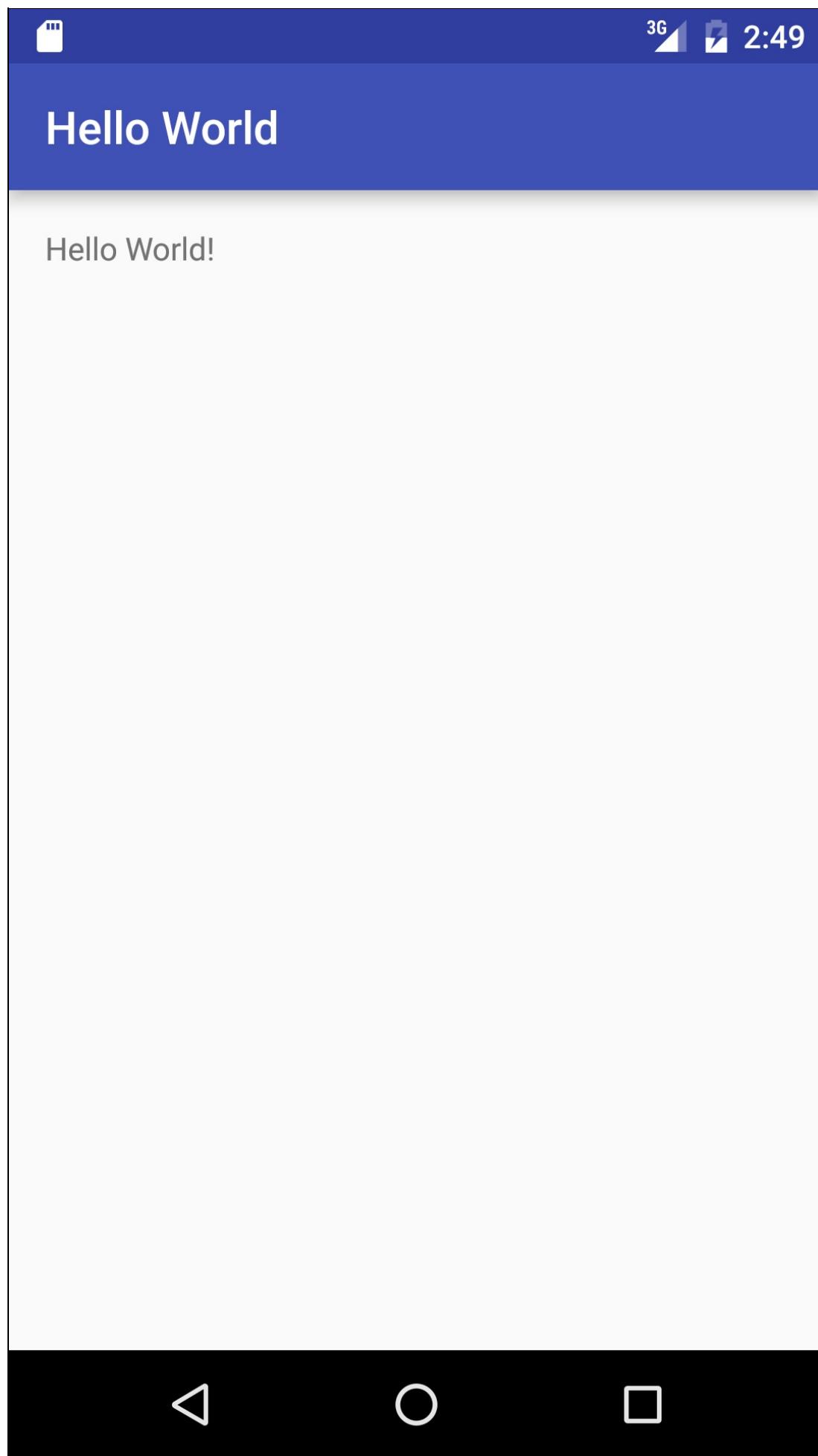
## What you will DO

- Install the Android Studio development environment.
- Create a an emulator (virtual device) to run your app on your computer.
- Create and run the Hello World app on the virtual and physical devices.
- Explore the project layout.
- Generate and view log statements from your app.
- Explore the AndroidManifest.xml file.

## App Overview

After you successfully install the Android Studio IDE, you will create a new Android project for the ‘Hello World’ app from a template. This simple app displays the string “Hello World” on the screen of the Android virtual or physical device. Here’s what it should look like once you successfully build it.”

Here’s what the finished app will look like:



# Task 1. Install Android Studio

Android Studio is Google's IDE for Android apps. Android Studio gives you an intelligent code editor and a set of app templates. In addition, it contains tools for development, debugging, testing, and performance that make it faster and easier to develop apps. You can test your apps with a large range of preconfigured emulators or on your own mobile device, and build production APKs for publication.

**Note:** Android Studio is continually being improved. For the latest information on system requirements and installation instructions, refer to the documentation at [developer.android.com](https://developer.android.com).

To get up and running with Android Studio:

- You may need to install the Java Development Kit - Java 7 or better.
- Install Android Studio

Android Studio is available for Windows, Mac, and Linux computers. The installation is similar for all platforms. Any differences will be noted.

## 1.1. Installing the Java Development Kit

1. On your computer, open a terminal window.
2. Type `java -version`

The output includes a line:

```
Java(TM) SE Runtime Environment (build1.X.0_05-b13)
```

X is the version number to look at.

- If this is 7 or greater, you can move on to installing Android Studio.
- If you see a Java SE version below 7 or if Java is not installed, you need to install the latest version of the Java SE development kit.

To download the Java Standard Edition () Development Kit (JDK):

1. Go to the [Oracle Java SE downloads page](#).
  2. Click the Java SE Downloads icon to open the [Java SE Development Kit 8 Downloads page](#).
  3. In the box for the latest Java SE Development kit, you need to accept the License Agreement to proceed. Then download the version appropriate for your computer.
- Important:** Do not go to the demos and samples (the menus look very similar, so make

- sure to read the heading at the top).
4. Install the development kit. Once the installation of the JDK is completed—it should only take a few minutes—you can confirm it's correct by checking the Java version from the command line.
  5. Open a terminal window and type `java -version` again to verify that installation has been successful.
  6. Set the `JAVA_HOME` environment variable to the installation directory of the JDK.

### Windows:

1. Set `JAVA_HOME` to the installation location.
  2. Start > Control Panel > System > Advanced System Settings > Environment Variables  
System Variables > New...
- Variable name: `JAVA_HOME`
- Variable value: `C:\Program Files\Java\jdk1.7.0_80` (or whatever your installation is!)
3. If the variable already exists, update it to this version of the JDK.
  4. Verify your `JAVA_HOME` variable from a cmd.exe terminal: `echo %JAVA_HOME%`

See also: [https://docs.oracle.com/cd/E19182-01/820-7851/inst\\_cli\\_jdk\\_javahome\\_t/](https://docs.oracle.com/cd/E19182-01/820-7851/inst_cli_jdk_javahome_t/)

### Mac:

1. Open Terminal.
2. Confirm you have JDK by typing “which java”. ...
3. Check you have the needed version of Java, by typing “`java -version`”.
4. Set `JAVA_HOME` using this command in Terminal: `export JAVA_HOME=$(/usr/libexec/java_home)`
5. `echo $JAVA_HOME` on Terminal to confirm the path.

### Linux:

See: [https://docs.oracle.com/cd/E19182-01/820-7851/inst\\_cli\\_jdk\\_javahome\\_t/](https://docs.oracle.com/cd/E19182-01/820-7851/inst_cli_jdk_javahome_t/)

**Important:** Do not move on with Android Studio install until after you have installed the JDK. Without a working copy of Java, the rest of the process will not work. If you can't get the download to work, look for error messages, and search online to find a solution.

#### Basic Troubleshooting:

- There is no UI, Control Panel, or Startup icon associated with the JDK.
- Verify that you have indeed installed the JDK by going to the directory where you installed it. If you don't know where that is, look at your PATH variable and/or search your computer for the "jdk" directory or the "java" or "javac" executable.

## 1.2. Installing Android Studio

1. Navigate to the [Android developers site](#) and follow the instructions to download and install Android Studio.
    - Accept the default configurations for all steps.
    - Make sure that all components are selected for installation.
  2. After finishing the install, the Setup Wizard will download and install some additional components. Be patient, this might take some time depending on your Internet speed.
  3. When you are finished, Android Studio will start, and you are ready to create your first project.
- Troubleshooting:** If you run into problems with your installation, check the latest documentation, programming forums, or get help from your instructors.

## Task 2: Create “Hello World” app

In this task, you will create and configure a project for the “Hello World” app.

**Why:** This task confirms your installation is correct, and familiarizes you with the Android Studio workflow.

### 2.1 Create the “Hello World” app

1. Start Android Studio if it is not already running.
2. In the main **Welcome to Android Studio** window, click “Start a new Android Studio project”.
3. In the **New Project** window, give your application an **Application Name**, such as Hello World.
4. Verify the Project location, or choose a different directory for storing your project.
5. Choose a unique **Company Domain**.
  - Apps published to the Google Play Store must have a unique package name. Since domains are unique, prepending your app's name with your or your company's domain name is going to result in a unique package name.
  - If you are not planning to publish your app, you can accept the default example domain. Be aware that changing the package name of your app later is extra work.
6. Verify that the default **Project location** is where you want to store your Hello World app and other Android Studio projects, or change it to your preferred directory. Click Next.
7. On the **Target Android Devices** screen, “Phone and Tablet” should be selected. And you should ensure that API 15: Android 4.0.3 IceCreamSandwich is set as the **Minimum SDK**. (Fix this if necessary.)
  - At the writing of this book, choosing this API level makes your “Hello World” app

app compatible with 97% of Android devices active on the Google Play Store.

- These are the settings used by the examples in this book.

8. Click **Next**.
9. If your project requires additional components for your chosen target SDK, Android Studio will install them automatically. Click **Next**.
10. **Customize the Activity** window. Every app needs at least one activity. An activity represents a single screen with a user interface and Android Studio provides templates to help you get started. For the Hello World project, choose the simplest template (as of this writing, the "Empty Activity" project template is the simplest template) available.
11. It is a common practice to call your main activity `MainActivity`. This is not a requirement.
12. Make sure the **Generate Layout file** box is checked (if visible).
13. Make sure the **Backwards Compatibility (App Compat)** box is checked.
14. Leave the **Layout Name** as `activity_main`. It is customary to name layouts after the activity they belong to. Accept the defaults and click **Finish**.

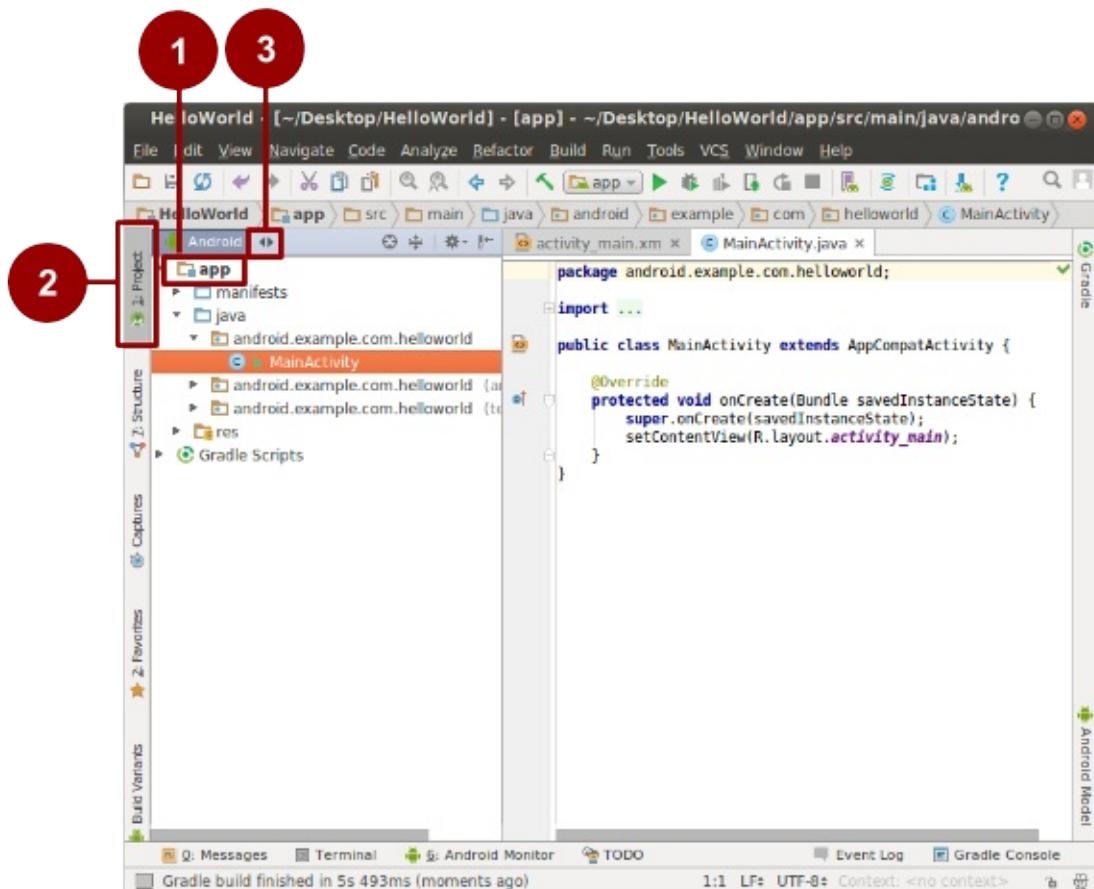
After these steps, Android Studio:

- Creates a folder for your Android Studio Projects.
- Builds your project with [Gradle](#) (this may take a few moments). Android Studio uses Gradle as its build system. See the [Configure your build](#) developer page for more information.
- Opens the code editor with your project.
- Displays a tip of the day. Android Studio offers many keyboard shortcuts, and reading the tips is a great way to learn them over time.

Your Android Studio window should look similar to the following diagram. You can look at the hierarchy of the files of your app in multiple ways.

1. Click on the Hello World folder to expand the hierarchy of files (1),
2. Click on **Project** (2).
3. Click on the **Android** menu (3).

4. Explore the different view options for your project.



**Note:** This book uses the **Android** view of the project files, unless specified otherwise.

## Task 3: Explore the project structure

In this practical, you will explore how the project files are organized in Android Studio.

**Why:** Being familiar with the project structure makes it easier to find and change files for later exercises.

These steps assume that your Hello World project starts out as shown in the diagram above.

### 3.1 Explore the project structure and layout

In the Project/Android view (see previous task), there are three top-level folders below your **app** folder: **manifests**, **java**, and **res**.

1. Expand the **manifests** folder.

This folder contains **AndroidManifest.xml**. This file describes all the components of your Android app and is read by the Android run-time system when your program is executed .

2. Expand the **java** folder. All your Java language files are organized in this folder. The **java** folder contains three subfolders:
  - **com.example.hello.helloworld (or whatever your domain was called)**: All the files for a package are in a folder named after the package. For your Hello World application, there is one package and it only contains MainActivity.java (the file extension may be omitted in the Project view).
  - **com.example.hello.helloworld(androidTest)**: This folder is for your instrumented tests and starts out with a skeleton ApplicationTest.java file.
  - **com.example.hello.helloworld(test)**: This folder is for your unit tests and starts out with an automatically created skeleton unit test file.
3. Expand the **res** folder. This folder contains all the resources for your app, including images, layout files, strings, icons, and styling. It includes these subfolders:
  - **drawable**. Store all your app's images in this folder.
  - **layout**. Every activity has at least one layout file that describes the UI in XML. For Hello World, this folder contains activity\_main.xml.
  - **mipmap**. Store your icons in this folder. There is a sub-folder for each supported screen density. Android uses the screen density, that is, the number of pixels per inch to determine the required image resolution. Android groups all actual screen densities into generalized densities, such as medium (mdpi), high (hdpi), or extra-extra-high (xxxhdpi). The ic\_launcher.png folder contains the default launcher icons for all the densities supported by your app.
  - **values**. Instead of hardcoding values into your XML files, it is best practice to define them in their respective values file. This makes it easier to change and be consistent across your app.
4. Expand the **values** subfolder within the res folder. It includes these subfolders:
  - **colors.xml**. Shows the default colors for your theme, and you can add your own colors or change them.
  - **dimens.xml**. Store sizes of views and objects for different resolutions.
  - **strings.xml**. Create resources for all your strings. This makes it easy to translate them to other languages.
  - **styles.xml**. All the CSS for your app goes here. Styles help you have a consistent look for all UI elements in your app.

## 3.2 The Gradle build system

Android Studio uses Gradle as its build system. You will learn about gradle what you need to build and run your apps as you progress.

1. Expand the **Gradle Scripts** folder. This folder contains all the files needed by the build system.
2. Look for the **build.gradle(Module:app)** file. When you are adding app-specific

dependencies, such as using additional libraries, they go into this file.

## Task 4: Create a virtual device (emulator)

In this task, you will use the [Android Virtual Device \(AVD\) manager](#) to create a virtual device or emulator that simulates the configuration for a particular type of Android device.

Using the AVD Manager, you define the hardware characteristics of a device and its API level, and save it as a virtual device configuration.

When you start the Android emulator, it reads a specified configuration and creates an emulated device on your computer that behaves exactly like a physical version of that device.

**Why:** With virtual devices, you can test your app on different devices (tablets, phones) with different API levels to make sure it looks good and works for most users. You do not depend on having a physical device available for app development.

### 4.1 Create a virtual device

In order to run an emulator on your computer, you have to create a configuration that describes the virtual device.

1. In Android Studio, select Tools > Android > AVD Manager, or click the AVD Manager icon  in the toolbar.
2. Click the +Create Virtual Device.... (If you have created a virtual device before, the window shows all your existing devices and the button is at the bottom.)

The Select Hardware screen appears showing a list of preconfigured hardware devices.

For each device, the table shows its diagonal display size (Size), screen resolution in pixels (Resolution), and pixel density (Density).

For the Nexus 5 device, the pixel density is xxhdpi, which means your app uses the icons in the xxhdpi folder of the mipmap folder. Likewise, your app will use layouts and drawables from folders defined for that density as well.

3. Choose the Nexus 5 hardware device and click **Next**.
4. On the **System Image** screen, from the **Recommended** tab, choose which version of the Android system to run on the virtual device. You can select the latest system image which as of July 2016 is Marshmallow 23 x86\_64 Android 6.0.

There are many more versions available than shown in the **Recommended** tab. Look at the **x86 Images** and **Other Images** tabs to see them.

5. If a **Download** link is visible next to a system image version, it is not installed yet, and you need to download. If necessary, click the link to start the download, and **Finish** when it's done.
6. On **System Image** screen, choose a system image and click **Next**.
7. Verify your configuration, and click **Finish**. (If the **Your Android Devices** AVD Manager window stays open, you can close it.)

## Task 5. Run your app on an emulator

In this task, you will finally run your Hello World app.

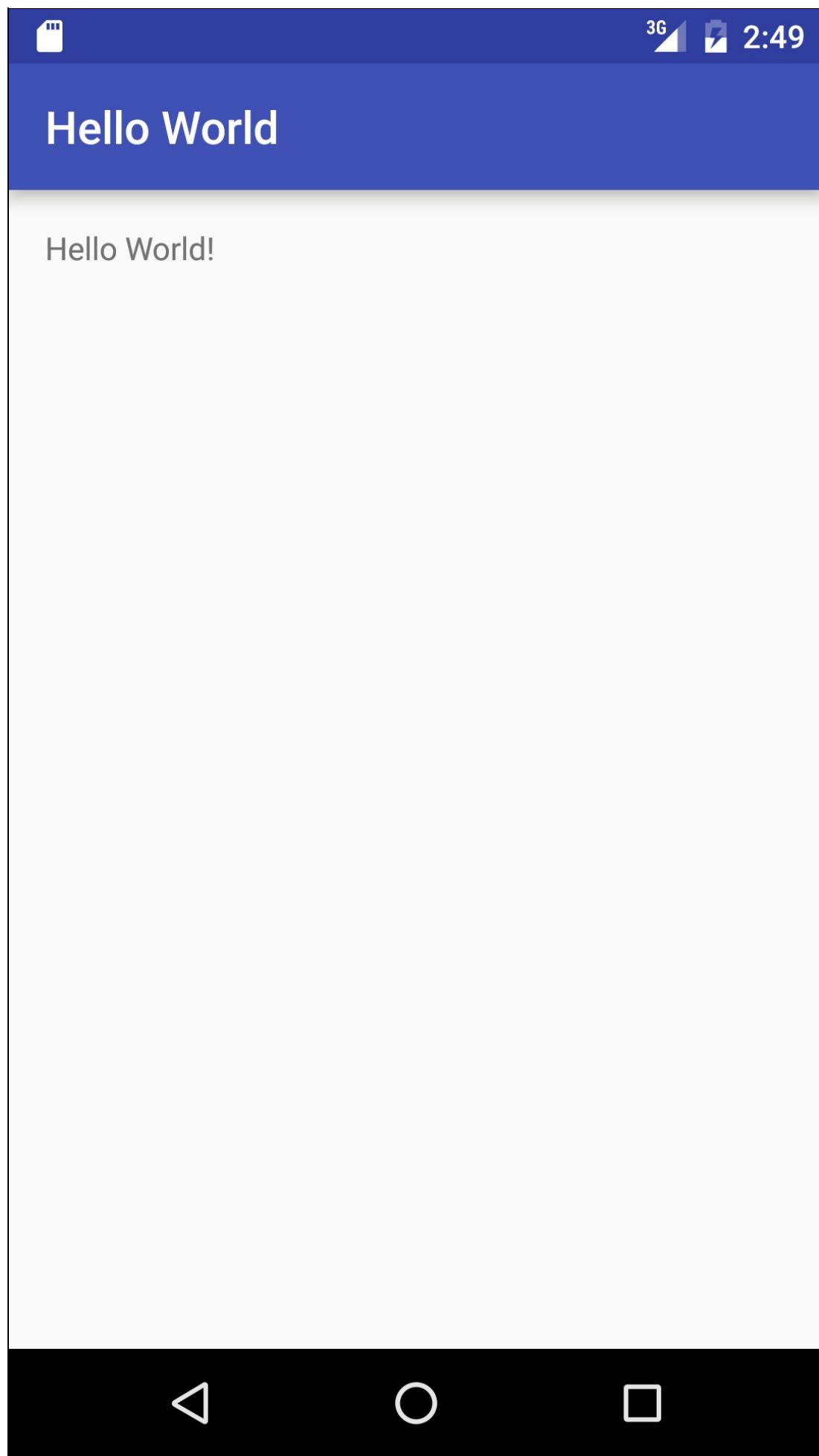
**Why:** Goal achieved!

### 5.1 Run your app on an emulator

1. In Android Studio, select **Run > Run app** or click the **Run icon**  in the toolbar.
2. In the **Select Deployment Target** window, under **Available Emulators**, select **Nexus 5 API 23** and click **OK**.

The emulator starts and boots just like a physical device. Depending on the speed of your computer, this may take a while. Your app builds, and once the emulator is ready, Android Studio will upload the app to the emulator and run it.

You should see the Hello World app as shown in the following screenshot.



**Note:** If you are testing on an emulator, it is good practice to start it up once at the very beginning of your session, and not to close it until you are done so that it doesn't have to go through the boot process again.

**Challenge:** You can fully customize your virtual devices.

- Study the [AVD Manager documentation](#).
- Create one or several custom virtual devices.

You may notice that not all combinations of devices and system versions work when you run your app. This is because not all system images can run on all hardware devices.

## Task 6. Add log statements to your app

In this practical, you will add log statements to your app, which are displayed in the logging window of the Android Monitor.

**Why:** Log messages are a powerful debugging tool that you can use to check on values, execution paths, and report exceptions.

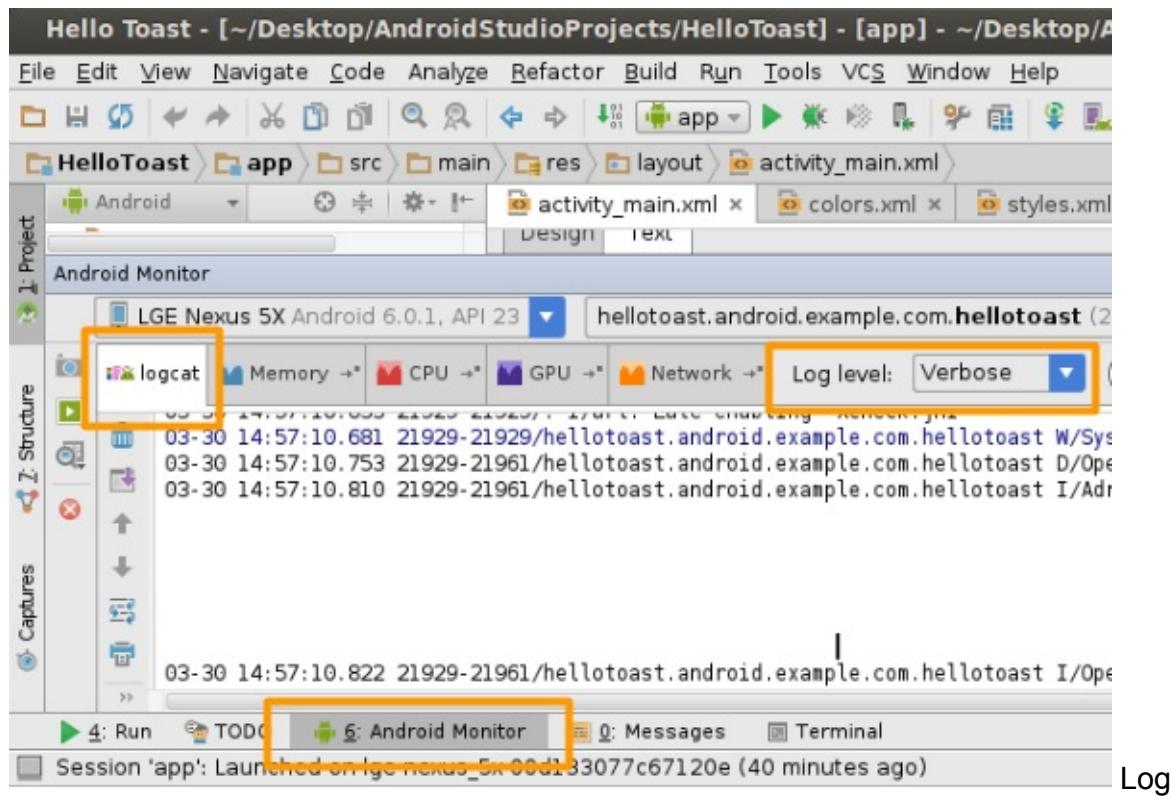
The **Android Monitor** displays information about your app.

1. Click the **Android Monitor** button at the bottom of Android Studio to open the Android Monitor.

By default, this opens to the **logcat** tab, which displays information about your app as it is running. If you add log statements to your app, they are printed here as well.

You can also monitor the Memory, CPU, GPU, and Network performance of your app from the other tabs of the Android Monitor. This can be helpful for debugging and performance tuning your code.

2. The default log level is **Verbose**. In the drop-down menu, change the log level to **Debug**.



statements that you add to your app code print a message specified by you in the logcat tab of the Android Monitor. For example:

```
Log.d("MainActivity", "Hello World");
```

The parts of the message are:

3. Log – The [Log class](#). API for sending log messages.
4. d – The Log level. Used to filter log message display in logcat. “d” is for debug. Other log levels are “e” for error, “w” for warning, and “i” for info.
5. “MainActivity” – The first argument is a tag which can be used to filter messages in logcat. This is commonly the name of the activity from which the message originates. However, you can make this anything that is useful to you for debugging.

By convention, log tags are defined as constants:

```
private static final String LOG_TAG = MainActivity.class.getSimpleName();
```

- “Hello world” – The second argument is the actual message.

## 6.1 Add log statements to your app

1. Open your Hello World app in Android studio, and open MainActivity file.
2. **File > Settings > Editor > General >Auto Import** (Mac: **Android Studio > Preferences > Editor > General >Auto Import**). Check all boxes and set **Insert**

**imports** on paste to **All**. Unambiguous imports are now added automatically to your files. Note the "add unambiguous imports on the fly" option is important for some Android features such as NumberFormat. If not checked, NumberFormat shows an error. Click on 'Apply' followed by clicking on the 'Ok' button.

3. In the onCreate method, add the following log statement:

```
Log.d("MainActivity", "Hello World");
```

4. If the Android Monitor is not already open, click the Android Monitor tab at the bottom of Android Studio to open it. (See screenshot.)
5. Make sure that the Log level in the Android Monitor logcat is set to Debug or Verbose (default).
6. Run your app.

### Solution Code:

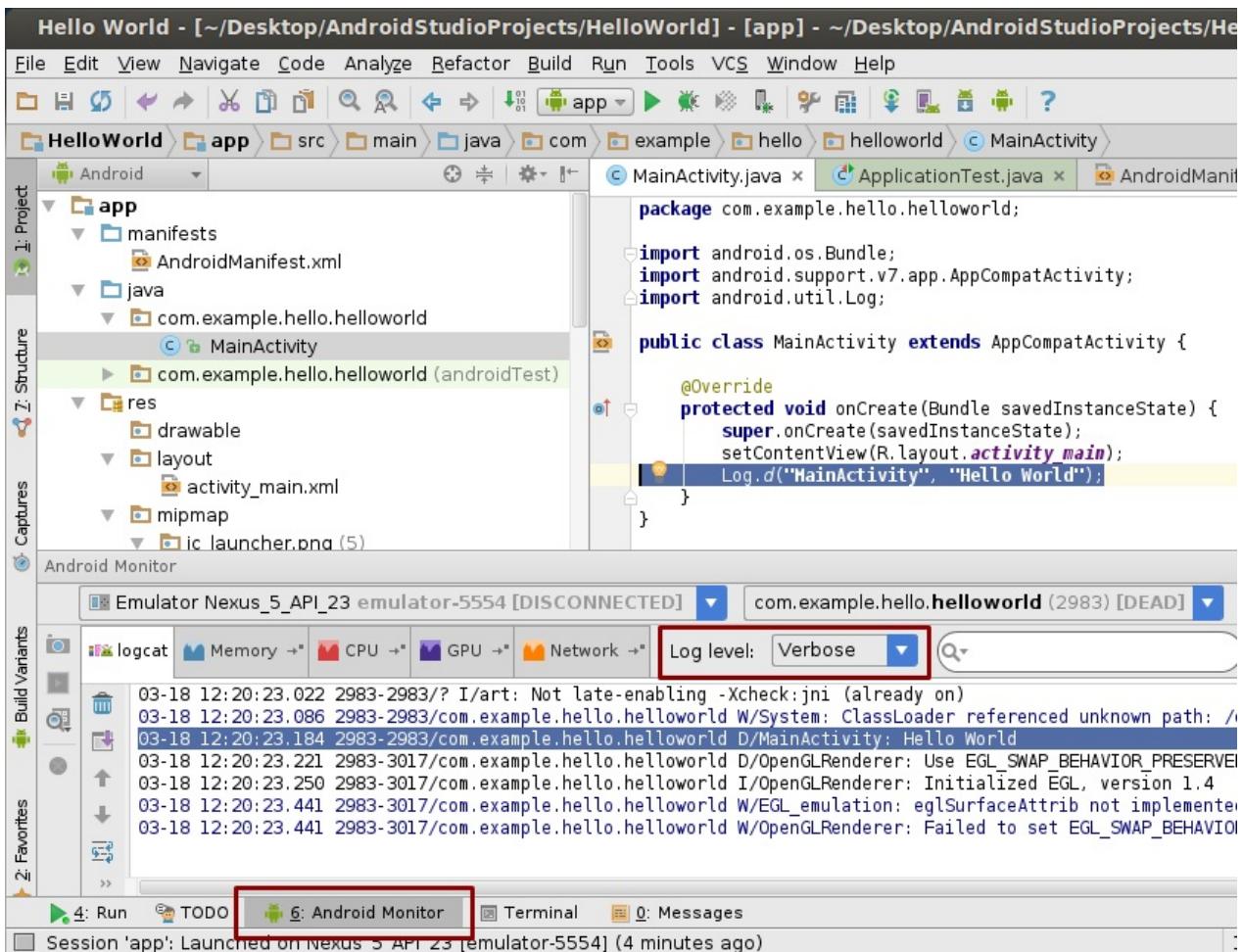
```
package com.example.hello.helloworld;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d("MainActivity", "Hello World");
    }
}
```

### Solution Log Message:

```
03-18 12:20:23.184 2983-2983/com.example.hello.helloworld D/MainActivity: Hello World
```



**Resources:** See [Reading and Writing Logs](#) for additional information.

**Challenge:** A common use of the Log class is to log Java exceptions when they occur in your program. There are some useful methods in the Log class that you can use for this purpose. Use the [Log class documentation](#) to find out what methods you can use to include an exception with a log message. Then, write code in the MainActivity.java file to trigger and log an exception.

## Task 7: Explore the AndroidManifest.xml file

Every app includes an Android Manifest file ( `AndroidManifest.xml` ). The manifest file contains essential information about your app and presents this information to the Android runtime system. Android must have this information before it can run any of your app's code.

In this practical you will find and read the `AndroidManifest.xml` file for the Hello World app.

**Why:** As your apps add more functionality and the user experience becomes more engaging and interactive, the `AndroidManifest.xml` file contains more and more information. In later lessons, you will modify this file to add features and feature permissions.

## 7.1 Explore the **AndroidManifest.xml** file

1. Open your Hello World app in Android studio, and in the **manifests** folder, open **AndroidManifest.xml**.
2. Read the file and consider what each line of code indicates. The code below is annotated to give you some hints.

**Annotated code:**

```
<!-- XML version and character encoding -->
<?xml version="1.0" encoding="utf-8"?>
<!-- Required starting tag for the manifest -->
<manifest
<!-- Defines the android namespace. Do not change. -->
xmlns:android="http://schemas.android.com/apk/res/android"
<!-- Unique package name of your app. Do not change once app is
published. -->
package="com.example.hello.helloworld">
<!-- Required application tag -->
<application
    <!-- Allow the application to be backed up and restored. -->
    android:allowBackup="true"
    <!-- Icon for the application as a whole,
        and default icon for application components. -->
    android:icon="@mipmap/ic_launcher"
    <!-- User-readable for the application as a whole,
        and default icon for application components. Notice that Android
        Studio first shows the actual label "Hello World".
        Click on it, and you see that the code actually refers to a string
        resource. Ctrl-click app_name to see where the resource is
        specified. You will learn more about this later. -->
    android:label="@string/app_name"
    <!-- Whether the app is willing to support right-to-left layouts.-->
    android:supportsRtl="true"
    <!-- Default theme for styling all activities. -->
    android:theme="@style/AppTheme">
    <!-- Declares an activity. One is required.
        All activities must be declared,
        otherwise the system cannot see and run them. -->
    <activity
        <!-- Name of the class that implements the activity;
            subclass of Activity. -->
        android:name=".MainActivity">
        <!-- Specifies the intents that this activity can respond to.-->
        <intent-filter>
            <!-- The action and category together determine what
                happens when the activity is launched. -->
            <!-- Start activity as the main entry point.
                Does not receive data. -->
            <action android:name="android.intent.action.MAIN" />
            <!-- Start this activity as a top-level activity in
                the launcher . -->
            <category android:name="android.intent.category.LAUNCHER" />
        <!-- Closing tags -->
        </intent-filter>
    </activity>
</application>
</manifest>
```

**Challenge:**

There are many other elements that can be set in the Android Manifest. Explore the [Android Manifest documentation](#) and learn about additional elements in the Android Manifest.

## Task 8. Explore the build.gradle file

Android Studio uses a build system called Gradle. Gradle does incremental builds, which allows for shorter edit-test cycles.

To learn more about Gradle, see:

- [Gradle site](#)
- [Configure your build](#) developer documentation
- Search the internet for "gradle tutorial".

In this task, you will explore the `build.gradle` file.

**Why:** When you add new libraries to your Android project, you may also have to update your **build.gradle file**. It's useful to know where it is and its basic structure.

### 8.1 Explore the build.gradle(Module app) file

1. In your project hierarchy, find **Gradle Scripts** and expand it. There several `build.gradle` files. One with directives for your whole project, and one for each app module. The module for your app is called "app". In the Project view, it is represented by the **app** folder at the top-level of the Project view.
2. Open **build.gradle (Module.app)**.
3. Read the file and consider what each line of code indicates.

**Solution:**

```
// Add Android-specific build tasks
apply plugin: 'com.android.application'
// Configure Android specific build options.
android {
    // Specify the target SDK version for the build.
    compileSdkVersion 23
    // The version of the build tools to use.
    buildToolsVersion "23.0.2"
    // Core settings and entries. Overrides manifest settings!
    defaultConfig {
        applicationId "com.example.hello.helloworld"
        minSdkVersion 15
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
    }
    // Controls how app is built and packaged.
    buildTypes {
        // Another common option is debug, which is not signed by default.
        release {
            // Code shrinker. Turn this on for production along with
            // shrinkResources.
            minifyEnabled false
            // Use ProGuard, a Java optimizer.
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}
// This is the part you are most likely to change as you start using
// other libraries.
dependencies {
    // Local binary dependency. Include any JAR file inside app/libs.
    compile fileTree(dir: 'libs', include: ['*.jar'])
    // Configuration for unit tests.
    testCompile 'junit:junit:4.12'
    // Remote binary dependency. Specify Maven coordinates of the Support
    // Library needed. Use the SDK Manager to download and install such
    // packages.
    compile 'com.android.support:appcompat-v7:23.2.1'
}
```

**Resources:** To learn more about Gradle, start with the [Gradle Wikipedia page](#).

### Challenge:

- For a deeper look into Gradle check out the [Build System Overview](#) and [Configuring Gradle Builds](#) documentation.
- There are tools to help you [shrink your code](#), remove unnecessary libraries/resource and even obfuscate your program to prevent unwanted reverse-engineering.

- Android Studio itself provides some useful features. Learn more about a valuable open-source tool called [ProGuard](#).

## Task 9. [Optional] Run your app on a device

In this final task, you will run your app on a physical mobile device such as a phone or tablet.

**Why:** Your users will run your app on physical devices. You should always test your apps on both virtual and physical devices.

What you need:

- An Android device such as a phone or tablet.
- A data cable to connect your Android device to your computer via USB port.
- If you are using a Linux or Windows OS, you may need to perform additional steps to run on a hardware device. Check the [Using Hardware Devices](#) documentation. On Windows, you may need to install the appropriate USB driver for your device. See [OEM USB Drivers](#).

### 9.1 [Optional] Run your app on a device

To let Android Studio communicate with your device, you must turn on USB Debugging on your Android device. This is enabled in the Developer options settings of your device. Note this is not the same as rooting your device.

On Android 4.2 and higher, the Developer options screen is hidden by default. To show Developer options and enable USB Debugging:

1. On your device, open **Settings > About phone** and tap **Build number** seven times.
2. Return to the previous screen (**Settings**). **Developer options** appears at the bottom of the list. Click **Developer options**.
3. Choose **USB Debugging**.

Now you can connect your device and run the app from Android Studio.

1. Connect your device to your development machine with a USB cable.
2. In Android Studio, at the bottom of the window, click the Android Monitor tab. You should see your device listed in the top-left drop-down menu.
3. Click the Run button  in the toolbar. The **Select Deployment** window opens.
4. Select your device, and click **OK**.

Android Studio should install and runs the app on your device.

## Troubleshooting

If you Android Studio does not recognize your device, try the following:

- Unplug and replug your device.
- Restart Android Studio.
- If your computer still does not find the device or declares it "unauthorized":
  1. Unplug the device.
  2. On the device, open Settings->**Developer Options**.
  3. Tap **Revoke USB Debugging authorizations**.
  4. Reconnect the device to your computer.
  5. When prompted, grant authorizations.
- You may need to install the appropriate USB driver for your device. See the [Using Hardware Devices documentation](#) document.
- Check the latest documentation, programming forums, or get help from you instructors.

## Coding challenge

**Note:** All coding challenges are optional and not a prerequisite for the material in the next chapter.

Now that you are set up and familiar with the basic development workflow, do the following:

1. Create a new project in Android Studio.
2. Change the greeting to "Happy Birthday to " and someone with a recent birthday.
3. Change the background of the app using a birthday-themed image.
4. Take a screenshot of your finished app and email it to someone whose birthday you forgot.

Explore Android Studio using the documentation at [developers.android.com](#).

## Summary

In this chapter, you:

- Installed Android Studio and deployed the Hello World app in the Android emulator and [optionally] on a mobile device.
- acquired a basic understanding of the structure of an Android app.
- added log statements that give you a basic tool for debugging.
- obtained a basic understanding of the development workflow in Android Studio.

# Resources

## Developer Documentation:

- [Android Studio download page](#)
- [Android Studio documentation](#)
- More about density, see [Supporting multiple screens](#)

# 1.2 P: Make Your First Interactive UI

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Create the "Hello Toast" project
- Task 2: Add views to "Hello Toast" in the Layout Editor
- Task 3: Edit the "Hello Toast" Layout in XML
- Task 4: Add on-click handlers for the buttons
- Coding challenge
- Summary
- Resources

The user interface displayed on the screen of a mobile Android device consists of a hierarchy of "views". [Views](#) are Android's basic user interface building blocks. For example, views can be components that:

- display text ([TextView](#) class)
- allow you to edit text ([EditText](#) class)
- represent pressable buttons ([Button](#) class) and other interactive components
- contain scrollable text ([ScrollView](#)) and scrollable items ([RecyclerView](#))
- show images ([ImageView](#))
- contain other views and position them ([LinearLayout](#)).
- pop up [menus](#) and other interactive components.

You specify the views in XML layout files. You can explore the view hierarchy of your app using [Hierarchy Viewer](#).

The Java code that displays and drives the user interface is contained in a class that extends [Activity](#) and contains methods to inflate views, that is, take the XML layout of views and display it on the screen. For example, the `MainActivity` in Hello World inflates a text view and prints Hello World. In more complex apps, an activity might implement click and other event handlers, request data from a database or the internet, or draw graphical content.

Android makes it straightforward to clearly separate UI elements and data from each other, and use the activity to bring them back together. This separation is an implementation of an [MVP \(Model-View-Presenter\)](#) architecture.

You will work with [Activities](#) and [Views](#) throughout this book.

In this practical you will:

- Create your app's user interface (UI) using the Android Studio Layout Editor and XML.
- Experiment with different UI elements.
- Programmatically access UI elements.
- Use string resources.
- Add on-click functionality to a button to programmatically change the UI.

## What you should already KNOW

For this practical you should be familiar with:

- How to create a Hello World app with Android Studio.

## What you will LEARN

- How to create interactive user interfaces in the Layout Editor, in XML, and programmatically.
- A lot of new terminology. Check out the [Vocabulary words and concepts glossary](#) for friendly definitions.

## What you will DO

- Create an app and add user interface elements such as buttons in the Layout Editor.
- Edit the app's layout in XML.
- Add a button to the app. Use a string resource for the label.
- Implement a click handler method for the button to display a message on the screen when the user clicks.
- Change the click handler method to change the message shown on the screen.

## App Overview

The "Hello Toast" app will consist of two buttons and one text view. When you click on the first button, it will display a short message, or toast, on the screen. Clicking on the second button will increase a click counter; the total count of mouse clicks will be displayed in the text view.

Here's what the finished app will look like:



# Task 1. Create a new "Hello Toast" project

In this practical, you will create and configure a project for the “Hello Toast” app.

## 1.1. Create the "Hello Toast" project

- Start Android Studio and create a new project with the following parameters:

Attribute	Value
Application Name	Hello Toast
Company Name	com.example.android or your own domain
Phone and Tablet Minimum SDK	API15: Android 4.0.3 IceCreamSandwich
Template	Empty Activity
Generate Layout file box	Checked
Backwards Compatibility cox	Checked

- Select **Run > Run app** or click the **Run icon**  in the toolbar to build and execute the app on the emulator from Practicals 1.1 or your device.

# Task 2: Add views to "Hello Toast" in the Layout Editor

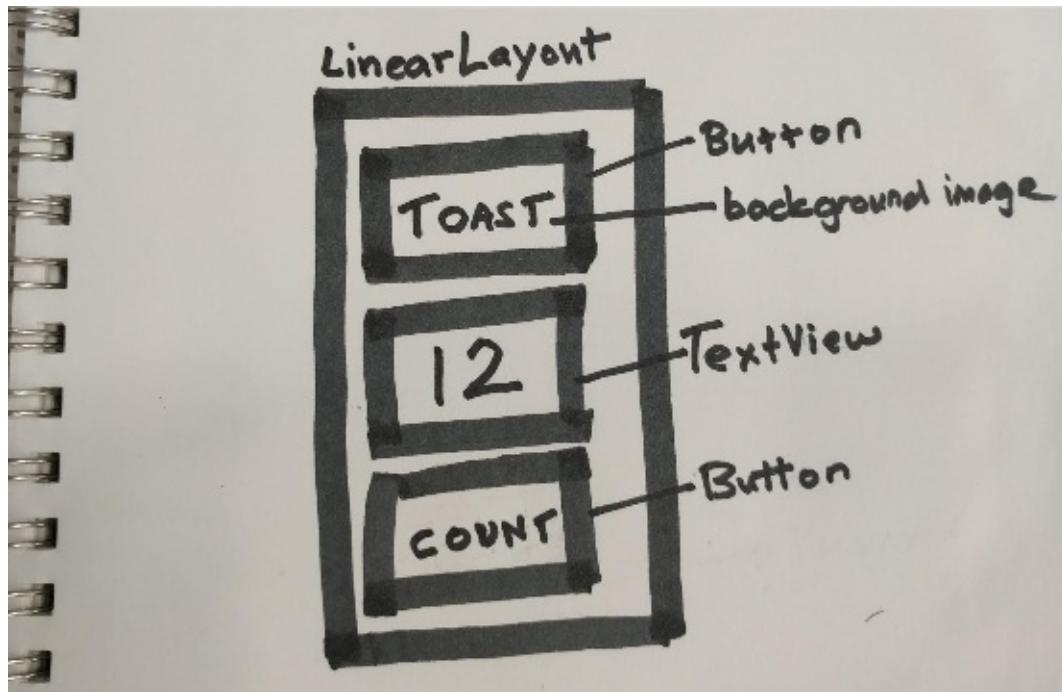
In this task, you will create and configure a user interface for the “Hello Toast” app by arranging view UI components on the screen.

**Why:** Every app should start with the user experience, even if the first implementation is very simple.

**Views** used for Hello Toast are:

- [TextView](#) - A view that displays text.
- [Button](#) - A button with a label that is usually associated with a click handler.
- [LinearLayout](#) - A view that acts as a container to arrange other view. This type of view extends the [ViewGroup](#) class and is also called a view group. LinearLayout is a basic view group that arranges its collection of views in a horizontal or vertical row.

Here is a rough sketch of the UI you will build in this exercise. Simple UI sketches can be very useful for deciding which views to use and how to arrange them, especially when your layouts become more sophisticated.



## 2.1 Explore the Layout Editor

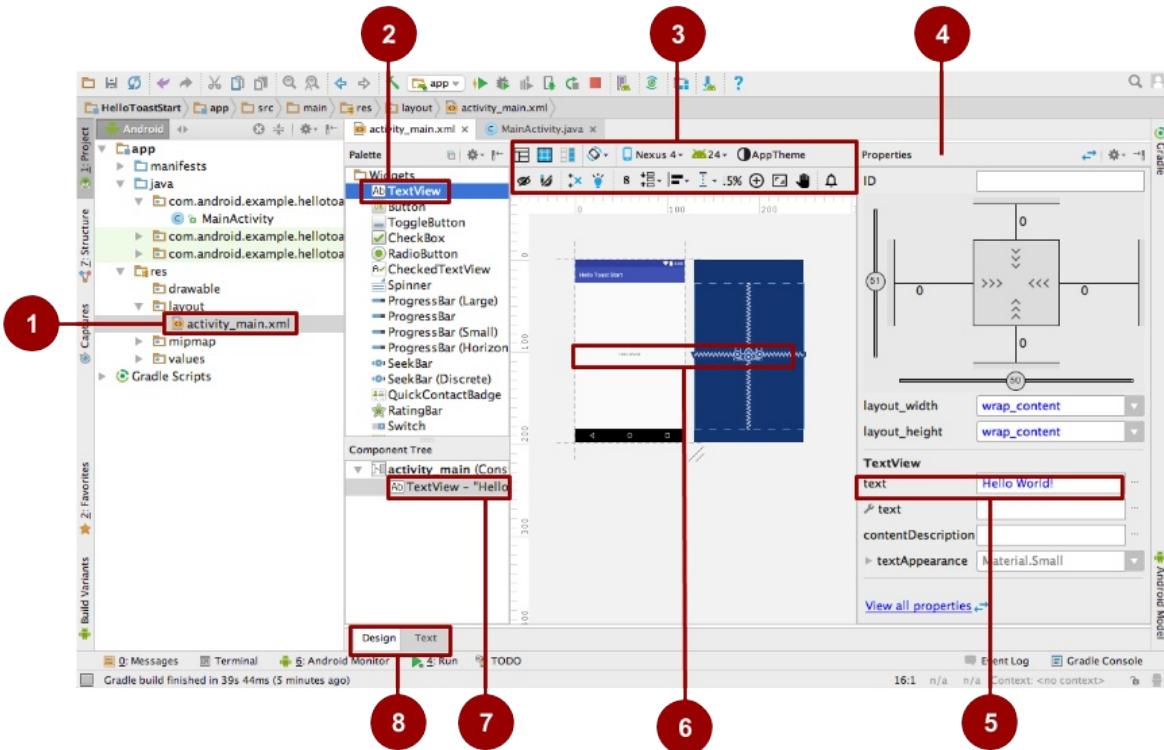
Use the Layout Editor to create the layout of the user interface elements, and to preview your app using different devices and app themes, resolutions, and orientations.

Refer to the screenshot below to match

1. In the **app > res > layout** folder, open the **activity\_main.xml** file (1).

Your Android Studio Screen should look similar to the screenshot below. If you see the XML code for the UI layout, click the **Design** tab below the Component Tree (8).

2. Using the annotated screenshot below as your guideline, explore the Layout Editor.



3. Find the different ways in which the "Hello World" string's UI element, a TextView, is represented.

- In the Palette of UI elements (2) you can create a text view by dragging it into the design pane.
- Visually, in the Design pane (6).
- In the **Component Tree** (7), as a component in a hierarchy of UI elements called the View Hierarchy. That is, views are organized into a tree hierarchy of parents and children, where children inherit properties of their parents.
- In the **Properties pane** (4), as a list of its properties, where "Hello Toast" is the value of the text property of the TextView (5).

4. Use the selectors above the virtual device (3) to do the following:

- Change the theme for your app.
- Change the rotation to landscape.
- Use a different version of the SDK.
- Preview right-to-left writing style.
- Select a UI item and jump to its source code.

Use the tooltips on the icons to help you discover their function.

5. Switch between the **Design** and **Text** tabs (8). Some UI changes can only be made in code, and some are quicker to accomplish in the virtual device.
6. When you are done, undo your changes (for UI changes, use **Edit > Undo** or the keyboard shortcut for your operating system).

See the [Android Studio User Guide](#) for the full Android Studio documentation.

**Note:** If you get an error about a missing App Theme, try **File > Invalidate Caches / Restart** or choose a theme that does not generate the error. You can find additional help in [this stackoverflow post](#).

## 2.2 Change the view group to a LinearLayout

The root of the view hierarchy is a view group, which as implied by the name, is a view that contains other views.

By default, the Blank Template uses a [RelativeLayout](#) view group. This layout offers a lot of flexibility in positioning views in the view groups.

A vertical linear layout is one of the most common layouts. It is simple, fast, and always a good starting point. Change the view group to a vertical linear layout as follows:

1. In the **Component Tree** pane (7 in the previous screenshot), find the top or root view directly below the Device Screen, which should be `RelativeLayout`.
2. Click the **Text** tab (8) to switch to the code view of the layout.
3. In the second line of the code, find `RelativeLayout` and change it to `LinearLayout`.
4. Make sure the closing tag at the end of the code has changed to `</LinearLayout>`. if it hasn't changed automatically, change it yourself.
5. The  `android:layout_height`  is defined as part of the template. The default layout orientation a horizontal row. To change the layout to be vertical, add the following code below  `android:layout_height . android:orientation="vertical"`
6. From the menu bar, select: **Code > Reformat Code...** You may see "no lines changed: code is already properly formatted".
7. Run your code to make sure it still works.
8. Switch back to **Design**.
9. Verify in the **Component Tree** pane that the top element is now a `LinearLayout` with its `orientation` attribute set to "vertical".

**Solution Code:**

```
<?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="vertical"
    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="hellotoast.android.example.com.hellotoast.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />
</LinearLayout>
```

## 2.3 Add views to the Linear Layout in the Layout Editor

In this task you will delete the current TextView (for practice), and add a new TextView and two buttons to the LinearLayout as shown in the UI sketch for this task. Refer to the UI diagram above, if necessary.

### Add UI Elements

1. Make sure you click the **Design** tab (8) to show the virtual device layout.
2. Click the TextView whose text value is "Hello World" in the virtual device layout or the **Component Tree** pane (7).
3. Press the Delete key to remove that TextView.
4. From the **Palette** pane (2), drag and drop a Button element, a Plain TextView, and another Button element, in that order, one below the other into the virtual device layout.

### Adjust the UI Elements

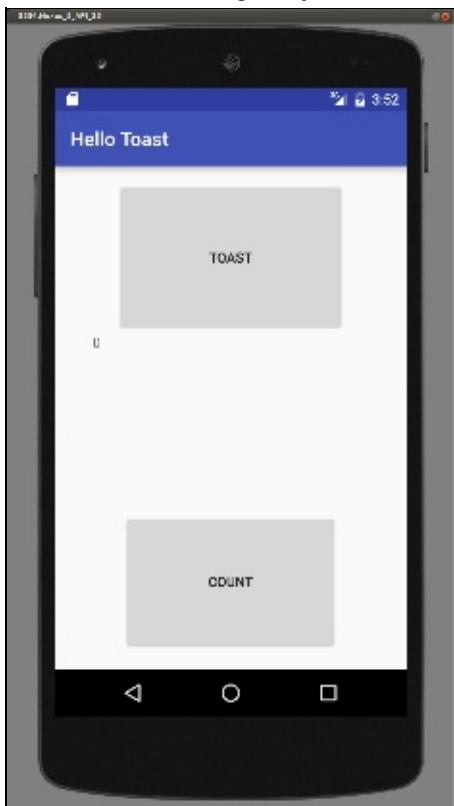
1. Move the elements in the device layout. Resize elements in the Component Tree by changing the `android:layout_width` and `android:layout_height` until each takes up a third of the screen and they roughly match the UI sketch.
2. To identify each view uniquely within an activity, each view needs a unique id. And to be of any use, the buttons need labels and the text views need to show some text. Double-click each element in the Layout Manager to see its properties and change the **text** and **id** strings as follows:

Element	Text	ID
Top button	Toast	button_toast
Text view	0	show_count
Bottom button	Count	button_count

3. Run your app.

### Solution Layout:

There should be three Views on your screen. They may not match the sizes on the image below, but as long as you have three Views in a vertical layout, you are doing fine!



[TODO] New Screenshot

**Challenge:** Think of an app you might want and create a project and layout for it using Layout Editor. Explore more of the features of Layout Editor. As mentioned before, the Layout Editor has a rich set of features and coding shortcuts. Check the [Android Studio documentation](#) to dive deeper.

**Challenge:** Use the [Hierarchy Viewer tool](#) to explore the view hierarchy of your app.

## Task 3: Edit the "Hello Toast" layout in XML

In this practical, you will edit the XML code for the Hello Toast app UI layout. You will also edit the properties of the views you have created. You can find the properties common to all views in the [View class documentation](#).

**Why:** While the Layout Editor is a powerful tool, some changes are easier to make directly in the XML source code. It is a personal preference to use either the graphical LayoutEditor or edit the XML file directly.

1. Open `res/layout/activity_main.xml` in Text mode.
2. In the menu bar select **Code > Reformat Code...**
3. Examine the code created by the Layout Editor.

Note that your code may not be an exact match, depending on what changes you made in the Layout Editor. Use the sample solutions as guidelines.

## 3.1 Examine LinearLayout properties

A LinearLayout is required to have these properties:

- `layout_width`
- `layout_height`
- `orientation`

The `layout_width` and `layout_height` can take one of three values:

- The **match\_parent** attribute expands the view to fill its parent by width or height. When the LinearLayout is the root view, it expands to the size of the device screen.
- The **wrap\_content** attribute shrinks the view dimensions just big enough to enclose its content. (If there is no content, the view becomes invisible.)
- Use a **fixed number of dp (device independent pixels)** to specify a fixed size, adjusted for the screen size of the device. For example, “16dp” means 16 device independent pixels.

The orientation can be:

- **horizontal:** views are arranged from left to right.
- **vertical:** views are arranged from top to bottom.

Change the LinearLayout of "Hello Toast" as follows:

Property	Value
<code>layout_width</code>	<code>match_parent</code> (to fill the screen)
<code>layout_height</code>	<code>match_parent</code> (to fill the screen)
<code>orientation</code>	<code>vertical</code>

## 3.2 Create string resources

Instead of hard-coding strings into the XML code, it is a best practice to use string resources, which represent the strings

**Why:** Having the strings in a separate file makes it easier to manage them, especially if you use these strings more than once. Also, string resources are mandatory for translating and localizing your app as you will create one string resource file for each language.

1. Place the cursor on the word "Toast".
2. Press **Alt-Enter (Option-Enter on the Mac)**.
3. Select **Extract string resources**.
4. Set the **Resource name** to `button_label_toast` and click **OK**. (If you make a mistake, undo the change with **Ctrl-Z**.)

This creates a string resource in the `values/res/string.xml` file, and the string in your code is replaced with a reference to the resource,

```
@string/button_label_toast
```

5. Extract and name the remaining strings from the views as follows:

View	String	Resource name
Button	Hello Toast!	button_label_toast
TextView	0	count_initial_value
Button	Count	button_label_count

6. In the Project view, navigate to **values/strings.xml** to find your strings. Now, you can edit all your strings in one place.

## 3.3 Resize and extract dimensions

Similar to strings, it is a best practice to extract view dimensions from the main XML file into a dimensions resource located in a file.

**Why:** This makes it easier to manage dimensions, especially if you need to adjust your layout for different device resolutions. It also makes it easy to have consistent sizing, and change the size of multiple objects by changing one property.

Do the following:

1. Change the `layout_width` of all elements inside the `LinearLayout` to "300dp".

If you want to use the graphical Layout Editor, click on the **Design** tab, select each element in the **Component Tree** pane and change the `layout:width` property in the **Properties** pane. If you want to directly edit the XML file, click on the **Text** tab, change the `android:layout_width` for the first Button, the TextView, and the last Button.

2. Click the **Text** tab to show the XML code, if you haven't already done so.
3. Place the cursor on one of the "300dp".
4. Press **Alt-Enter (Option-Enter on the Mac)**.
5. Click **Extract dimension resource**.
6. Set the **Resource name** to `my_view_width`, and click **OK**. (If you make a mistake, you can undo the change with **Ctrl-Z**).
7. Copy the new resource name `@dimen/my_view_width` and paste it to replace the "300dp" in each element.
8. In the Project view, navigate to **values/dimens.xml** to find your dimensions. The `dimens.xml` file applies to all devices. The `dimens.xml` file for `w820dp` applies only to devices that are wider than 820dp.
9. Change the `layout_height` of the TextView to "300dp".
10. Repeat the process (steps 3-7) to extract the height dimension for the text view.
11. Set the **Resource name** to `counter_height` and click **OK**.

## 3.4 Set colors and backgrounds

Styles and colors are additional properties that can be extracted into resources. All views can have backgrounds that can be colors or images.

**Why:** Extracting styles and colors makes it easy to use them consistently throughout the app, and straightforward to change across all UI elements.

Experiment with the following changes:

1. Change the text size of the `show_count` TextView. "sp" stands for scale-independent pixel, and like dp, is a unit that scales with the size of the device.

```
    android:textSize="200sp"
```

2. Extract the text size of the TextView as a dimension resource named `count_text_size`.
3. Change the text weight of the `show_count` TextView to bold.

```
    android:textStyle="bold"
```

4. Change the text color of the text view in the show\_count text view to the primary color of the theme.

The `colorPrimary` color is one of the predefined theme base colors and is used for the app bar. In a production app, you could, for example, customize this to fit your brand. Using the base colors for other UI elements creates a uniform UI. See [Using the Material Theme](#). You will learn more about app themes and material design in a later practical.

```
    android:textColor="@color/colorPrimary"
```

5. Change the color of both buttons to be the primary color of the theme.

```
    android:background="@color/colorPrimary"
```

6. Change the color of the text in both buttons to white.

```
    android:textColor="@android:color/white"
```

Note this code accesses a resource provided by the `android` package. See [Accessing Resources](#).

7. Add a background color to the `TextView`.

```
    android:background="#FFFF00"
```

8. In the Layout Editor (Text tab), place your mouse cursor over this text view color and press **Alt-Enter (Option-Enter on the Mac)**.
9. Select **Choose color**, which brings up the color picker, and choose a color you like.
10. Open `values/colors.xml`. Notice that `colorPrimary` that you used earlier is defined here.
11. Using the colors in `values/colors.xml` as an example, add a resource named `myBackgroundColor` for your background color, and then use it to set the background of the text view.

## 3.5 Gravity

Specifying gravity properties gives you additional control over arranging views and content in linear layouts.

- The `android:layout_gravity` attribute specifies how a view is aligned within its *parent*

*View.*

- The `android:gravity` attribute specifies the alignment of the *content of a View within the View itself*.
- For all three views, add the **layout\_gravity** property to center the views horizontally on the screen.

```
    android:layout_gravity="center_horizontal"
```

- To center the text in a the TextView horizontally and vertically, add:

```
    android:gravity="center"
```

### Sample Solution: strings.xml

```
<resources>
    <string name="app_name">Hello Toast</string>
    <string name="button_label_count">Count</string>
    <string name="button_label_toast">Toast</string>
    <string name="count_initial_value">0</string>
    <string name="toast_button_toast">Hello Toast!</string>
</resources>
```

### Sample Solution: dimens.xml

```
<resources>
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
    <dimen name="my_view_width">300dp</dimen>
    <dimen name="count_text_size">200sp</dimen>
    <dimen name="counter_height">300dp</dimen>
</resources>
```

### Sample Solution: colors.xml

```
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
    <color name="myBackgroundColor">#FFF043</color>
</resources>
```

### Sample Solution: activity\_main.xml

```
<?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="vertical"
    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="hellotoast.android.example.com.hellotoast.MainActivity">

    <Button
        android:id="@+id/button_toast"
        android:layout_width="@dimen/my_view_width"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="@string/button_label_toast"
        android:background="@color/colorPrimary"
        android:textColor="@android:color/white" />

    <TextView
        android:id="@+id/show_count"
        android:layout_width="@dimen/my_view_width"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:gravity="center"
        android:text="@string/count_initial_value"
        android:textSize="@dimen/count_text_size"
        android:textStyle="bold"
        android:textColor="@color/colorPrimary"
        android:background="@color/myBackgroundColor" />

    <Button
        android:id="@+id/button_count"
        android:layout_width="@dimen/my_view_width"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="@string/button_label_count"
        android:background="@color/colorPrimary"
        android:textColor="@android:color/white" />
</LinearLayout>
```

**Note:** You'll notice that when the device is rotated, the counter button becomes hidden. This is because the height of the TextView was hardcoded to a fixed amount of dp's, greater than the height of the screen when rotated. To resolve this issue, see the Challenge questions below.

### Challenges:

- What are the `android:weightSum` and `android:layout_weight` attributes for? Find out

using the [developer.android.com documentation](https://developer.android.com).

- Change the Hello Toast app to behave properly when the screen is rotated using the `android:weightSum` and `android:layout_weight` attributes.
- Create a new project with 5 views. Have one view use the top-half of the screen, and the other 4 views share the bottom half of the screen. Use only a LinearLayout and weights to accomplish this.
- Use an image as the background of the Hello Toast app. Add an image to the drawable folder, then set it as the background of the root view. For a deep dive into drawables, see the [Drawable Resources documentation](#).

### Resources:

- All Views are subclasses of the [View class](#) and therefore inherit many properties of the View superclass.
- You can find information on all Button properties in the [Button class documentation](#), and all the TextView properties in the [TextView class documentation](#).
- You can find information on all the LinearLayout properties in the [LinearLayout class documentation](#).
- The [Android resources documentation](#) will describe other types of resources.
- Android color constants: [Android standard R.color resources](#)
- More information about dp and sp units can be found at [Supporting Different Densities](#)

## Task 4: Add onClick handlers for the buttons

In this task, you will add methods to your MainActivity that execute when the user clicks on each button.

**Why:** Interactive apps must respond to user input

To connect a user action in a view to application code, you need to do two things:

- Write a method that performs a specific action when a user; for example: when a user clicks an on-screen button.
- Associate this method to the view, so the method executes when the user interacts with the view.

### 4.1 Add an onClick property to a button

A click handler is a method that is invoked when the user clicks on a user interface element.

In Android, you can specify the name of the click handler method for each view in the XML layout file with the `android:onClick` property.

1. Open `res/layout/activity_main.xml`.
2. Add the following property to the `button_toast` button.

```
    android:onClick="showToast"
```

3. Add the following attribute to the `button_count` button.

```
    android:onClick="countUp"
```

4. Inside of `activity_main.xml`, place your mouse cursor over each of these method names.
5. Press **Alt-Enter (Option-Enter on the Mac)**, and select **Create onClick event handler**.
6. Choose the **MainActivity** and click **OK**.

This creates placeholder method stubs for the `onClick` methods in `MainActivity.java`.

**Note:** You can also add click handlers to views programmatically, which you will do in a later practical. Whether you add click handlers in XML or programmatically is largely a personal choice; though, there are situations where you can only do it programmatically.

#### Solution `MainActivity.java`:

```
package hellotoast.android.example.com.hellotoast;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;

public class MainActivity extends AppCompatActivity {

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

    public void countUp(View view) {
        // What happens when user clicks on the button_count Button goes here.
    }

    public void showToast(View view) {
        // What happens when user clicks on the button_toast Button goes here.
    }
}
```

## 4.2 Show a toast when the Toast button is clicked

A [toast](#) provides simple feedback about an operation in a small popup. It only fills the amount of space required for the message and the current activity remains visible and interactive. Toasts provide another way for you to test the interactivity of your app.

In `MainActivity.java`, add code to the `showToast()` method to show a toast message.

To create an instance of a [Toast](#), you call `makeText()` on the `Toast` class, supplying a context (see below), the message to display, and the duration of display. You display the toast calling `show()`. This is a boilerplate pattern, so you can reuse the code you are going to write.

1. Get the context of the application.

Displaying a toast requires a context. The [context](#) of an application contains global information about the application environment. Since a toast displays on top of the visible UI, the system needs information about the application.

```
Context context = getApplicationContext();
```

2. The length of a toast string can be either short or long, and you specify which one by using a `Toast` constant.

- `Toast.LENGTH_LONG`
- `Toast.LENGTH_SHORT`

The actual lengths are about 3.5s for the long toast and 2s for the short toast. The values are specified in the Android source code. See [this Stackoverflow post](#) details.

3. Create an instance of the `Toast` class with the context, message, and duration.

- The context is the application context we got earlier.
- The message is the string you want to display
- The duration is one of the

```
Toast toast = Toast.makeText(context, "Hello Toast", duration);
```

4. Extract the "Hello Toast" string into a string resource and call it

```
** toast_message </strong>
```

- i. Place the cursor on the string `"Hello Toast!"`.
- ii. Press **Alt-Enter (Option-Enter on the Mac)**.
- iii. Select **Extract string resources**.
- iv. Set the **Resource name** to `toast_message` and click **OK**.

This will store "Hello World" as a string resource name `toast_message` in the string resources file `res/values/string.xml`. The string parameter in your method call is replaced with a reference to the resource.

5. R. identifies the parameter as a resource.
6. string references the name of the XML file where the resources is defined.
7. toast\_message is the name of the resource.

```
Toast toast = Toast.makeText(context, R.string.toast_message, duration);
```

8. Display the toast.

```
toast.show();
```

9. Run your app and verify the toast shows when the **Toast** button is tapped.

**Solution:**

```
/*
 * When the TOAST button is clicked, show a toast.
 *
 * @param view The view that triggers this onClick handler.
 *             Since a toast always shows on the top, view is not used.
 */
public void showToast(View view) {
    // Interface to global information about an application environment
    Context context = getApplicationContext();
    int duration = Toast.LENGTH_LONG; // LENGTH_SHORT for a short toast

    // Create a toast and don't forget to also show it.
    Toast toast = Toast.makeText(context, R.string.toast_message, duration);
    toast.show();
}
```

## 4.3 Increase the count in the text view when the Count button is clicked

To display the current count in the text view:

- Keep track of the count as it changes.
- Send the updated count to the text view to display it on the user interface.

Implement this as follows:

1. In `MainActivity.java`, add a class variable `count` to track the count and start it at 0.
2. In the `countUp()` method, increase the value of the `count` variable each time the button is clicked.
3. Get a reference to the text view using the id you set in the layout file.

Views, like strings and dimensions, are resources that can have an id. The [findViewById](#)) call takes the id of a view as its parameter and returns the view. Because the method returns a View, you have to cast the result to the view you expect.

```
TextView showCount = (TextView) findViewById(R.id.show_count);
```

4. Set the text in the text view to the value of the count variable.

```
showCount.setText("" + count);
```

5. Run your app to verify that the count increases when the **Count** button is pressed.

**Solution:**

**Class definition and initializing count variable:**

```
public class MainActivity extends AppCompatActivity {  
    int count = 0;  
  
    coutUp Method:  
    public void countUp(View view) {  
        count++;  
        TextView showCount = (TextView) findViewById(R.id.show_count);  
        showCount.setText("" + count);  
    }  
}
```

**Resources:**

- Learn more about handling [Android Input Events](#).
- [Context class documentation](#)

## Coding challenge

**Note:** All coding challenges are optional and not a prerequisite for the material in the next chapter.

Even a simple app like Hello Toast can be the foundation of many scoring or product ordering apps. Write one app would be of use to you, or try one of these examples:

- Create a coffee ordering app. Add buttons to change the number of coffees ordered. Calculate and display the total price.
- Create a scoring app for your favorite team sport. Make the background an image that represents that sport. Create buttons to count the scores for each team.

# Summary

In this chapter, you:

- Added UI elements to an app in the Layout Editor and using XML.
- Made the UI interactive with buttons. and click listeners
- Add click listeners that update a text view in response to user input.
- Displayed information to users using a toast.

# Resources

## Developer Documentation:

- [Android Studio documentation](#)
- [Vocabulary words and concepts glossary](#)
- [developer.android.com Layouts](#)
- [View class documentation](#)
- [device independent pixels](#)
- [Button class documentation](#)
- [TextView class documentation](#)
- [Android resources documentation](#)
- [Complete code for the Hello Toast app](#)

# 1.3 P: Working with TextView Elements

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App Overview](#)
- [Task 1: Add several text views](#)
- [Task 2: Add active web links and a ScrollView](#)
- [Task 3: Scroll multiple elements](#)
- [Coding Challenge](#)
- [Summary](#)
- [Resources](#)

The [TextView](#) class is a subclass of the [View](#) class that displays text on the screen. You can control how the text appears with TextView attributes in the XML layout file. This practical shows how to work with multiple TextView elements, including one that the user can scroll its contents vertically.

If the information you want to show in your app is larger than the device's display, you can create a *scrolling view* that the user can scroll vertically by swiping up or down, or horizontally by swiping right or left.

You would typically use a scrolling view for news stories, articles, or any lengthy text that doesn't completely fit on the display. You can also use a scrolling view to enable users to enter multiple lines of text, or to combine View elements (such as a text field and a button) within a scrolling view.

The [ScrollView](#) class provides the layout for the scrolling view. ScrollView is a subclass of [FrameLayout](#), which means that you should place only *one* View as a child within it, where the child View contains the entire contents to scroll. And this child View may itself be a layout manager with a complex hierarchy of objects, such as a [LinearLayout](#). Note that complex layouts may suffer performance issues with child Views such as images. A good choice for a View within a ScrollView is a [LinearLayout](#) that is arranged in a vertical orientation, presenting top-level items that the user can scroll through.

With a ScrollView, all of your views are in memory and in the View hierarchy even if they aren't displayed on screen. This makes ScrollView ideal for scrolling pages of free-form text smoothly, because the text is already in memory. However, ScrollView can use up a lot of

memory, which can affect the performance of the rest of your app. To display long lists of items that users can add to, delete from, or edit, you may want to consider using a [RecyclerView](#), which is described in a separate practical.

## What you should already KNOW

You should be familiar with:

- Creating a Hello World app with Android Studio.
- Running an app on the emulator or a device.
- Implementing a `TextView` in a layout for an app.
- Creating and using string resources.
- Converting layout dimensions to resources.

## What you will LEARN

- Using XML code to add multiple `TextView` elements.
- Using XML code to define a scrolling view.
- Displaying free-form text with some HTML formatting tags.
- Styling the `TextView` background color and text color.
- Including a web link in the text.

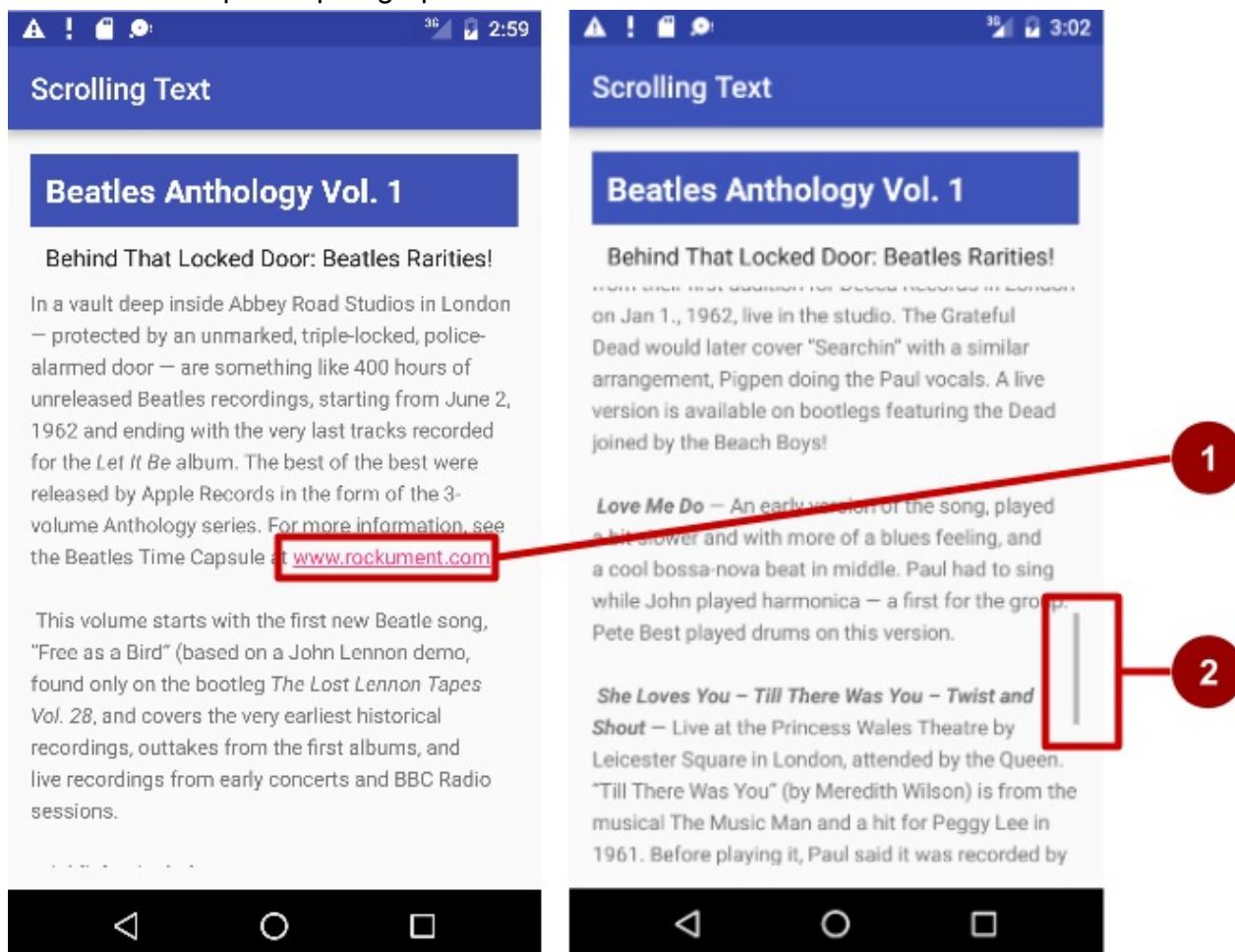
## What you will DO

In this practical application you will:

- Create the Scrolling Text app.
- Add two `TextView` elements for the article heading and subheading.
- Use `TextAppearance` styles and colors for the article heading and subheading.
- Use HTML tags in the text string to control formatting.
- Use the `lineSpacingExtra` attribute to add line spacing for readability.
- Add a `ScrollView` to the layout to enable scrolling a `TextView` element.
- Add the `autoLink` attribute to enable URLs in the text to be active and clickable.

## App Overview

The Scrolling Text app demonstrates the ScrollView UI component. ScrollView is a ViewGroup that in this example contains a TextView. It shows a lengthy page of text — in this case, a music album review — that the user can scroll vertically to read by swiping up and down. A scroll bar appears in the right margin. The app shows how you can use text formatted with minimal HTML tags for setting text to bold or italic, and with new-line characters to separate paragraphs. You can also include active web links in the text.



In the above figure, the following appear:

1. An active web link embedded in free-form text
2. The scroll bar that appears when scrolling the text

## Task 1: Add several text views

In this practical, you will create an Android project for the Scrolling Text app, add TextViews for an article title and subtitle, and change the existing “Hello World” TextView element to show a lengthy article.

You will make all these changes in the XML code and in the strings.xml file. You will edit the XML code for the layout in the Text pane, which you show by clicking the **Text** tab, rather than clicking the **Design** tab for the Design pane. Some changes to UI elements and

attributes are easier to make directly in the Text pane using XML source code.

## 1.1 Create the project and TextView elements

1. In Android Studio create a new project with the following parameters:

Attribute	Value
Application Name	Scrolling Text
Company Name	android.example.com (or your own domain)
Phone and Tablet Minimum SDK	API15: Android 4.0.3 IceCreamSandwich
Template	Empty Activity
Generate Layout File checkbox	Checked

2. In the **app > res > layout** folder, open the **activity\_main.xml** file, and click the **Text** tab to see the XML code if it is not already selected.

At the top, or *root*, of the view hierarchy is a ViewGroup called [RelativeLayout](#). Like other ViewGroups, RelativeLayout is a view that contains other views. In addition, it also allows you to position its child Views relative to each other or relative to the parent RelativeLayout itself. The default “Hello World” TextView element that is created for you by the Empty Layout template is a child View within the RelativeLayout view group. For more information about using a RelativeLayout, see [Relative Layout](#).

3. Add a `TextView` element above the “Hello World” `TextView`. As you enter `<TextView` to start a TextView, Android Studio automatically adds the ending `>`, which is shorthand for `</TextView>`. Add the following attributes to the TextView:

TextView #1 Attribute	Value
<code>layout_width</code>	<code>"match_parent"</code>
<code>layout_height</code>	<code>"wrap_content"</code>
<code>android:id</code>	<code>"@+id/article_heading"</code>
<code>android:background</code>	<code>"@color/colorPrimary"</code>
<code>android:textColor</code>	<code>"@android:color/white"</code>
<code>android:padding</code>	<code>"10dp"</code>
<code>android:textAppearance</code>	<code>"@android:style/TextAppearance.Large"</code>
<code>android:textStyle</code>	<code>"bold"</code>
<code>android:text</code>	<code>"Article Title"</code>

**Note:** The attributes for styling the text and background are summarized in the [TextView class documentation](#).

- Extract the string resource for the `android:text` attribute's hard-coded string "Article Title" in the `TextView` to create an entry for it in `strings.xml`.

Place the cursor on the hard-coded string, press Alt-Enter (Option-Enter on the Mac), and select **Extract string resource**. Then edit the Resource name for the string value to `article_title`.

**Note:** String resources are described in detail in the [String Resources documentation](#).

- Extract the dimension resource for the `android:padding` attribute's hard-coded string "10dp" in the `TextView` to create an entry in `dimens.xml`.

Place the cursor on the hard-coded string, press Alt-Enter (Option-Enter on the Mac), and select **Extract dimension resource**. Then edit the Resource name to `padding_regular`.

- Add another `TextView` element above the "Hello World" `TextView` and below the `TextView` you created in the previous steps. Add the following attributes to the `TextView`:

TextView #2 Attribute	Value
<code>layout_width</code>	"match_parent"
<code>layout_height</code>	"wrap_content"
<code>android:id</code>	"@+id/article_subheading"
<code>android:layout_below</code>	"@+id/article_heading"
<code>android:padding</code>	"@dimen/padding_regular"
<code>android:textAppearance</code>	"@android:style/TextAppearance"
<code>android:text</code>	"Article Subtitle"

Note that since you extracted the dimension resource for the "10dp" string to `padding_regular` in the previously created `TextView`, you can use

"@dimen/padding\_regular" for the `android:padding` attribute in this `TextView`.

- Extract the string resource for the `android:text` attribute's hard-coded string "Article Subtitle" in the `TextView` to `article_subtitle`.
- Add the following `TextView` attributes to the "Hello World" `TextView` element, and change the `android:text` attribute:

TextView Attribute	Value
<code>android:id</code>	"@+id/article"
<code>android:lineSpacingExtra</code>	"5sp"
<code>android:layout_below</code>	"@+id/article_subheading"
<code>android:text</code>	Change to "Article text"

9. Extract the string resource for "Article text" to **article\_text**, and extract the dimension resource for "5sp" to **line\_spacing**.

## 1.2 Add the text of the article

In a real app that accesses magazine or newspaper articles, the articles that appear would probably come from an online source through a content provider, or might be saved in advance in a database on the device. For this practical, you will create the article as a single long string in the strings.xml resource.

1. In the **app > res > values** folder, open **strings.xml**.
2. Enter the values for the strings `article_title` and `article_subtitle` with a made-up title and a subtitle for the article you are adding. The string values for each should be single-line text without HTML tags or multiple lines.
3. Enter or copy and paste text for the `article_text` string.

Use the text provided for the `article_text` string in the **strings.xml** file of the finished Scrolling View app , or use made-up plain text. You can copy and then paste the same sentence over and over, as long as the result is a long section of text that will not fit entirely on the screen. Keep in mind the following (refer to the figure below for an example):

- i. As you enter or paste text in the **strings.xml** file, the text lines don't wrap around to the next line — they extend beyond the right margin. This is the correct behavior — each new line of text starting at the left margin represents an entire paragraph.
- ii. Enter `\n` to represent the end of a line, and another `\n` to represent a blank line.

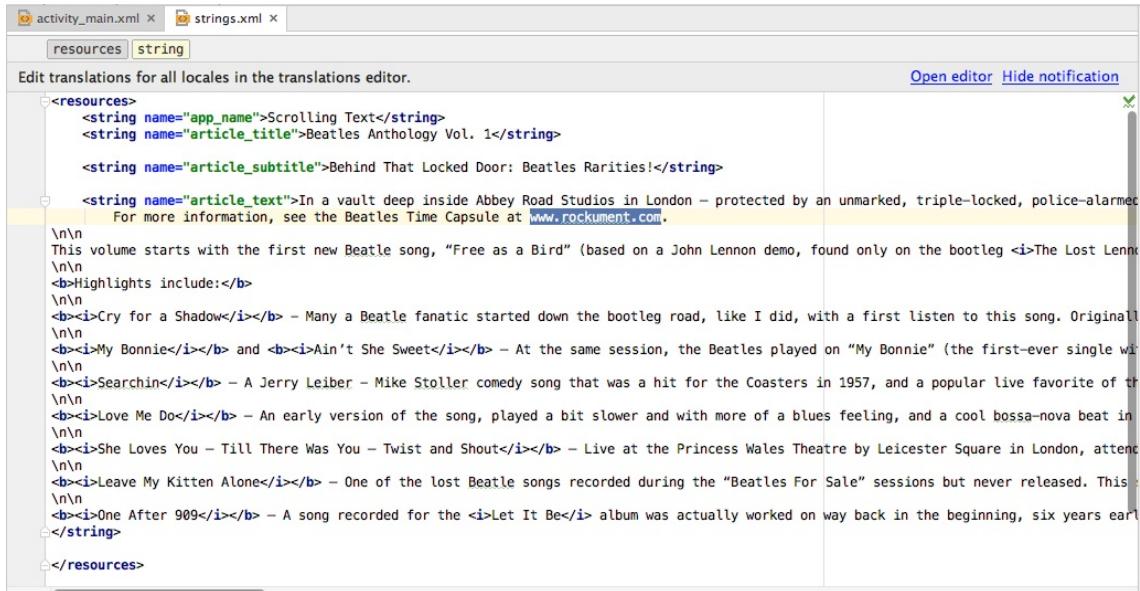
**Why?** You need to add end-of-line characters to keep paragraphs from running into each other.

**Tip:** If you want to see the text wrapped in strings.xml, you can press Return to enter hard line endings, or format the text first in a text editor with hard line endings.

- i. If you have an apostrophe ('') in your text, you must *escape* it by preceding it with a backslash (\'). If you have a double-quote in your text, you must also escape it (\"). You must also escape any other non-ASCII characters. See the "[Formatting and Styling](#)" section of String Resources for more details.
- ii. Enter the HTML and `</b>` tags around words that should be in bold.
- iii. Enter the HTML and `</i>` tags around words that should be in italics. Note, however, that if you use curled apostrophes within an italic phrase, you should replace them with straight apostrophes.
- iv. You can combine bold and italics by combining the tags, as in ... words...`</i></b>`. Other HTML tags are ignored.
- v. Enclose The entire text within `<string name="article_text"> </string>` in the

strings.xml file.

- vi. Include a web link to test, such as [www.google.com](http://www.google.com) (the example below uses [www.rockument.com](http://www.rockument.com)). *Don't* use an HTML tag — any HTML tags except the bold and italic tags will be ignored and presented as text, which is not what you want.



```

<resources>
    <string name="app_name">Scrolling Text</string>
    <string name="article_title">Beatles Anthology Vol. 1</string>
    <string name="article_subtitle">Behind That Locked Door: Beatles Rarities!</string>
    <string name="article_text">In a vault deep inside Abbey Road Studios in London – protected by an unmarked, triple-locked, police-alarmed
        For more information, see the Beatles Time Capsule at www.rockument.com.
    \n\n
        This volume starts with the first new Beatle song, "Free as a Bird" (based on a John Lennon demo, found only on the bootleg The Lost Len...
    \n\n
        <b>Highlights include:</b>
    \n\n
        <b><i>Cry for a Shadow</i></b> – Many a Beatle fanatic started down the bootleg road, like I did, with a first listen to this song. Originally...
    \n\n
        <b><i>My Bonnie</i></b> and <b><i>Ain't She Sweet</i></b> – At the same session, the Beatles played on "My Bonnie" (the first-ever single wi...
    \n\n
        <b><i>Searchin</i></b> – A Jerry Leiber – Mike Stoller comedy song that was a hit for the Coasters in 1957, and a popular live favorite of th...
    \n\n
        <b><i>Love Me Do</i></b> – An early version of the song, played a bit slower and with more of a blues feeling, and a cool bossa-nova beat in...
    \n\n
        <b><i>She Loves You – Till There Was You – Twist and Shout</i></b> – Live at the Princess Wales Theatre by Leicester Square in London, attend...
    \n\n
        <b><i>Leave My Kitten Alone</i></b> – One of the lost Beatle songs recorded during the "Beatles For Sale" sessions but never released. This...
    \n\n
        <b><i>One After 909</i></b> – A song recorded for the Let It Be album was actually worked on way back in the beginning, six years ear...
    </string>
</resources>

```

#### 4. Run the app.

The article appears, but notice you can't scroll it to see all of the text. Note also that tapping a web link does not currently do anything.

The activity\_main.xml layout file should now look like the following:

```
<?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: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.android.scrollingtext.MainActivity">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/article_heading"
        android:background="@color/colorPrimary"
        android:textColor="@android:color/holo_orange_light"
        android:textColorHighlight="@color/colorAccent"
        android:padding="10dp"
        android:textAppearance="@android:style/TextAppearance.Large"
        android:textStyle="bold"
        android:text="@string/article_title"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/article_subheading"
        android:layout_below="@+id/article_heading"
        android:padding="10dp"
        android:textAppearance="@android:style/TextAppearance"
        android:text="@string/article_subtitle"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/article"
        android:layout_below="@+id/article_subheading"
        android:lineSpacingExtra="5sp"
        android:text="@string/article_text"/>

</RelativeLayout>
```

## Task 2: Add active Web links and a ScrollView

In the previous task you created the Scrolling Text app with TextViews for an article title, subtitle, and lengthy article text. You also included a web link, but the link is not yet active. You will add the code to make it active.

Also, the `TextView` by itself can't enable users to scroll the article text to see all of it. You will add a new view group called `ScrollView` to the XML layout that will make the `TextView` scrollable.

## 2.1 Add the `autoLink` attribute for active web links

Add the `android:autoLink="web"` attribute to the `article` `TextView`. The XML code for this `TextView` should now look like this:

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/article"  
    android:lineSpacingExtra="5sp"  
    android:autoLink="web"  
    android:text="@string/article_text"/>
```

## 2.2 Add a `ScrollView` to the layout

To make a view (such as a `TextView`) scrollable, embed the view *inside* a `ScrollView`.

1. Add a `ScrollView` between the `article_subheading` `TextView` and the `article` `TextView`. As you enter `<ScrollView`, Android Studio automatically adds `</ScrollView>` at the end, and presents the `android:layout_width` and `android:layout_height` attributes with suggestions. Choose `wrap_content` from the suggestions for both attributes. The code should now look like this:

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/article_subheading"
    android:layout_below="@id/article_heading"
    android:padding="10dp"
    android:textAppearance="@android:style/TextAppearance"
    android:text="@string/article_subtitle"/>

<ScrollView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/article_subheading"></ScrollView>

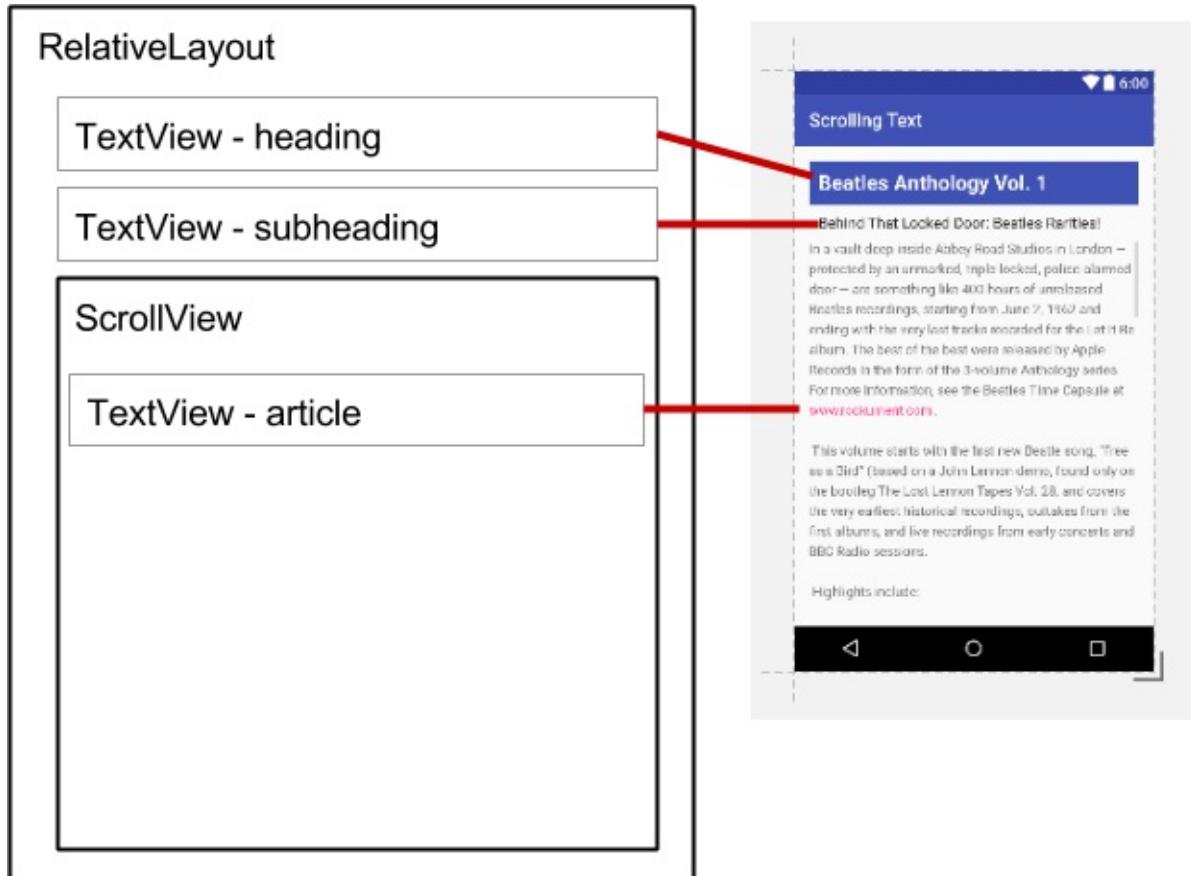
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/article"
    android:layout_below="@id/article_subheading"
    android:lineSpacingExtra="5sp"
    android:autoLink="web"
    android:text="@string/article_text"/>
```

Now move the ending `</ScrollView>` code *after* the `article` `TextView` so that the `article` `TextView` attributes are inside the `ScrollView` XML element.

2. Remove the following attribute from the `article` `TextView`, because the `ScrollView` itself will be placed below the `article_subheading` element, and this attribute for `TextView` would conflict with the `ScrollView`:

```
    android:layout_below="@id/article_subheading"
```

The layout should now look like this:

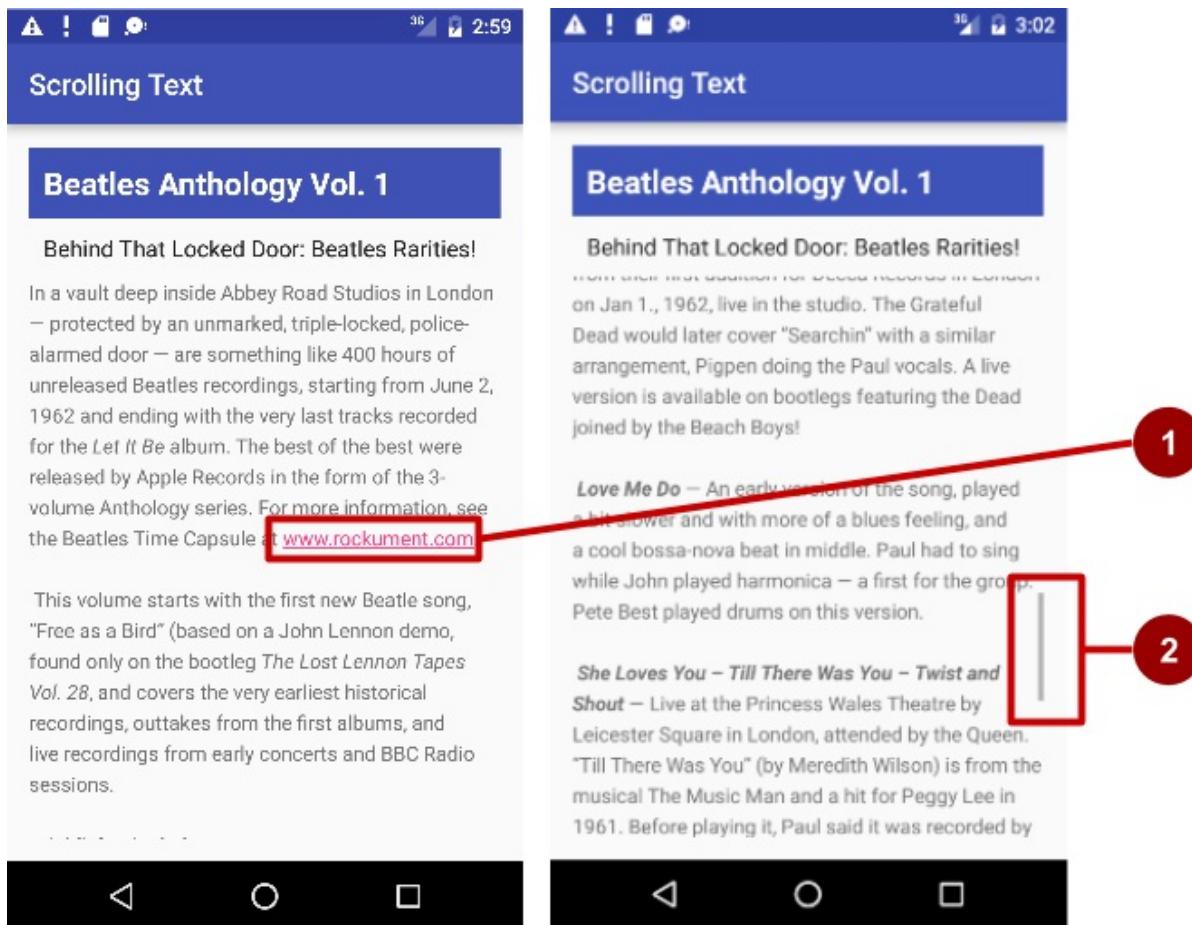


3. Choose **Code > Reformat Code** to reformat the XML code so that the `<article>` `</article>` `</TextView>` now appears indented inside the `<ScrollView>` `</ScrollView>` code.
4. Run the app.

Swipe up and down to scroll the article. The scroll bar appears in the right margin as you scroll.

Tap the web link to go to the web page. The `android:autoLink` attribute turns any recognizable URL in the `<TextView>` (such as [www.rockument.com](http://www.rockument.com)) into a web link.

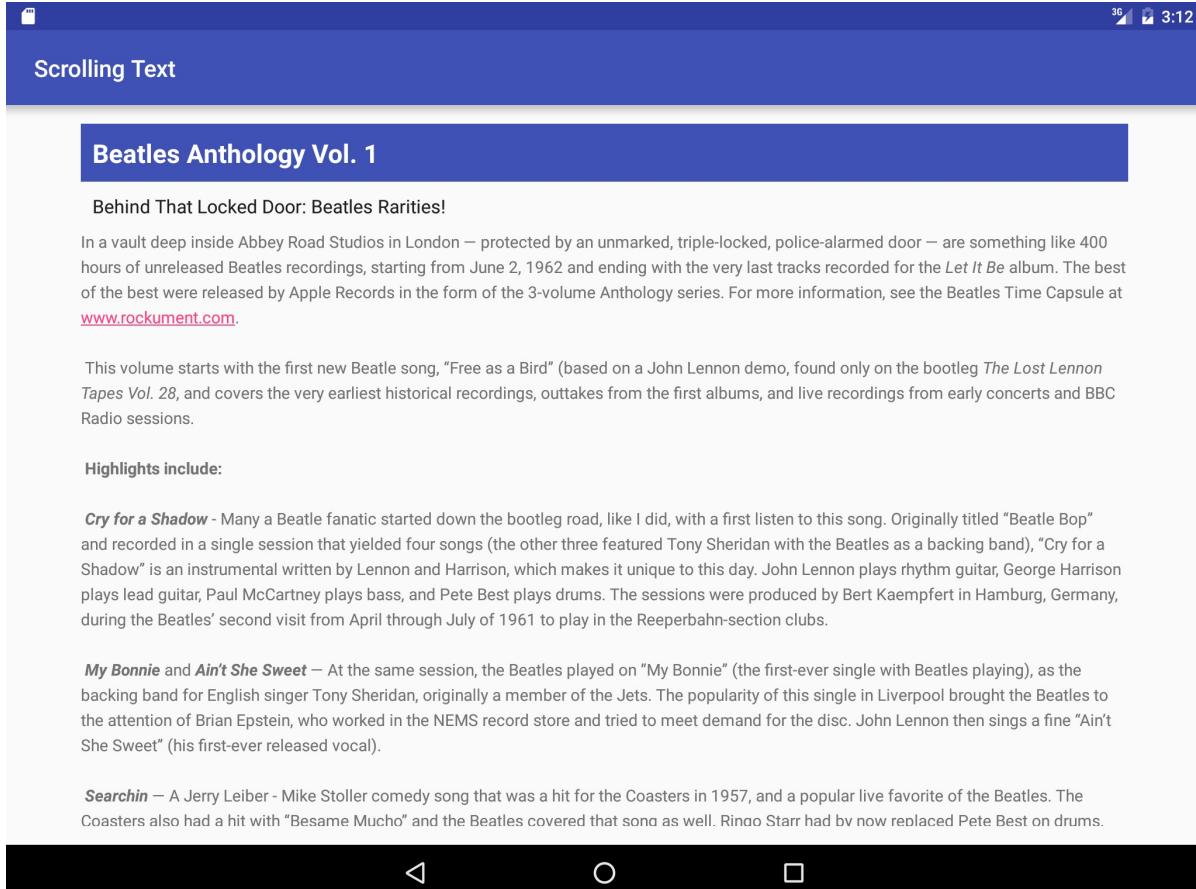
5. Rotate your device or emulator while running the app. Notice how the scrolling view widens to use the full display and still scrolls properly.
6. Run the app on a tablet or tablet emulator. Notice how the scrolling view widens to use the full display and still scrolls properly.



In the above figure, the following appear:

7. An active web link embedded in free-form text

## 8. The scroll bar that appears when scrolling the text



The **activity\_main.xml** layout file should now look like the following:

```
<?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: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.android.scrollingtext.MainActivity">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/article_heading"
        android:background="@color/colorPrimary"
        android:textColor="@android:color/white"
        android:paddingTop="10dp"
        android:paddingBottom="10dp"
        android:paddingLeft="10dp"
        android:paddingRight="10dp"
        android:textAppearance="@android:style/TextAppearance.Large"
        android:textStyle="bold"
        android:text="@string/article_title"/>
```

```
<TextView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/article_subheading"  
    android:layout_below="@id/article_heading"  
    android:paddingTop="10dp"  
    android:paddingBottom="10dp"  
    android:paddingLeft="10dp"  
    android:paddingRight="10dp"  
    android:textAppearance="@android:style/TextAppearance"  
    android:text="@string/article_subtitle"/>  
  
<ScrollView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_below="@id/article_subheading">  
  
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/article"  
    android:lineSpacingExtra="5sp"  
    android:autoLink="web"  
    android:text="@string/article_text"/>  
  
</ScrollView>  
</RelativeLayout>
```

## Task 3: Scroll multiple elements

As noted before, the ScrollView view group can contain only one View (such as the `article` `TextView` you created for the article); however, that View can be another view group that contains Views, such as `LinearLayout`. You can *nest* a view group such as `LinearLayout` *within* the `ScrollView` view group, thereby scrolling everything that is inside the `LinearLayout`.

For example, if you want the subheading of the article to scroll along with the article, add a `LinearLayout` within the `ScrollView`, and move the subheading, along with the article, into the `LinearLayout`. The `LinearLayout` view group becomes the single child View in the `ScrollView`, and the user can scroll the entire view group: the subheading and the article.

### 3.1 Add a `LinearLayout` to the `ScrollView`

1. On your computer, make a copy of Android Studio's project folder for `ScrollingText`, and rename the project to be `ScrollingText2`. To copy and rename a project, follow the

- “Copy and rename a project” instructions in the [Appendix](#).
2. Open **ScrollingText2** in Android Studio, and open the **activity\_main.xml** file to change the XML layout code.
  3. Add a **LinearLayout** above the **article** **TextView** in the **ScrollView**. As you enter **<LinearLayout**, Android Studio automatically adds **</LinearLayout>** to the end, and presents the **android:layout\_width** and **android:layout\_height** attributes with suggestions. Choose **match\_parent** and **wrap\_content** from the suggestions for its width and height, respectively. The code should now look like this:

```
<LinearLayout  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"></LinearLayout>
```

- You use **match\_parent** to match the width of the parent view group, and **wrap\_content** to make the view group only big enough to enclose its contents and padding.
4. Move the ending **</LinearLayout>** code *after* the **article** **TextView** but *before* the closing **</ScrollView>** so that the **LinearLayout** includes the **article** **TextView** and is completely inside the **ScrollView**.
  5. Add the **android:orientation="vertical"** attribute to the **LinearLayout** in order to set the orientation of the **LinearLayout** to vertical. The **LinearLayout** within the **ScrollView** should now look like this (choose **Code > Reformat Code** to indent the view groups correctly):

```
<ScrollView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_below="@+id/article_subheading">  
  
    <LinearLayout  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:orientation="vertical">  
  
        <TextView  
            android:id="@+id/article"  
            android:layout_width="wrap_content"  
            android:layout_height="wrap_content"  
            android:autoLink="web"  
            android:lineSpacingExtra="5sp"  
            android:text="@string/article_text" />  
  
    </LinearLayout>  
</ScrollView>
```

6. Move the **article\_subheading** **TextView** to a position inside the **LinearLayout** above the

```
article TextView.
```

7. Remove the `android:layout_below="@+id/article_heading"` attribute from the `article_subheading` `TextView`. Since this `TextView` is now within the `LinearLayout`, this attribute would conflict with the `LinearLayout` attributes.
8. Change the `ScrollView` layout attribute from  
 `android:layout_below="@+id/article_subheading" to`  
 `android:layout_below="@+id/article_heading" .` Now that the subheading is part of the `LinearLayout`, the `ScrollView` must be placed below the heading, not the subheading.
9. Run the app.

Swipe up and down to scroll the article, and notice that the subheading now scrolls along with the article while the heading stays in place.

## Solution code

## Coding challenge

Add another UI element — a `Button` — to the `LinearLayout` view group that is contained within the `ScrollView`. Make the `Button` appear below the article. The user would have to scroll to the end of the article to see the button. Use the text “Add Comment” for the `Button`, for users to click to add a comment to the article. For this challenge, there is no need to create a button-handling method to actually add a comment; it is sufficient to just place the `Button` element in the proper place in the layout.

The screenshot shows a smartphone screen with a blue header bar. In the top left corner of the header are four white icons: a triangle, an exclamation mark, a document, and a circle. In the top right corner are three signal strength icons and the time '10:36'. Below the header is a dark blue section containing the title 'Scrolling Text' in white. The main content area has a white background. At the top of this area is a dark blue box containing the title 'Beatles Anthology Vol. 1' in large white font. Below this box is a paragraph of text in gray. Further down is a larger block of text in gray, followed by another block of text in gray. At the bottom of the screen is a light gray button with the text 'ADD COMMENT' in black. A black navigation bar at the very bottom contains three white icons: a triangle pointing left, a circle, and a square.

was first recorded in 1962 by the Isley Brothers). A film of the performance shows the Queen smiling at John's remark.

***Leave My Kitten Alone*** – One of the lost Beatle songs recorded during the “Beatles For Sale” sessions but never released. This song, written by Little Willie John, Titus Turner, and James McDougal, was a 1959 R&B hit for Little Willie John and covered by Johnny Preston before the Beatles tried it and shelved it. A reference to a “big fat bulldog” may have influenced John’s “Hey Bulldog” (Yellow Submarine album), which is a similar rocker.

***One After 909*** – A song recorded for the *Let It Be* album was actually worked on way back in the beginning, six years earlier. This take shows how they did it much more slowly, with an R&B feel to it.

**ADD COMMENT**

# Summary

In this exercise you learned:

- Adding multiple [TextView](#) elements to the XML layout.
- Adding a [ScrollView](#) view group to the layout to define a scrolling view with one of the [TextView](#) elements.
- Displaying free-form text in a [TextView](#) with HTML formatting tags for bold and italics.
- Using `\n` as an end-of-line character in free-form text to keep a paragraph from running into the next paragraph.
- Using the `android:autoLink="web"` attribute to make web links in the text clickable.
- Extracting string values into string names in the `strings.xml` file.
- Adding a [LinearLayout](#) view group within a [ScrollView](#) in order to scroll several [TextView](#) elements together.
- Challenge: Adding a [Button](#) to the [ScrollView](#) to scroll along with the [TextViews](#).

# Resources

## Developer Documentation:

- [TextView](#)
- [ScrollView](#)
- [String Resources](#)
- [View](#)
- [Relative Layout](#)

## Other:

- Android Developers Blog: [Linkify your Text!](#)
- Codepath: [Working with a TextView](#)

# 1.4 P: Learning About Available Resources

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Explore the official Android documentation](#)
- [Task 2: Use project templates](#)
- [Task 3: Learn from example code](#)
- [Task 4: Many more resources](#)
- [Summary](#)
- [Resources](#)

In this practical you will:

- Explore some of the many resources available to Android developers of all levels.
- Add a home screen icon to your Word List app that will launch it when tapped.

## What you should already KNOW

For this practical you should be familiar with:

- The basic workings of Android Studio.

## What you will LEARN

- Where to find developer resources within Android Studio, the [official Android developer documentation](#), and elsewhere on the Internet.

## What you will DO

- Explore and use Android developer resources.
- Use developer resources to figure out how to add an icon to the home screen of your device. When this icon is clicked, your app launches.

# App Overview

You will use the existing Hello Toast app and add a launcher icon to it.

## Task 1. Explore the official Android developer documentation

You can find the official Android developer documentation at:

<http://developer.android.com/index.html>

This documentation contains a wealth of information that is kept current by Google.

### 1.1. Explore the official Android documentation

1. Go to <http://developer.android.com/index.html>.
2. At the top of the page, look for the **Design**, **Develop**, and **Distribute** links. Follow each of the links and familiarize yourself with the navigation structure.
  - **Design** is all about Material Design, which is a conceptual design philosophy that outlines how apps should look and work on mobile devices. Scroll to the bottom of the landing page for links to resources such as sticker sheets and [color palettes](#).
  - **Develop** is where you can find API information, reference documentation, tutorials, tool guides, and code samples. You can use the site navigation or search to find what you need.
  - **Distribute** is about everything that happens after you've written your app: putting it on the Play Store, growing your user base, and [earning money](#).
3. Use search or navigate the documentation to complete the following tasks:
  - Add a launcher icon to the Word List app. See [Launcher Icons](#) to learn more about how to design effective launcher icons.
  - Learn how to monitor your app's resource usage in Android Studio.

#### Solution:

Use [Image Asset Studio](#) to create and add a launcher icon.

1. Open the Hello Toast app in Android Studio.
2. Right-click the **res** folder of your project and select **New > Image Asset** from menu.  
This opens the Image Asset Studio window, where you can create a text icon, choose from available clipart, or add your own custom icon.

Note that the panel on the top-left is scrollable; scroll to see additional customizations.

### Add a custom text icon:

1. Change the **icName** of the icon to ic\_launcher\_text, if you don't want to overwrite the default Android ic\_launcher icon that comes with your project.
2. In the **Asset Type** row, select Text.
3. Type "Toast Me!" into the text box.
4. Experiment with adjusting the font.
5. Scroll down and change font and background colors.
6. Click **Next**.
7. The **Confirm Icon Path** window shows how an icon with your specified text will be created for each resolution and the default storage location and path in your app.
8. Click **Finish**.
9. Go to the **res/mipmap** folder. It now contains your new icon, a default version at the top level, and size-adjusted versions for different resolutions.
10. To use the new icon, open the Android Manifest. Change the android icon line to from referencing ic\_launcher to ic\_launcher\_text.

```
        android:icon="@mipmap/ic_launcher_text"
```

11. Run your app.
12. After the app has launched, go to the home screen and open the list of apps.
13. Scroll and you should see your icon listed along with the other installed apps.

### Add clipart icon:

Follow the previous steps except:

1. Change the **icName** to ic\_launcher\_clipart.
2. Choose **Clip Art** as the **Asset Type**.
3. In the Clip Art row, click the button showing the current icon, the default Android.
4. Choose an icon from the popup window of clip art.

### Add a custom icon:

Follow the previous steps except:

1. Change the **icName** to ic\_launcher\_image.
2. Choose **Image** as the **Asset Type**.
3. In the **Path** row, choose an image. This can be an image that you've added to your project or an image in your computer.

The [Android Monitor page](#) discusses how to monitor your app's performance. Android Studio and your mobile device offer tools to measure your app's memory use, GPU, CPU, and Network performance. App crashes are often related to memory leaks, that is, your app allocates memory and does not release it. Potentially your app can, eventually use up all the available memory on your device. You can use the Memory Monitor that comes with Android Studio to observe how your app uses memory.

1. In Android Studio, at the bottom of the window, click the **Android Monitor** tab. By default this opens on logcat.
2. Click the **Monitors** tab next to logcat. Scroll or make the window larger to see all four monitors: Memory, CPU, Networking, and GPU.
3. Run your app and interact with it. The monitors update to reflect the app's use of resources.

App performance is a big topic on its own and you will learn about it in a later lesson.

## Task 2. Use project templates

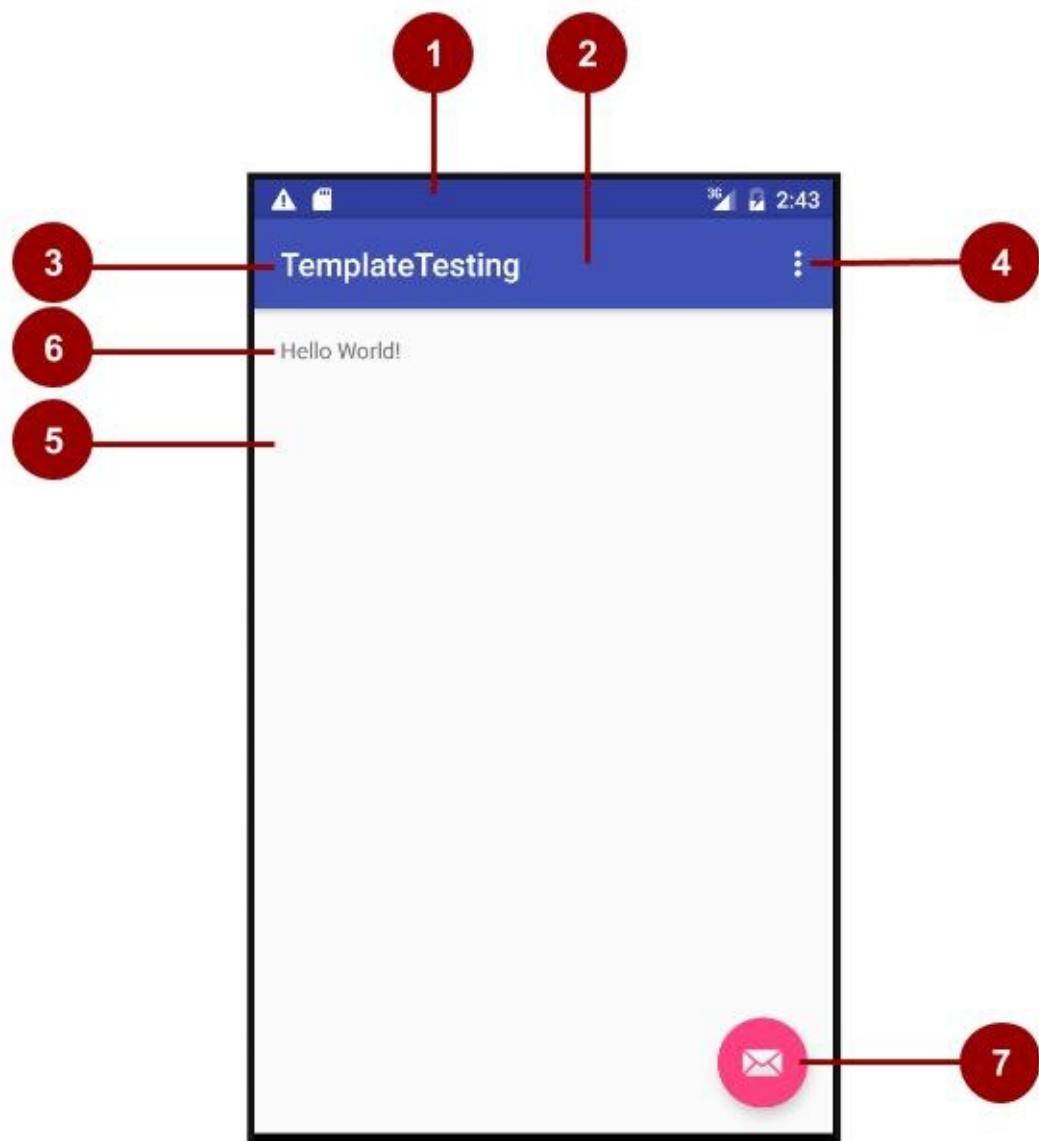
Android Studio provides templates for common and recommended app and activity designs. Using templates saves time, and helps you follow design best practices.

Each template incorporates an skeleton activity and user interface. You've already used the Empty Activity template. The Basic Activity template has more features and incorporates recommended app features, such as the options menu.

### 2.1. Explore the Basic Activity architecture

The Basic Activity template is a versatile template that is straightforward to use.

1. Create a new project with the Basic Activity template.
2. Build and run the app.
3. Identify the labelled parts on the screenshot and table below. Find their equivalents on your device or emulator screen.



**Architecture of the Basic Activity template**

#	UI Description	Code reference
1	<b>Status bar</b> This bar is provided and controlled by the Android system.	Not visible in the template code. It's possible to access it from your activity. For example, you can <a href="#">hide the status bar</a> , if necessary.
2	<b>AppBarLayout &gt; Toolbar</b> App bar (also called Action bar) provides visual structure, standardized visual elements, and navigation. For backwards compatibility, the <a href="#">AppBarLayout</a> in the template embeds a <a href="#">Toolbar widget</a> with the same functionality. <a href="#">ActionBar class Challenge: App Bar Tutorial</a>	<b>activity_main.xml</b> Look for android.support.v7.widget.Toolbar inside android.support.design.widget.AppBarLayout. Change the toolbar to change the appearance of its parent, the app bar.
3	<b>Application name</b> This is derived from your package name, but can be anything you choose.	<b>AndroidManifest.xml</b> android:label="@string/app_name"
4	<b>Options menu</b> Menu items for the activity, as well as global options, such as "Search". Your app menu items go into this menu.	<b>MainActivity.java</b> onOptionsItemSelected() implements what happens when a menu item is selected. <b>res &gt; menu &gt; menu_main.xml</b> Resource that specifies the menu items for the options menu.
5	<b>CoordinatorLayout</b> <a href="#">CoordinatorLayout</a> is a feature-rich layout that provides mechanisms for views to interact. Your app's user interface goes inside this view group.	<b>activity_main.xml</b> Notice that there are no views specified in this layout; rather, it includes another layout with <code>&lt;include layout="@layout/content_main"/&gt;</code> where the views are specified. This separates system views from the views unique to your app.
6	<b>TextView</b> In the example, use to display "Hello World". Replace this with the views for your app.	<b>content_main.xml</b> All your app's views are defined in this file.
7	Floating Action button (FAB)	<b>activity_main.xml</b> MainActivity.java > onCreate has a stub that sets an onClick listener on the FAB.

4. Also inspect the corresponding Java code and XML configuration files.

Being familiar with the Java source code and XML files will help you extend and customize this template for your own needs.

See [Accessing Resources](#) for details on the XML syntax for accessing resources.

5. After you understand the template code, try the following:
  - Change the color of the app bar (toolbar).
  - Look at the styles associated with the app bar (toolbar).
  - Change the name of your app that's displayed in the app bar (toolbar).

## 2.2. Explore how to add an activity using templates

For the practicals so far, you've used the Empty Activity and Basic Activity templates.vlh later lessons, the templates you use will vary, depending on the task.

These activity templates are also available from inside your project, so that you can add more activities to your app after the initial project setup. (You will learn more about this this in a later chapter.)

1. Create a new project or choose an existing project.
2. In your project, in the Android view, in the **java** folder, right-click the folder with your Java files).
3. Choose **New > Activity > Gallery**.
4. Add one of those **activities**, for example, the Navigation Drawer Activity. Find the layout files for the Navigation Drawer Activity and display them in **Design**.

## Task 3. Learn from example code

Android Studio, as well as the Android documentation provide many code samples that you can study, copy, and incorporate with your projects.

### 3.1. Android code samples

You can explore hundreds of code samples directly from Android Studio.

1. In Android Studio, choose **File > New > Import Sample**.
2. Browse the samples.
3. Look at the Description and Preview tabs to learn more about each sample.
4. Choose a sample and click **Next**.
5. Accept the defaults and click **Finish**.
6. Run the sample.

### 3.2. Use the SDK Manager to install offline documentation

Installing Android Studio also installs essentials of the Android SDK (Software Development Kit). However, additional libraries and documentation are available, and you can install them using the SDK Manager.

1. Choose **Tools > Android > SDK Manager**. This opens the Default Preferences settings.
2. In the left-hand navigation, find and open the settings for **Android SDK**.
3. Click **SDK Platforms** in the settings window. You can install additional versions of the Android system from here.
4. Click on **SDK Update Sites**. Android Studio checks the listed and checked sites regularly for updates.
5. Click on the **SDK Tools** tab. Here you can install additional SDK Tools that are not installed by default, as well as an offline version of the Android developer documentation. This gives you access to documentation even when you are not connected to the internet.
6. Check "Documentation for Android SDK", click **Apply**, and follow the prompts.
7. Navigate to the **Android/sdk** directory and open the **docs** folder.
8. Find **index.html** and open it.

## Task 4. Many more resources

- The [Android Developer YouTube channel](#) is a great source of tutorials and tips.
- The Android team posts news and tips on the [Official Android blog](#).
- [Stack Overflow](#) is a community of millions of programmers helping each other. If you run into a problem, chances are, someone else has already posted an answer on this forum.
- And last but not least, type your questions into Google search, and the Google search engine will collect relevant results from all of these resources. For example, "What is the most popular Android OS version in India?"

## Summary

- Official Android Developer Documentation - <http://developer.android.com>
- Material Design is a conceptual design philosophy that outlines how apps should look and work on mobile devices.
- The Google Play Store is Google's digital distribution system for apps developed with the Android SDK.
- Image Asset Studio is used within Android Studio to create app launcher icons for your home screen.

- Android Monitor is used within Android Studio to monitor CPU, memory, network and even GPU behavior of your app.
- Android Studio provides templates for common and recommended app and activity designs. These templates offer working code for common use cases.
- Activities can be created from templates when you first create a project in Android Studio, or afterwards during the development of your app.
- A Floating Action Button (FAB) is a small circled icon that “floats” above the UI. It represents the promoted action in an Activity.
- Android Studio provides many code samples that you can study, copy, and incorporate with your projects.
- The Android SDK Manager provides the SDK tools, platforms, and other components you need to develop your apps. And it allows you to keep them updated.

## Resources

### Developer Documentation:

- [Official Android documentation](#)
- [Android code samples for developers](#)
- [Official Android blog](#)
- [Stack Overflow](#)

### Videos

- [Android Developer YouTube channel](#)

## 2.1 P: Create and Start Activities

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Create the TwoActivities project](#)
- [Task 2. Create and launch the second activity](#)
- [Task 3. Send data from the main activity to the second activity](#)
- [Task 4. Return data back to the main activity](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

An activity represents a single screen in your app with which your user can perform a single, focussed task such as dial the phone, take a photo, send an email, or view a map. Activities are usually presented to the user as full-screen windows, although they can also be embedded inside other activities or presented as floating windows.

An app usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when the app is launched. Each activity can then start other activities in order to perform different actions.

Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. The back stack abides to the basic "last in, first out" stack mechanism, so, when the user is done with the current activity and presses the Back button, it is popped from the stack (and destroyed) and the previous activity resumes.

Android activities are started or activated with an *intent*. Intents are asynchronous messages that request an action from one activity to another and connect those activities together at runtime. You use intents to start one activity from another and to pass data between activities.

There are two kinds of intents: *explicit* and *implicit*. An explicit intent is one in which you know the target of that intent, that is, you already know the fully-qualified class name of that specific activity. An implicit intent is one in which you do not have the name of the target

component, but have a general action to perform. In this practical you'll learn about explicit intents. You'll find out about implicit intents in a later practical.

## What you should already KNOW

From the previous chapter you should be familiar with:

- How to create and run apps in Android Studio.
- How to create and edit UI elements with the graphical Layout Editor, or directly in an XML layout file.
- How to add onClick functionality to a button.

## What you will LEARN

- How to create a new activity in Android studio.
- How to define parent and child activities for "Up" navigation.
- How to start activities with explicit intents.
- How to pass data between activities with intent extras.

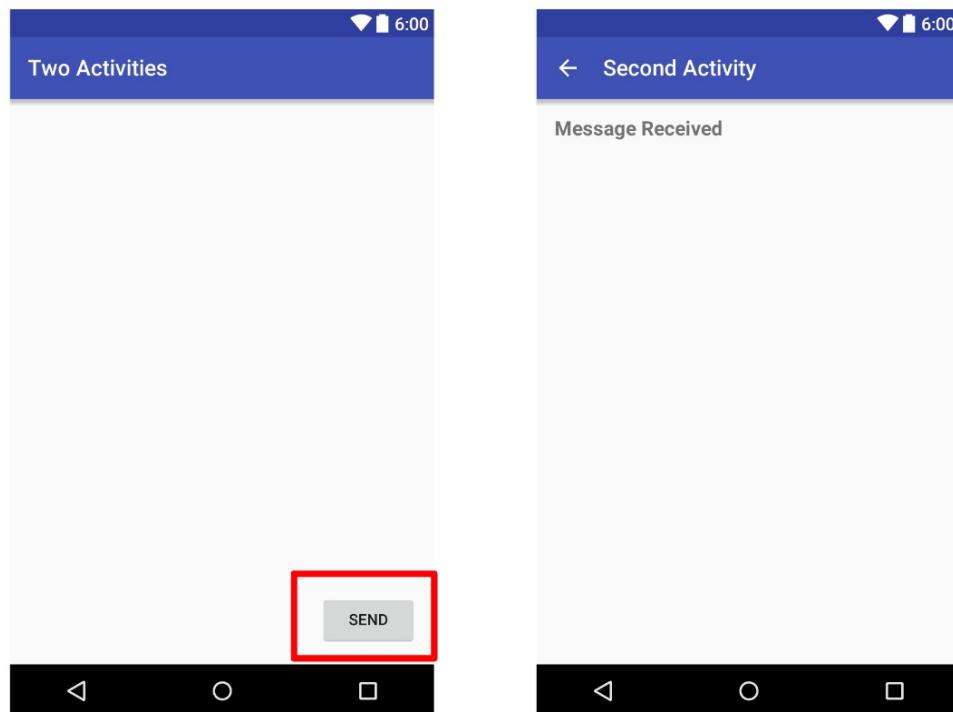
## What you will DO

- Create a new Android app with two activities.
- Pass some data (a string) from the main activity to the second using an intent, and display that data in the second activity.
- Send a second different bit of data back to the main activity, also using intents.

## App Overview

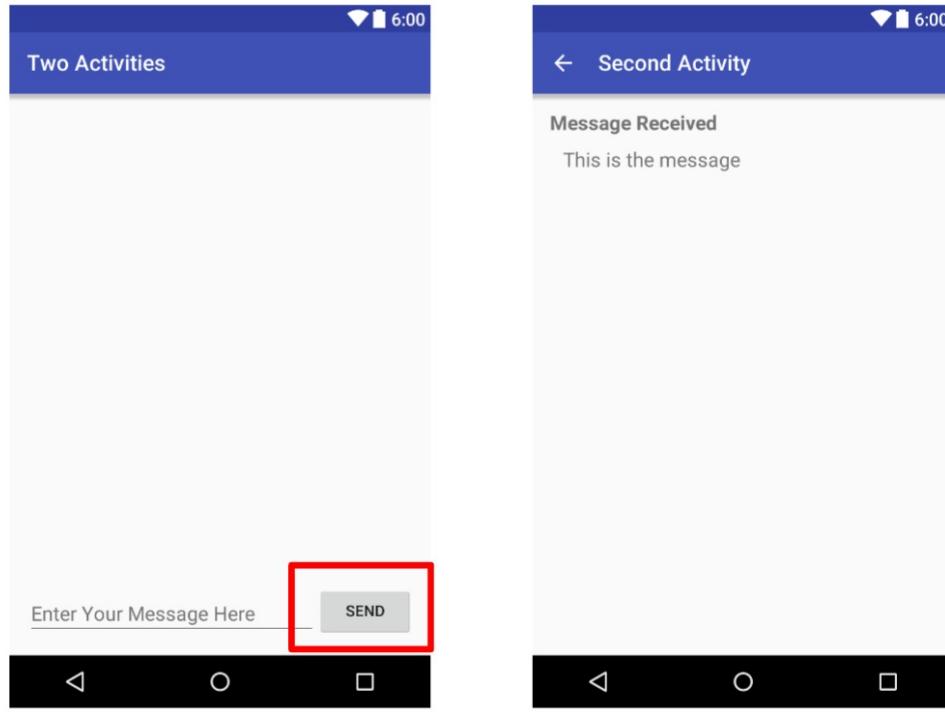
In this chapter you will create and build an app called TwoActivities that, unsurprisingly, contains two activities. You'll build this app in three stages.

In the first stage, you'll create an app whose main activity contains only one button (Send). When the user clicks this button, your main activity uses an intent to start the second activity.



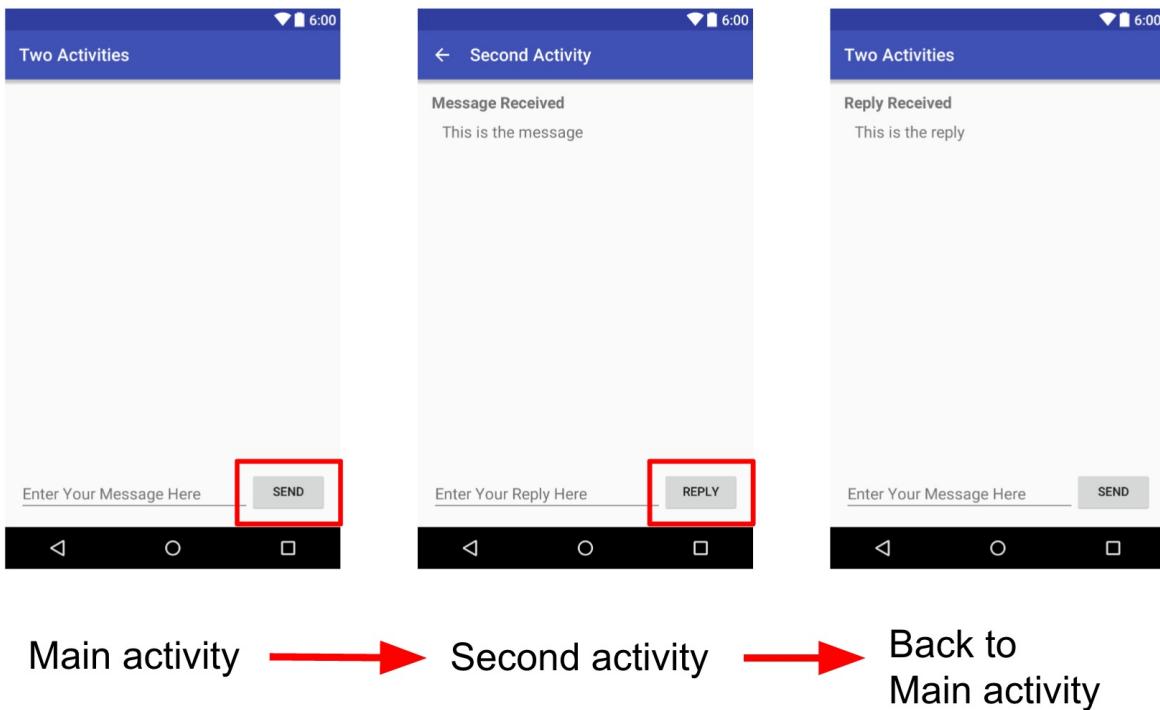
Main activity → Second activity

In the second stage, you'll add an EditText view to the main activity. The user enters a message and clicks Send. The main activity uses an intent to both start the second activity, and to send the user's message to the that activity. The second activity displays the message it received.



Main activity → Second activity

In final stage of the TwoActivities app you'll add an EditText view and a Reply button to the second activity. The user can now type a reply message and click Reply, and the reply is displayed on the main activity. You'll also use an intent here to pass the reply message back from the second activity to the main activity.



## Task 1. Create the TwoActivities project

In this task you'll set up the initial project with a main activity, define the layout, and define a skeleton method for the `onclick` button event.

### 1.1 Create the TwoActivities project

1. Start Android Studio and create a new Android Studio project.  
Call your application "Two Activities" and change the company domain to "android.example.com." Choose the same Minimum SDK that you did in the previous projects.
2. Choose **Empty Activity** for the project template. Click **Next**.
3. Accept the default activity name (MainActivity). Make sure the **Generate Layout file** box is checked. Click **Finish**.

### 1.2 Define the layout for the main activity

1. Open `res/layout/activity_main.xml`. If you are not already in the Layout Editor, click the Design tab at the bottom of the screen.
2. Delete the TextView that says "Hello World."
3. Add a Button to the layout in any position.

4. Switch to the XML Editor (click the Text tab) and modify these attributes in the Button:

Attribute	Value
android:id	"@+id/button_main"
android:layout_width	wrap_content
android:layout_height	wrap_content
android:layout_alignParentRight	"true"
android:layout_alignParentBottom	"true"
android:layout_alignParentEnd	"true"
android:text	"Send"
android:onClick	"launchSecondActivity"

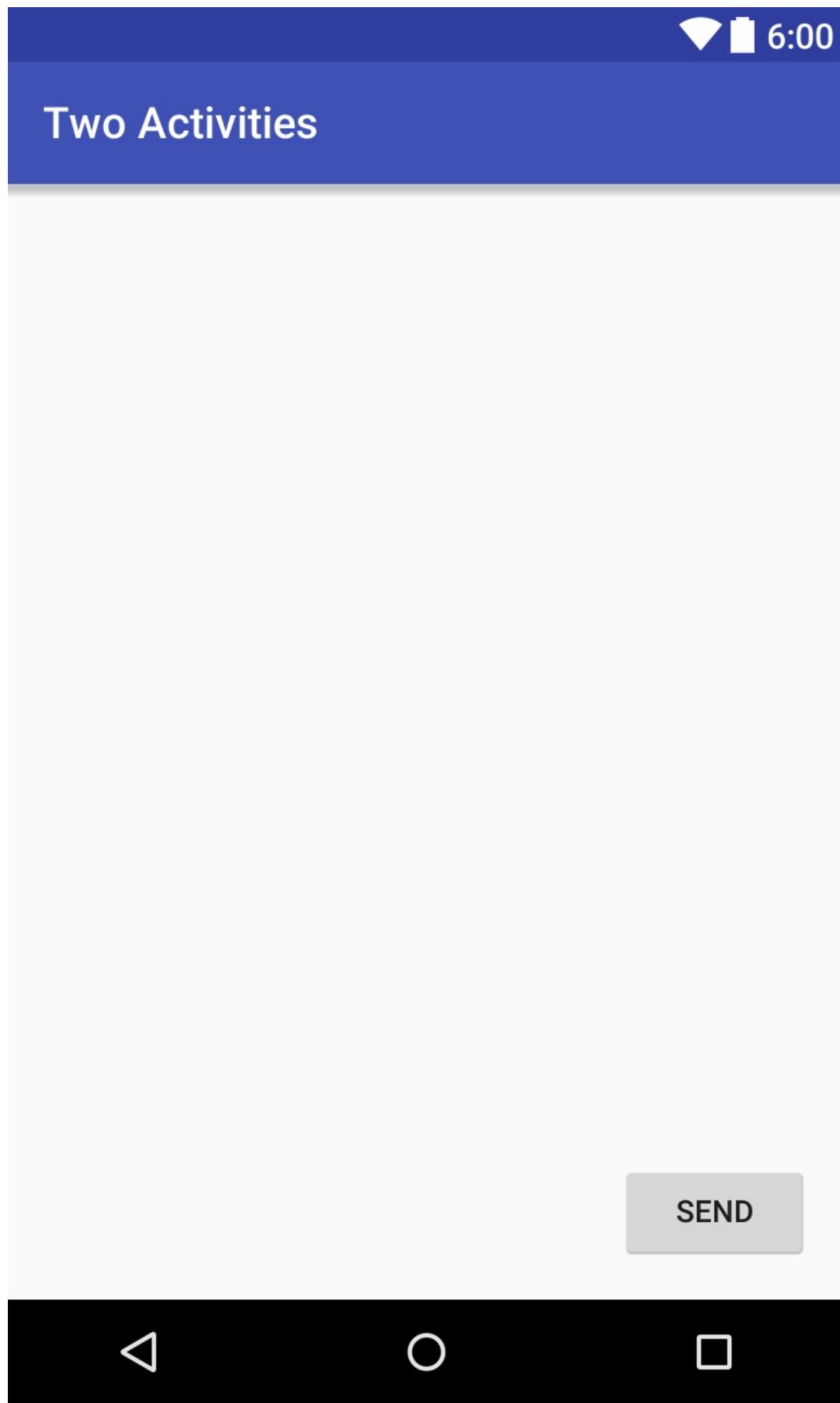
You may see an error that "Method launchSecondActivity is missing in MainActivity."

You can ignore this error for now. We'll add it in the next task.

5. Place the cursor on the word "Send".
6. Press **Alt-Enter (Option-Enter on the Mac)** and select **Extract string resources**.
7. Set the **Resource name** to `button_main` and click **OK**.

This creates a string resource in the values/res/string.xml file, and the string in your code is replaced with a reference to that string resource.

8. Preview the layout of the main activity using the Layout Editor. The layout should look like this:



**Solution Code:**

```

<?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: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.android.twoactivities.MainActivity">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_main"
        android:id="@+id/button_main"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:onClick="launchSecondActivity"/>
</RelativeLayout>

```

## 1.3 Define the button action

In this task, implement the onClick method you defined in the layout.

1. In the XML Editor, place the cursor on the word "launchSecondActivity" .
2. Press **Alt-Enter (Option-Enter on the Mac)** and select **Create 'launchSecondActivity(View)' in 'MainActivity'.**

The MainActivity.java file opens, and Android Studio generates a skeleton method for the onClick handler.

3. Inside `launchSecondActivity` , add a log statement that says "Button Clicked!"

```
Log.d(LOG_TAG, "Button clicked!");
```

`LOG_TAG` will show as red. You'll add the definition for that variable in a later step.

4. Place the cursor on the word "Log" and press **Alt-Enter (Option-Enter on the Mac)**.  
Android Studio adds an import statement for `android.util.Log`.
5. At the top of the class, add a constant for the `LOG_TAG` variable:

```
private static final String LOG_TAG = MainActivity.class.getSimpleName();
```

- This constant uses the name of the class itself as the tag.
6. Run your app. When you click the "Send" button you will see the "Button Clicked!" message in the Android Monitor (logcat). If there's too much output in the monitor, type MainActivity into the search box and the log will only show lines that match that tag.

### Solution Code:

```
package com.example.android.twoactivities;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;

public class MainActivity extends AppCompatActivity {
    private static final String LOG_TAG = MainActivity.class.getSimpleName();

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

    public void launchSecondActivity(View view) {
        Log.d(LOG_TAG, "Button clicked!");
    }
}
```

## Task 2. Create and launch the second activity

Each new activity you add in your project has its own layout and Java files, separate from those of the main activity. They also have their own <activity> elements in the Android manifest.

All the activities in your app are only loosely connected with each other. However, you can define an activity as a parent of another activity in the AndroidManifest.xml file. This parent-child relationship enables Android to add navigation hints such as left-facing arrows in the title bar for each activity.

Activities communicate with each other (both in the same app and across different apps) with *intents*. There are two types of intents, explicit and implicit. An explicit intent is one in which you know the target of that intent, that is, you already know the fully-qualified class name of that specific activity. An implicit intent is one in which you do not have the name of the target component, but have a general action to perform. You'll learn about implicit intents in a later practical.

In this task you'll add a second activity to our app, with its own layout. You'll modify the Android manifest to define the main activity as the parent of the second activity. Then you'll modify the `onClick` event method in the main activity to include an intent that launches the second activity when you click the button.

## 2.1 Create the second activity

1. Choose **File > New > Activity > Empty Activity**.
2. Name the new activity "SecondActivity." Make sure **Generate Layout File** is checked, and layout name will be filled in as `activity_second`.
3. Click **Finish**. Android Studio adds both a new activity layout (`activity_second`) and a new Java file (`SecondActivity`) to your project for the new activity. It also updates the Android manifest to include the new activity.

## 2.2 Modify the Android manifest

1. Open `manifests/AndroidManifest.xml`.
2. Find the `<activity>` element that Android Studio created for the second activity.

```
<activity android:name=".SecondActivity"></activity>
```

3. Add these attributes to the `<activity>` element:

Attribute	Value
<code>android:label</code>	"Second Activity"
<code>android:parentActivityName</code>	".MainActivity"

The `label` attribute adds the title of the activity to the action bar.

The `parentActivityName` attribute indicates that the main activity is the parent of the second activity. This parent activity relationship is used for "up" navigation within your app. By defining this attribute, the action bar for the second activity will appear with a left-facing arrow to enable the user to navigate "up" to the main activity.

4. Place the cursor on "Second Activity" and press **Alt-Enter (Option-Enter on the Mac)**.
5. Choose **Extract string resource**, name the resource "activity2\_name", and click **OK**. Android Studio adds a string resource for the activity label.
6. Add a `<meta-data>` element inside the `<activity>` element for the second activity. Use these attributes:

Attribute	Value
android:name	"android.support.PARENT_ACTIVITY"
android:value	"com.example.android.twoactivities.MainActivity"

The <meta-data> element provides additional arbitrary information about the activity as key-value pairs. In this case these attributes accomplish the same thing as the android:parentActivityName attribute -- they define a relationship between two activities for the purpose of up navigation. These attributes are required for older versions of Android. android:parentActivityName is only available for API 16 and higher.

### Solution Code:

```

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

    <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=".SecondActivity"
            android:label="@string/activity2_name"
            android:parentActivityName=".MainActivity">
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value="com.example.android.twoactivities.MainActivity" />
        </activity>
    </application>
</manifest>

```

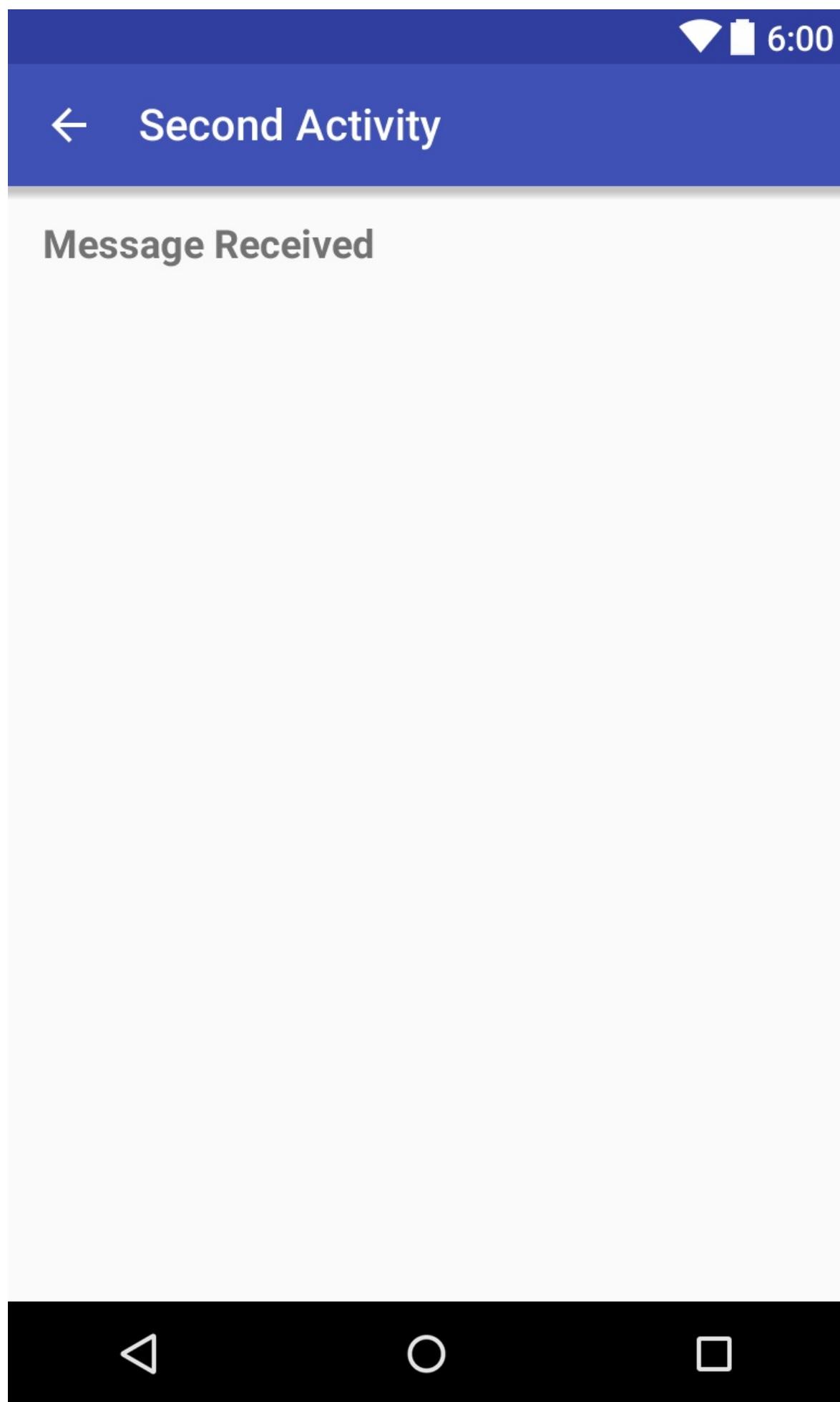
## 2.3 Define the layout for the second activity

1. Open `res/layout/activity_second.xml`.
2. Add a TextView ("Plain Textview" in the Layout Editor). Give the TextView these attributes:

Attribute	Value
android:id	"@+id/text_header"
android:layout_width	wrap_content
android:layout_height	wrap_content
android:layout_marginBottom	"@dimen/activity_vertical_margin"
android:text	"Message Received"
android:textAppearance	?android:attr/textAppearanceMedium
android:textStyle	"bold"

The value of `textAppearance` is a special Android theme attribute that defines basic font styles for small, medium, and large fonts. You'll learn more about themes in a later lesson.

3. Extract the "Message Received" string into a resource named `text_header`.
4. Preview the layout in the Layout Editor. The layout should look like this:



**Solution Code:**

```

<?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: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=".SecondActivity">

    <TextView
        android:id="@+id/text_header"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="@dimen/activity_vertical_margin"
        android:text="@string/text_header"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:textStyle="bold" />
</RelativeLayout>

```

## 2.4 Add an intent to the main activity

In this task you'll add an explicit intent to the main activity. This intent is used to activate the second activity when the Send button is clicked.

1. Open the Java file for `MainActivity` (`java/com.example.android.twoactivities/MainActivity`).
2. Create a new intent in the `launchSecondActivity()` method.

The Intent constructor takes two arguments for an explicit intent: an application `Context` and the specific component that will receive that intent. Here you should use `this` as the context, and `SecondActivity.class` as the specific class.

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

3. Place the cursor on Intent and press **Alt-Enter (Option-Enter on the Mac)** to add an import for the Intent class.
4. Call the `startActivity()` method with the new intent as the argument.

```
startActivity(intent);
```

5. Run the app.

When you click the Send button the second activity starts and appears on the screen. To get back to the main activity, click the Android Back button at the bottom left of the screen. Or you can use the left arrow at the top of the second activity to return to the main activity.

**Challenge:** What happens if you remove the  `android:parentActivityName` and the `<meta-data>` elements from the manifest? Make this change and run your app.

## Task 3. Send data from the main activity to the second activity

In the last task you added an explicit intent to the main activity that activated the second activity. You can also use intents to send data from one activity to another.

In this task you'll modify the explicit intent in the main activity to include additional data (in this case, a user-entered string) in the intent extras. You'll then modify the second activity to get that data back out of the intent extras and display it on the screen.

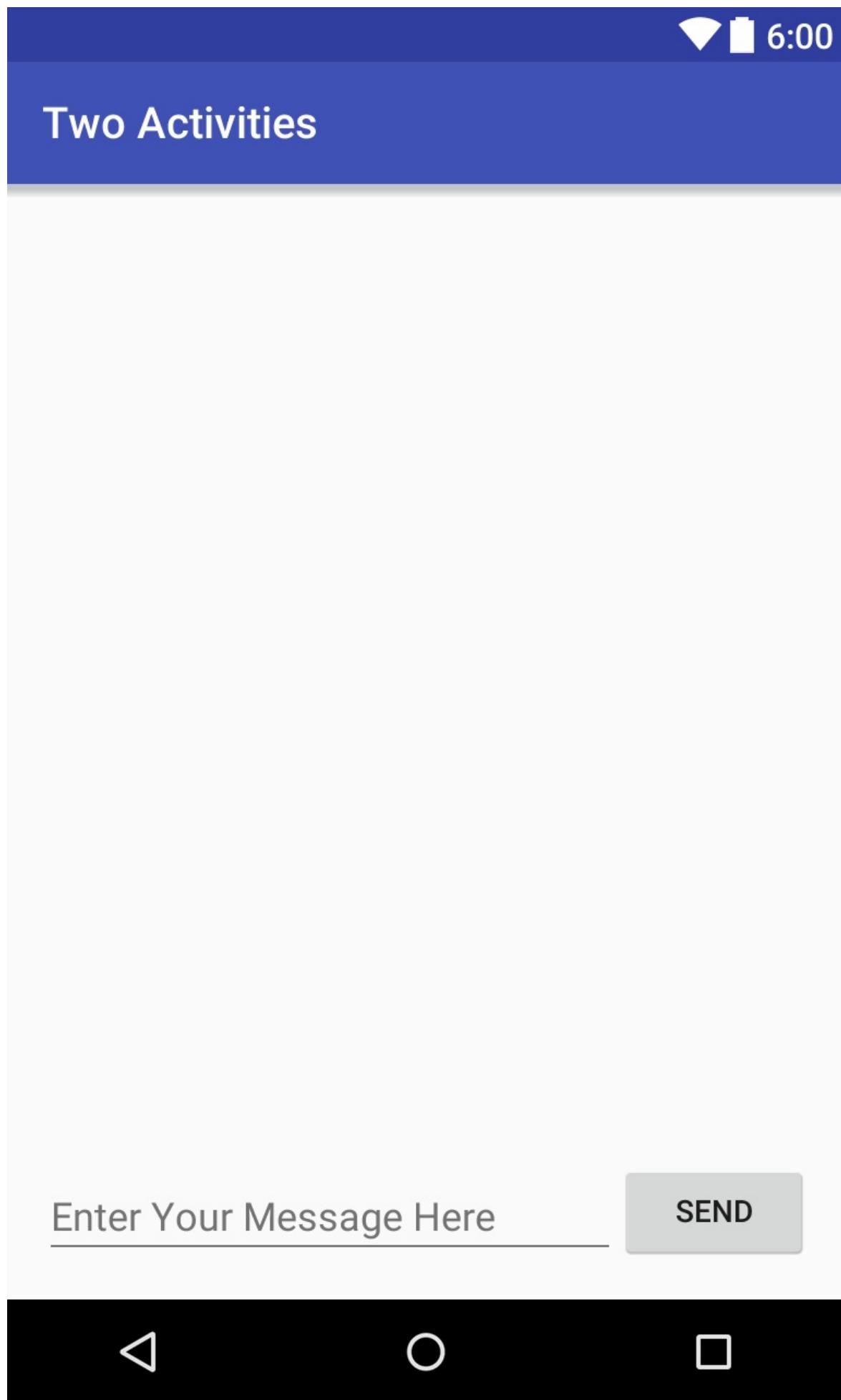
### 3.1 Add an EditText to the main activity layout

1. Open `res/layout/activity_main.xml`.
2. Add an `EditText` view (Plain Text in the Layout Editor.) Give the `EditText` these attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/editText_main"</code>
<code>android:layout_width</code>	<code>match_parent</code>
<code>android:layout_height</code>	<code>wrap_content</code>
<code>android:layout_toLeftOf</code>	<code>"@+id/button_main"</code>
<code>android:layout_toStartOf</code>	<code>"@+id/button_main"</code>
<code>android:layout_alignParentBottom</code>	<code>"true"</code>
<code>android:hint</code>	<code>"Enter Your Message Here"</code>

3. Extract the "Enter Your Message Here" string into a resource named `editText_main`.

The new layout for the main activity looks like this:



**Solution Code:**

```
<?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: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.android.twoactivities.MainActivity">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_main"
        android:id="@+id/button_main"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:onClick="launchSecondActivity"/>

    <EditText
        android:id="@+id/editText_main"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_toLeftOf="@+id/button_main"
        android:layout_toStartOf="@+id/button_main"
        android:hint="@string/editText_main" />
</RelativeLayout>
```

### 3.2 Add a string to the main activity's intent extras

Your Intent object can pass data to the target activity in two ways: in the data field, or in the extras. The intent's data is a URI indicating the specific data to be acted on. If the information you want to pass to an activity through an intent is not a URI, or you have more than one piece of information you want to send, you can put that additional information into the extras instead.

The intent **extras** are key/value pairs in a **Bundle**. A *bundle* is a collection of data, stored as key/value pairs. To pass information from one activity to another, you put keys and values into the intent extra bundle from the sending activity, and then get them back out again in the receiving activity.

1. Open `java/com.example.android.twoactivities/MainActivity` .

2. Add a public constant at the top of the class to define the key for the intent extra:

```
public static final String EXTRA_MESSAGE =  
"com.example.android.twoactivities.extra.MESSAGE";
```

3. Add a private variable at the top of the class to hold the EditText object. Import the EditText class.

```
private EditText mMessageEditText;
```

4. In the onCreate() method, use findViewById to get a reference to the EditText instance and assign it to that private variable:

```
mMessageEditText = (EditText) findViewById(R.id.editText_main);
```

5. In the launchSecondActivity() method, just under the new intent, get the text from the EditText as a string:

```
String message = mMessageEditText.getText().toString();
```

6. Add that string to the intent as an extra with the EXTRA\_MESSAGE constant as the key and the string as the value:

```
intent.putExtra(EXTRA_MESSAGE, message);
```

### Solution Code:

```
package com.example.android.twoactivities;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.EditText;

public class MainActivity extends AppCompatActivity {
    private static final String LOG_TAG = MainActivity.class.getSimpleName();

    public static final String EXTRA_MESSAGE =
"com.example.android.twoactivities.extra.MESSAGE";

    private EditText mMessageEditText;

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

        mMessageEditText = (EditText) findViewById(R.id.editText_main);
    }

    public void launchSecondActivity(View view) {
        Log.d(LOG_TAG, "Button clicked!");

        Intent intent = new Intent(this, SecondActivity.class);
        String message = mMessageEditText.getText().toString();

        intent.putExtra(EXTRA_MESSAGE, message);
        startActivity(intent);
    }
}
```

### 3.3 Add a TextView for the message to the layout of the second activity

1. Open `res/layout/activity_second.xml`.
2. Add a second TextView. Give the TextView these attributes:

Attribute	Value
android:id	"@+id/text_message"
android:layout_width	wrap_content
android:layout_height	wrap_content
android:layout_below	"@+id/text_header"
android:layout_marginLeft	"@dimen/activity_horizontal_margin"
android:layout_marginStart	"@dimen/activity_horizontal_margin"
android:textSize	"?android:attr/textAppearanceMedium"

3. Delete the android:text attribute (if it exists).

The new layout for the second activity looks the same as it did in the previous task, because the new TextView does not (yet) contain any text, and thus does not appear on the screen.

### Solution Code:

```

<?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: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.android.twoactivities.SecondActivity">

    <TextView
        android:id="@+id/text_header"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/text_header"
        android:layout_marginBottom="@dimen/activity_vertical_margin"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:textStyle="bold"/>

    <TextView
        android:id="@+id/text_message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/text_header"
        android:layout_marginLeft="@dimen/activity_horizontal_margin"
        android:layout_marginStart="@dimen/activity_horizontal_margin"
        android:textAppearance="?android:attr/textAppearanceMedium" />

</RelativeLayout>

```

### 3.4 Modify the second activity to get the extras and display the message

1. Open `java/com.example.android.twoactivities/SecondActivity` .
2. In the `onCreate()` method, get the intent that activated this activity:

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

3. Get the string containing the message from the intent extras using the

```
MainActivity.EXTRA_MESSAGE static variable as the key:
```

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

4. Use `findViewById` to get a reference to the `TextView` for the message from the layout (you may need to import the `TextView` class):

```
TextView textView = (TextView) findViewById(R.id.text_message);
```

5. Set the text of that `TextView` to the string from the intent extra:

```
textView.setText(message);
```

6. Run the app. When you type a message in the main activity and click Send, the second activity is launched and displays that message.

#### Solution Code:

```
package com.example.android.twoactivities;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

public class SecondActivity extends AppCompatActivity {

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

        Intent intent = getIntent();
        String message =
            intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
        TextView textView = (TextView) findViewById(R.id.text_message);
        textView.setText(message);
    }
}
```

## Task 4. Return data back to the main activity

Now that you have an app that launches a new activity and sends data to it, the final step is to return data from the second activity back to the main. You'll also use intents and intent extras for this task.

### 4.1 Add an EditText and a Button to the second activity layout

1. Copy the EditText and Button from the main activity layout file and paste them into the second layout.
2. In the `activity_second.xml` file, modify the attribute values for both the Button and EditText views. Use these values:

Old Attribute (Button)	New Attribute (Button)
android:id="@+id/button_main"	android:id="@+id/button_second"
android:onClick="launchSecondActivity"	android:onClick="returnReply"
android:text= "@string/button_main"	android:text= "@string/button_second"
Old Attribute (EditText)	New Attribute (EditText)
android:id="@+id/editText_main"	android:id="@+id/editText_second"
android:layout_toLeftOf="@+id/button_main"	android:layout_toLeftOf="@+id/button_second"
android:layout_toStartOf="@+id/button_main"	android:layout_toStartOf="@+id/button_second"
android:hint= "@string/editText_main"	android:hint= "@string/editText_second"

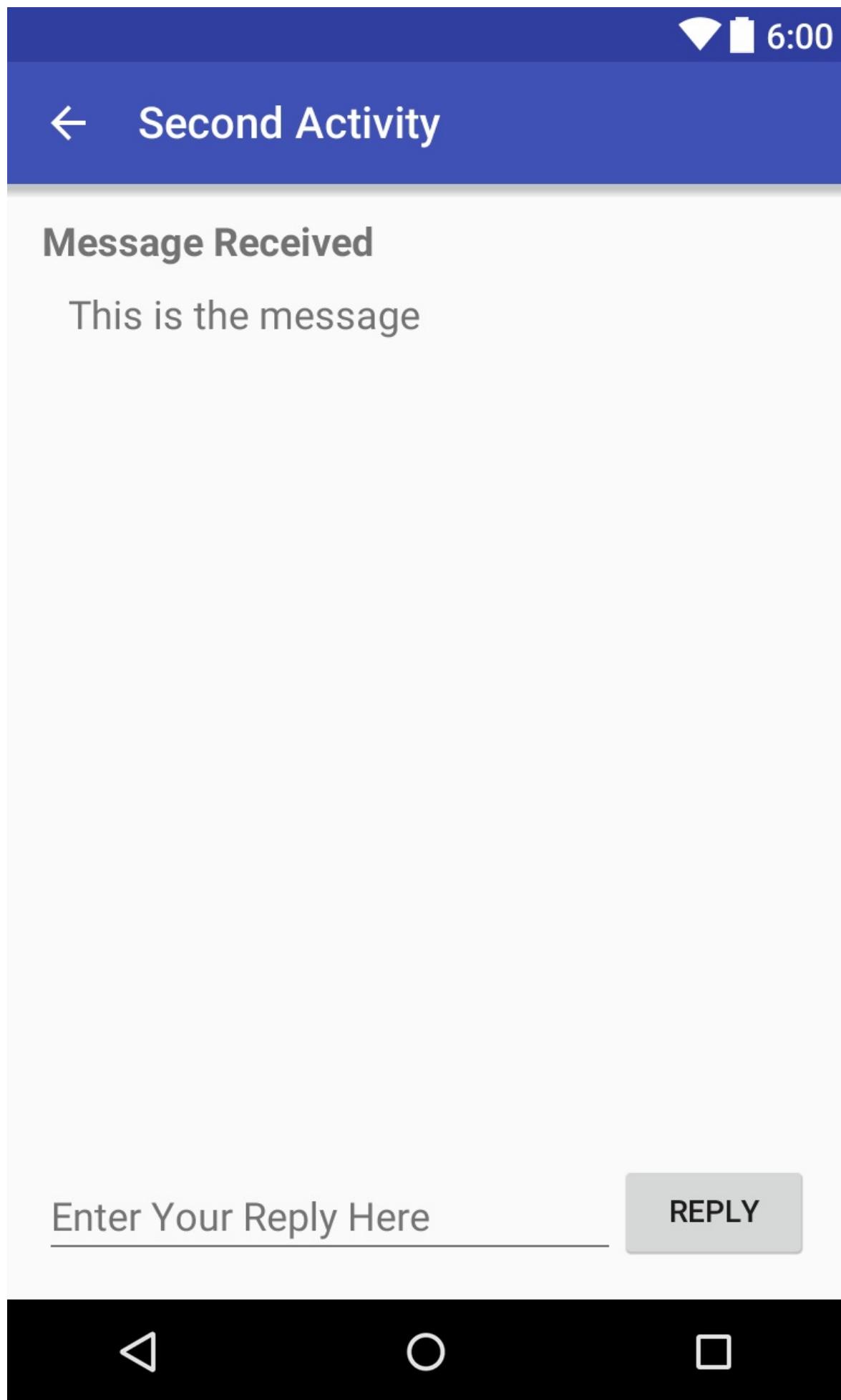
3. Open `res/values/strings.xml` and add string resources for the button text and the hint in the EditText:

```
<string name="button_second">Reply</string>
<string name="editText_second">Enter Your Reply Here</string>
```

4. In the XML layout editor, place the cursor on `"returnReply"`, press **Alt-Enter (Option-Enter on the Mac)** and select **Create 'launchSecondActivity(View)' in 'SecondActivity'**.

The SecondActivity.java file opens, and Android Studio generates a skeleton method for the onClick handler.

The new layout for the second activity looks like this:



**Solution Code:**

```
<?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: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.android.twoactivities.SecondActivity">

    <TextView
        android:id="@+id/text_header"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/text_header"
        android:layout_marginBottom="@dimen/activity_vertical_margin"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:textStyle="bold"/>

    <TextView
        android:id="@+id/text_message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/text_header"
        android:layout_marginLeft="@dimen/activity_horizontal_margin"
        android:layout_marginStart="@dimen/activity_horizontal_margin"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_second"
        android:id="@+id/button_second"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:onClick="returnReply"/>

    <EditText
        android:id="@+id/editText_second"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_toLeftOf="@+id/button_second"
        android:layout_toStartOf="@+id/button_second"
        android:hint="@string/editText_second" />

</RelativeLayout>
```

## 4.2 Create a response intent in the second activity

1. Open `java/com.example.android.twoactivities/SecondActivity`.
2. At the top of the class, add a public constant to define the key for the intent extra:

```
public static final String EXTRA_REPLY =  
    "com.example.android.twoactivities.extra.REPLY";
```

3. Add a private variable at the top of the class to hold the `EditText` object.

```
private EditText mReply;
```

4. In the `onCreate()` method, use `findViewById()` to get a reference to the `EditText` instance and assign it to that private variable:

```
mReply = (EditText) findViewById(R.id.editText_second);
```

5. In the `returnReply()` method, get the text of the `EditText` as a string:

```
String reply = mReply.getText().toString();
```

6. Create a new intent for the response.

**Note:** Do not reuse the intent object you got from the original request. Create a new intent for the response.

```
Intent replyIntent = new Intent();
```

7. Add the reply string from the `EditText` to the new intent as an intent extra. Since extras are key/value pairs, here the key is `EXTRA_REPLY` and the value is the reply:

```
replyIntent.putExtra(EXTRA_REPLY, reply);
```

8. Set the result to `RESULT_OK` to indicate the response was successful. Result codes (including `RESULT_OK` and `RESULT_CANCELLED`) are defined by the [Activity](#) class.

```
setResult(RESULT_OK, replyIntent);
```

9. Call `finish()` to close the activity and return to the main activity.

```
finish();
```

### Solution Code:

```
package com.example.android.twoactivities;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class SecondActivity extends AppCompatActivity {
    public static final String EXTRA_REPLY =
        "com.example.android.twoactivities.extra.REPLY";

    private EditText mReply;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
        mReply = (EditText) findViewById(R.id.editText_second);

        Intent intent = getIntent();
        String message =
            intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
        TextView textView = (TextView) findViewById(R.id.text_message);
        textView.setText(message);
    }

    public void returnReply(View view) {
        String reply = mReply.getText().toString();

        Intent replyIntent = new Intent();
        replyIntent.putExtra(EXTRA_REPLY, reply);
        setResult(RESULT_OK, replyIntent);
        finish();
    }
}
```

## 4.3 Add TextViews to the main activity layout to display the reply

The main activity needs a way to display the reply sent back from the second activity. In this task you'll add TextViews to the main activity layout to display that reply. To make things easier you can copy the TextViews you used in the second activity.

1. Copy the two TextViews for the message display from the second activity layout file and paste them into the main layout above the existing EditText and Button views.
2. Modify the attribute values for both of these new TextViews. Use these values:

Old Attribute (header TextView)	New Attribute (header TextView)
android:id="@+id/text_header"	android:id="@+id/text_header_reply"
android:text="@string/text_header"	android:text= "@string/text_header_reply"
Old Attribute (message TextView)	New Attribute (message TextView)
android:id="@+id/text_message"	android:id="@+id/text_message_reply"
android:layout_below="@+id/text_header"	android:layout_below="@+id/text_header_reply"

3. Add the  `android:visibility` attribute to each of the TextViews to make them initially invisible. (Having them visible on the screen, but without any content, can be confusing to the user.) You will make these TextViews visible after the response data is passed back from the second activity.

```
    android:visibility="invisible"
```

4. Open `res/values/strings.xml` and add a string resource for the reply header:

```
<string name="text_header_reply">Reply Received</string>
```

The layout for the main activity looks the same as it did in the previous task -- although you have added two new TextViews to the layout, their visibility is invisible, and thus they do not appear on the screen.

### Solution Code:

```
<?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: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.android.twoactivities.MainActivity">
    <TextView
        android:id="@+id/text_header_reply"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/text_header_reply"
        android:visibility="invisible"
        android:layout_marginBottom="@dimen/activity_vertical_margin"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:textStyle="bold"/>

    <TextView
        android:id="@+id/text_message_reply"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/text_header_reply"
        android:visibility="invisible"
        android:layout_marginLeft="@dimen/activity_horizontal_margin"
        android:layout_marginStart="@dimen/activity_horizontal_margin"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_main"
        android:id="@+id/button_main"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:onClick="launchSecondActivity"/>

    <EditText
        android:id="@+id/editText_main"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_toLeftOf="@+id/button_main"
        android:layout_toStartOf="@+id/button_main"
        android:hint="@string/editText_main" />
</RelativeLayout>
```

## 4.4 Get the reply from the intent extra and display it

When you use an explicit intent to start another activity, you may often not expect to get any data back -- you're just activating that activity. In that case you use `startActivity()` to start the new activity, as you did earlier in this lesson. If you want to get data back from the activated activity, however, you'll need to start it with `startActivityForResult()`. In this task you'll modify the app to start the second activity and expect a result, to extract that return data from the intent, and to display that data in the `TextViews` you created in the last task.

1. Open `java/com.example.android.twoactivities/MainActivity`.
2. Add a public constant at the top of the class to define the key for a particular type of response you're interested in:

```
public static final int TEXT_REQUEST = 1;
```

3. Add two public variables to hold the reply header and reply `TextViews`:

```
private TextView mReplyHeadTextView;  
private TextView mReplyTextView;
```

4. In the `onCreate()` method, initialize those text views:

```
mReplyHeadTextView = (TextView) findViewById(R.id.text_header_reply);  
mReplyTextView = (TextView) findViewById(R.id.text_message_reply);
```

5. In the `launchSecondActivity()` method, modify the call to `startActivity( )` to be `startActivityForResult()`, and include the `TEXT_REQUEST` key as an argument:

```
startActivityForResult(intent, TEXT_REQUEST);
```

6. Create the `onActivityResult()` callback method with this signature:

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {}
```

7. Inside `onActivityResult()`, call `super.onActivityResult()`:

```
super.onActivityResult(requestCode, resultCode, data);
```

8. Add code to test for both `TEXT_REQUEST` (to process the right intent result, in case there are multiple ones) and the `RESULT_CODE` (to make sure the request was successful):

```
if (requestCode == TEXT_REQUEST) {  
    if (resultCode == RESULT_OK) {  
    }  
}
```

9. Inside the inner if block, get the intent extra from the response intent (`data`). Here the key for the extra is the `EXTRA_REPLY` constant from `SecondActivity`:

```
String reply = data.getStringExtra(SecondActivity.EXTRA_REPLY);
```

10. Set the visibility of the reply header to true:

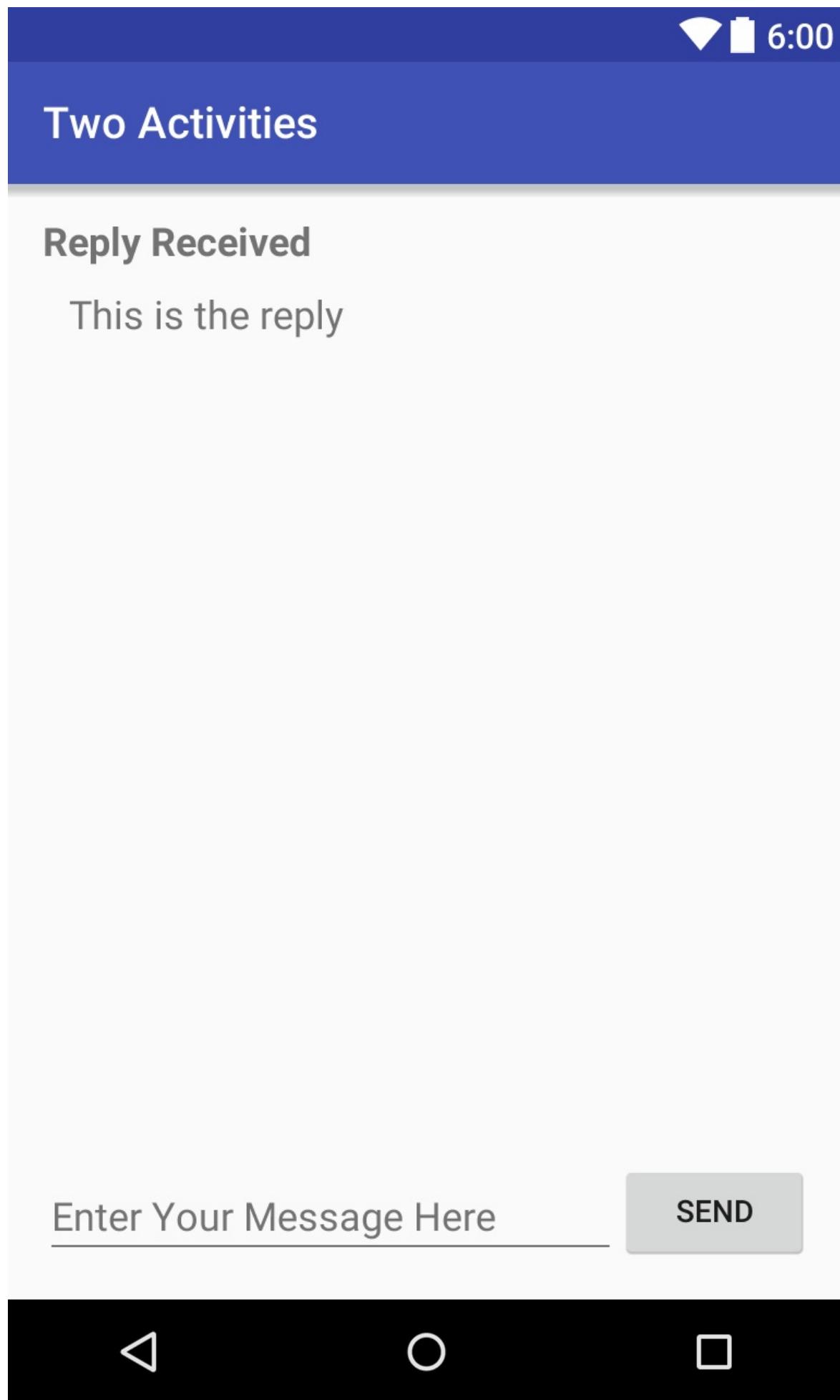
```
mReplyHeadTextView.setVisibility(View.VISIBLE);
```

11. Set the reply textview text to the reply, and set its visibility to true:

```
mReplyTextView.setText(reply);  
mReplyTextView.setVisibility(View.VISIBLE);
```

12. Run the app.

Now, when you send a message to the second activity and get a reply back, the main activity updates to display the reply.



**Solution Code:**

```
package com.example.android.twoactivities;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    private static final String LOG_TAG = MainActivity.class.getSimpleName();

    public static final String EXTRA_MESSAGE =
"com.example.android.twoactivities.extra.MESSAGE";

    public static final int TEXT_REQUEST = 1;

    private EditText mMessageEditText;
    private TextView mReplyHeadTextView;
    private TextView mReplyTextView;

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

        mMessageEditText = (EditText) findViewById(R.id.editText_main);
        mReplyHeadTextView = (TextView) findViewById(R.id.text_header_reply);
        mReplyTextView = (TextView) findViewById(R.id.text_message_reply);
    }

    public void launchSecondActivity(View view) {
        Log.d(LOG_TAG, "Button clicked!");

        Intent intent = new Intent(this, SecondActivity.class);
        String message = mMessageEditText.getText().toString();

        intent.putExtra(EXTRA_MESSAGE, message);
        startActivityForResult(intent, TEXT_REQUEST);
    }

    public void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        if (requestCode == TEXT_REQUEST) {
            if (resultCode == RESULT_OK) {
                String reply =
data.getStringExtra(SecondActivity.EXTRA_REPLY);
```

```
        mReplyHeadTextView.setVisibility(View.VISIBLE);
        mReplyTextView.setText(reply);
        mReplyTextView.setVisibility(View.VISIBLE);
    }
}
}
}
```

## Coding challenge

Create an app with three buttons labelled: Text One, Text Two, and Text Three. When any of those buttons are clicked, launch a second activity. That second activity should contain a ScrollView that displays one of three text passages (you can include your choice of passages). Use intents to both launch the second activity and intent extras to indicate which of the three passages to display.

**Note:** : All coding challenges are optional.

## Summary

- An Activity is an application component that provides a single screen focussed on a single user task.
- Each activity has its own user interface layout file.
- You can assign your activities a parent/child relationship to enable "up" navigation within your app.
- Intents allow you to request an action from another component in your app, for example, to start one activity from another. Intents can be explicit or implicit.
- With explicit intents you indicate the specific target component to receive the message.
- With implicit intents you specify the functionality you want but not the target component.
- Intents can include data on which to perform an action (as a URI) or additional information as intent extras.
- Intent extras are key/value pairs in a bundle that are sent along with the intent.
- Views can be made visible or invisible with the `android:visibility` attribute

## Resources

- [Android Application Fundamentals](#)
- [Starting Another Activity](#)
- [Activity \(API Guide\)](#)

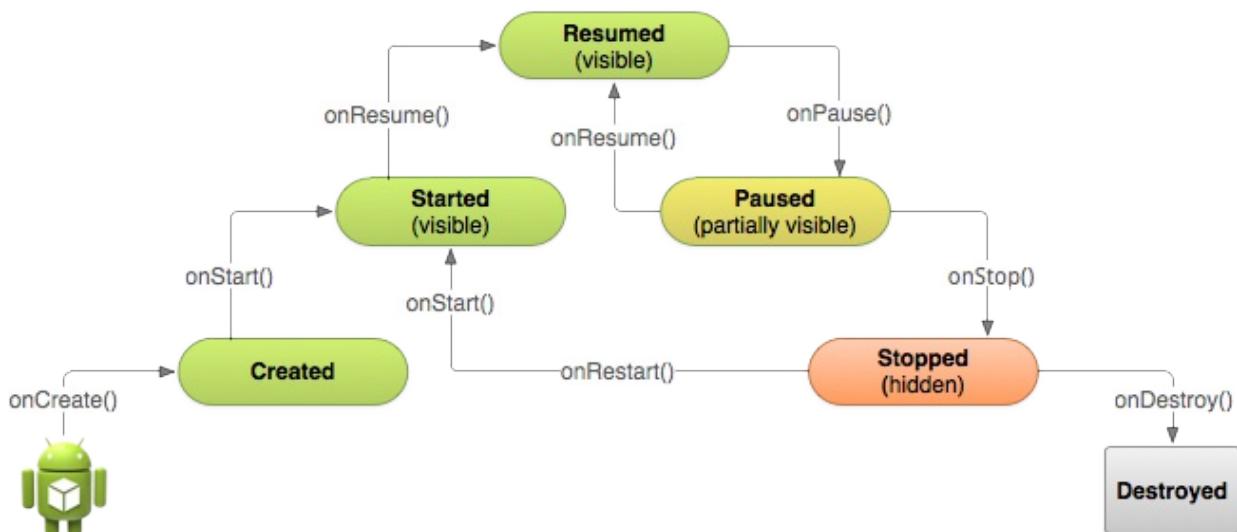
- [Activity \(API Reference\)](#)
- [Intents and Intent Filters \(API Guide\)](#)
- [Intent \(API Reference\)](#)

## 2.2 P: Activity Lifecycle and Instance State

### Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Add Lifecycle Callbacks to TwoActivities
- Task 2. Save and restore activity state
- Coding challenge
- Conclusion
- Resources

In this practical you'll learn more about the *activity lifecycle*. As a user navigates between activities in your app (as well as into and out of your app), those activities each transition between different states in the activity lifecycle.



Each stage in the lifecycle of an activity has a corresponding callback method (`onCreate()`, `onStart()`, `onPause()`, and so on). When an activity changes state, the associated callback method is invoked. You've already seen one of these methods: `onCreate()`. By overriding any of the lifecycle callback methods in your activity classes, you can change the default behavior of how your activity behaves in response to different user or system actions.

Changes to the activity state can also occur in response to device configuration changes such as rotating the device from portrait to landscape. These configuration changes result in the activity being destroyed and entirely recreated in its default state, which may cause the loss of information the user has entered in that activity.

In this practical you'll add logging statements to the TwoActivities app and observe the lifecycle changes as you use the app in various ways. Later in this practical we'll experiment with configuration changes and learn how to preserve the state of your activities in response to device configuration changes or other Activity lifecycle events.

## What you should already KNOW

From the last chapter and the previous section you should be familiar with:

- Creating and running an app project in Android Studio.
- Adding log statements to your app and viewing those logs in the Android Monitor (logcat).
- What activities and intents are, and how they interact.

## What you will LEARN

- About the activity lifecycle, and when activities start, pause, stop, and are destroyed.
- The lifecycle callback methods associated with activity changes.
- Actions such as configuration changes that can result in activity lifecycle events.
- How to retain activity state across lifecycle events.

## What you will DO

- You'll extend the TwoActivities app from the previous practical to implement the various activity lifecycle callbacks to include logging statements, and you'll observe the state changes as your app runs and as you interact with the activities in that app.
- You'll modify your app to retain the instance state of an activity that is unexpectedly recreated in response to user behavior or configuration changes.

## App Overview

For this practical you'll add onto the TwoActivities app. The app looks and behaves mostly the same as it does in the last section: with two activities and two messages you can send in between them. The changes you make to the app in this practical will not affect its visible user behavior.

# Task 1. Add Lifecycle Callbacks to TwoActivities

In this task you will implement all of the activity lifecycle callback methods to print messages to logcat when those methods are invoked. These log messages will allow you to see when the activity lifecycle changes state, and how those lifecycle state changes affect your app as it runs.

## 1.1 (Optional) Copy the TwoActivities Project

For the tasks in this practical, you will modify the existing TwoActivities project you built in the last practical. If you'd prefer to keep the previous TwoActivities app intact, follow the steps in the [Appendix](#) to make a copy of the project.

## 1.2 Implement callbacks in to MainActivity

1. Open `java/com.example.android.twoactivities/MainActivity`.
2. In the `onCreate()` method, add the following log statements:

```
Log.d(LOG_TAG, "-----");
Log.d(LOG_TAG, "onCreate");
```

3. Add a new method for the `onStart()` callback, with a statement to the log for that event:

```
@Override
public void onStart(){
    super.onStart();
    Log.d(LOG_TAG, "onStart");
}
```

TIP: Select **Code > Override Methods** in Android Studio. A dialog appears with all the possible methods you can override in your class. Choosing one or more callback methods from the list inserts a complete template for those methods, including the required call to the superclass. You can also type the name of the method you're interested in, and that override method is highlighted.

4. Use the `onStart()` method as a template to implement the other lifecycle callbacks:
5. `onPause()`
6. `onRestart()`
7. `onResume()`
8. `onStop()`
9. `onDestroy()`

All the callback methods have the same signatures (except for the name). If you copy and paste onStart() to create these other callback methods, don't forget to update the contents to call the right method in the superclass, and to log the correct method.

10. Build and run your app.

**Solution Code (not the entire class):**

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    Log.d(LOG_TAG, "-----");  
    Log.d(LOG_TAG, "onCreate");  
  
    mMessageEditText = (EditText) findViewById(R.id.editText_main);  
    mReplyHeadTextView = (TextView) findViewById(R.id.text_header_reply);  
    mReplyTextView = (TextView) findViewById(R.id.text_message_reply);  
}  
  
@Override  
public void onStart(){  
    super.onStart();  
    Log.d(LOG_TAG, "onStart");  
}  
  
@Override  
public void onRestart() {  
    super.onRestart();  
    Log.d(LOG_TAG, "onRestart");  
}  
  
@Override  
public void onResume() {  
    super.onResume();  
    Log.d(LOG_TAG, "onResume");  
}  
  
@Override  
public void onPause() {  
    super.onPause();  
    Log.d(LOG_TAG, "onPause");  
}  
  
@Override  
public void onStop() {  
    super.onStop();  
    Log.d(LOG_TAG, "onStop");  
}  
  
@Override  
public void onDestroy() {  
    super.onDestroy();  
    Log.d(LOG_TAG, "onDestroy");  
}
```

## 1.3 Implement lifecycle callbacks in SecondActivity

Now that you've implemented the lifecycle callback methods for MainActivity, do the same for SecondActivity.

1. Open java/com.example.android.twoactivities/SecondActivity.
2. At the top of the class, add a constant for the LOG\_TAG variable:

```
private static final String LOG_TAG =  
    SecondActivity.class.getSimpleName();
```

3. Add the lifecycle callbacks and log statements to the second activity. (You can also just copy and paste the callback methods from MainActivity)
4. Add a log statement to the returnReply() method (just before finish()):

```
Log.d(LOG_TAG, "End SecondActivity");
```

**Solution Code (not the entire class):**

```
private static final String LOG_TAG = SecondActivity.class.getSimpleName();

public void returnReply(View view) {
    String reply = mReply.getText().toString();

    Intent replyIntent = new Intent();
    replyIntent.putExtra(EXTRA_REPLY, reply);
    setResult(RESULT_OK, replyIntent);

    Log.d(LOG_TAG, "End SecondActivity");
    finish();
}

@Override
protected void onStart() {
    super.onStart();
    Log.d(LOG_TAG, "onStart");
}

@Override
public void onRestart() {
    super.onRestart();
    Log.d(LOG_TAG, "onRestart");
}

@Override
public void onResume() {
    super.onResume();
    Log.d(LOG_TAG, "onResume");
}

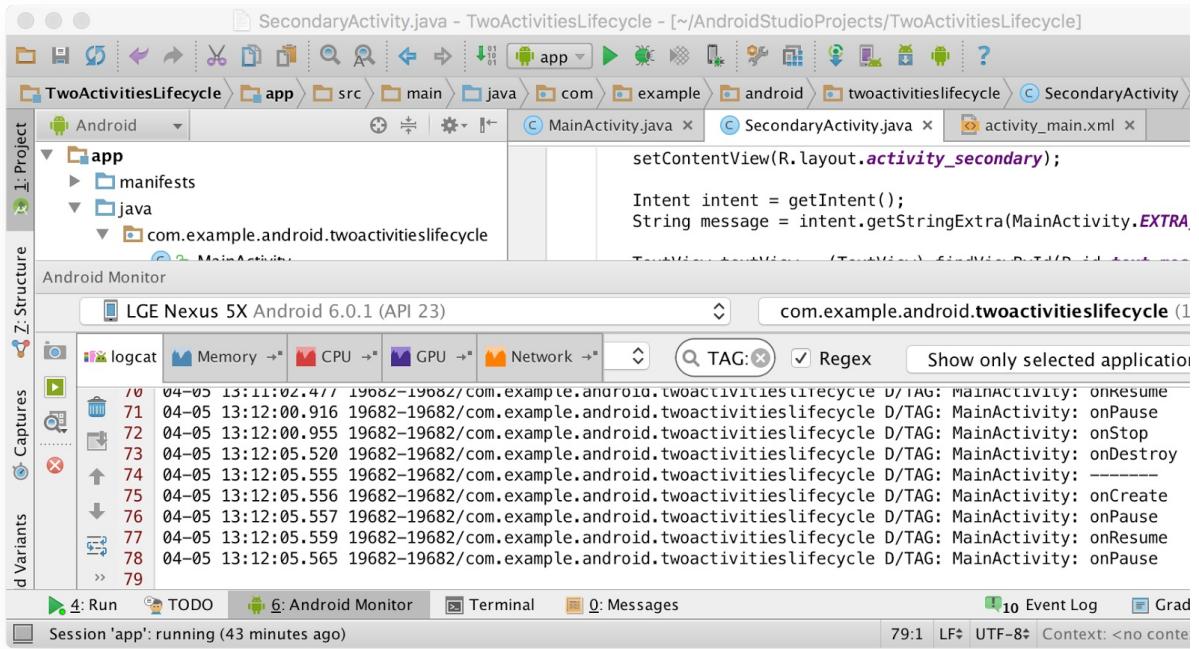
@Override
public void onPause() {
    super.onPause();
    Log.d(LOG_TAG, "onPause");
}

@Override
public void onStop() {
    super.onStop();
    Log.d(LOG_TAG, "onStop");
}

@Override
public void onDestroy() {
    super.onDestroy();
    Log.d(LOG_TAG, "onDestroy");
}
```

## 1.4 Observe the log as the app runs

1. Run your app.
2. Click **Android Monitor** at the bottom of Android Studio to open the Android Monitor.
3. Select the logcat tab.
4. Type "Activity:" in the Android Monitor search box. The Android logcat can be very long and cluttered. Because the LOG\_TAG variable in each class contains either the words MainActivity or SecondActivity, this keyword lets you filter the log for only the things you're interested in.



5. Experiment using your app and note the lifecycle events that occur in response to different actions. In particular, try these things:

- Use the app normally (send a message, reply with another message.)
- Use the back button to go back from the second activity to the main activity.
- Use the left arrow in the action bar to go back from the second activity to the main activity.
- Rotate the device on both the main and second activity at different times in your app and observe what happens in the log and on the screen. **TIP:** If you're running your app in an emulator, you can simulate rotation with Ctrl-F11 or Ctrl-Fn-F11.
- Press the overview button (the square button to the right of Home) and close the app (tap the X).
- Return to the home screen and restart your app.

**Challenge:** Watch for onDestroy() in particular. Why is onDestroy() called sometimes (after clicking the back button, or on device rotation) and not others (manually stopping and restarting the app)?

## Task 2. Save and restore the activity instance state

Depending on system resources and user behavior, the activities in your app may be destroyed and reconstructed far more frequently than you think they will be. You may have noticed this in the last section you rotated the device or emulator. Rotating the device is one example of a device *configuration change*. Although rotation is the most common one, all configuration changes result in the current activity being destroyed and recreated as if it were new. If you don't account for this behavior in your code, when a configuration change occurs, your activity's layout may revert to its default appearance and initial values, and your user may lose their place or the state of their progress in your app.

To keep from losing data in your activities when they are unexpectedly recreated for any reason, you need to implement the `onSaveInstanceState()` method. The system calls this method on your activity (between `onPause()` and `onStop()`) when there is a possibility the activity may be destroyed and recreated.

Note that the data you save in the instance state is specific to only this instance of this specific activity during the current app session. When you stop and restart a new app session, the activity instance state is lost and your activities will revert to their default appearance. If you need to save user data between app sessions, use shared preferences or a database. You'll learn about both of these in a later practical.

### 2.1 Save the activity instance state with `onSaveInstanceState()`

You may have noticed that rotating the device does not affect the state of the second activity at all. This is because the second activity's layout and state are generated from the layout and the intent that activated it. Even if the activity is recreated, the intent is still there and the data in that intent is still used each time the second activity's `onCreate()` is called.

In addition, you may notice that in both activities, any text you typed into message or reply `EditTexts` is retained even when the device is rotated. This is because the state information of some user interface widgets is automatically saved across configuration changes, and the current value of an `EditText` is one of those cases.

So the only activity state you're interested in are the `TextViews` for the reply header and the reply text in the main activity. Both `TextViews` are invisible by default; they only appear once you send a message back to the main activity from the second activity.

In this task you'll add code to preserve the instance state of these two `TextViews` using `onSaveInstanceState()`.

1. Open `java/com.example.android.twoactivities/MainActivity`.
2. Add this skeleton implementation of `onSaveInstanceState()` to the activity, or use **Code > Override Methods** to insert a skeleton override.

```
@Override  
public void onSaveInstanceState(Bundle outState) {  
    super.onSaveInstanceState(outState);  
}
```

3. Check to see if the header is currently visible, and if so put that visibility state into the state bundle with the `putBoolean()` method and the key "reply\_visible".

```
if (mReplyHeadTextView.getVisibility() == View.VISIBLE) {  
    outState.putBoolean("reply_visible", true);  
}
```

Remember that the reply header and text are marked invisible until there is a reply from the second activity. If the header is visible, then there is reply data that needs to be saved. Note that we're only interested in that visibility state -- the actual text of the header doesn't need to be saved, because that text never changes.

4. Inside that same check, add the reply text into the bundle.

```
outState.putString("reply_text", mReplyTextView.getText().toString());
```

If the header is visible you can assume that the reply message itself is also visible. You don't need to test for or save the current visibility state of the reply message. Only the actual text of the message goes into the state bundle with the key "reply\_text".

Note that we only save the state of those views that can potentially change after the activity is created. The other views in your app (the `EditText`, the `Button`) can be recreated from the default layout at any time. Note also the system will save the state of some views (such as the contents of the `EditText`).

#### Solution Code (not the entire class):

```

@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    // If the heading is visible, we have a message that needs to be saved.
    // Otherwise we're still using default layout.
    if (mReplyHeadTextView.getVisibility() == View.VISIBLE) {
        outState.putBoolean("reply_visible", true);
        outState.putString("reply_text", mReplyTextView.getText().toString());
    }
}

```

## 2.2 Restore the activity instance state in onCreate()

Once you've saved the activity instance state, you also need to restore it when the activity is recreated. You can do this either in `onCreate()`, or by implementing the `onRestoreInstanceState()` callback. Typically `onCreate()` is the better place to do it, to ensure that your user interface, including any saved state, is back up and running as quickly as possible.

1. In the `onCreate()` method, add a test to make sure the bundle is not null.

```

if (savedInstanceState != null) {
}

```

When your activity is created, the system passes the state bundle to `onCreate()` as its only argument. The first time `onCreate()` is called, this bundle is null -- there's no existing state the first time your app starts. Subsequent calls to `onCreate()` have a bundle populated with any the data you stored in `onSaveInstanceState()`.

2. Inside that check, get the current visibility (true or false) out of the bundle with the key "reply\_visible"

```

if (savedInstanceState != null) {
    boolean isVisible =
        savedInstanceState.getBoolean("reply_visible");
}

```

3. Add a test below that previous line for the `isVisible` variable.

```

if (isVisible) {
}

```

If there's a `reply_visible` key in the state bundle (and `isVisible` is thus true), we will need to restore the state.

4. Inside the `isVisible` test, make the header visible.

```
mReplyHeadTextView.setVisibility(View.VISIBLE);
```

5. Get the text reply message from the bundle with the key "reply\_text", and set the reply TextView to show that string.

```
mReplyTextView.setText(savedInstanceState.getString("reply_text"));
```

6. Make the reply TextView visible as well:

```
mReplyTextView.setVisibility(View.VISIBLE);
```

7. Run the app. Try rotating the device or the emulator to ensure that the reply message (if there is one) remains on the screen even after the activity is recreated.

### Solution Code (not the entire class):

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

    Log.d(LOG_TAG, "-----");
    Log.d(LOG_TAG, "onCreate");

    // Initialize all the view variables.
    mMessageEditText = (EditText) findViewById(R.id.editText_main);
    mReplyHeadTextView = (TextView) findViewById(R.id.text_header_reply);
    mReplyTextView = (TextView) findViewById(R.id.text_message_reply);

    // Restore the saved state. See onSaveInstanceState() for what gets saved.
    if (savedInstanceState != null) {
        boolean isVisible = savedInstanceState.getBoolean("reply_visible");
        // Show both the header and the message views. If isVisible is
        // false or missing from the bundle, use the default layout.
        if (isVisible) {
            mReplyHeadTextView.setVisibility(View.VISIBLE);

            mReplyTextView.setText(savedInstanceState.getString("reply_text"));
            mReplyTextView.setVisibility(View.VISIBLE);
        }
    }
}
```

## Coding challenge

Create a simple shopping list builder app with two activities. The main activity contains the list itself, which is made up of ten (empty) text views. A button on the main activity labelled "Add Item" launches a second activity that contains a list of common shopping items (Cheese, Rice, Apples, and so on). Use Buttons to display the items. Choosing an item returns you to the main activity, and updates an empty TextView to include the chosen item.

Use intents to pass information between the two activities. Make sure that the current state of the shopping list is saved when you rotate the device.

**Note:** : All coding challenges are optional.

## Summary

- The Activity lifecycle is a set of states an activity can be in from when it is first created to when the Android system reclaims that activity's resources.
- As the user navigates between activities and inside and outside of your app, each activity moves between states in the activity lifecycle.
- Each state in the activity lifecycle has a corresponding callback method you can override in your Activity class. Those lifecycle methods are:
  - `onCreate()`
  - `onStart()`
  - `onPause()`
  - `onRestart()`
  - `onResume()`
  - `onStop()`
  - `onDestroy()`
- Overriding a lifecycle callback method enables you to add behavior that occurs when your activity transitions into that state.
- You can add skeleton override methods to your classes in Android Studio with Code > Override.
- Device configuration changes such as rotation results in the activity being destroyed and recreated as if it were new.
- Some activity state is preserved on a configuration change, including the current values of of EditTexts. For all other data you must explicitly save that data yourself.
- Save activity instance state in the `onSaveInstanceState()` method.
- Instance state data is stored as simple key/value pairs in a Bundle. Use the Bundle methods to put data into and get data back out of the bundle.
- Restore the instance state in `onCreate()` (preferred) or `onRestoreInstanceState()`.

## Resources

- [Activity \(API Guide\)](#)
- [Activity \(API Reference\)](#)
- [Managing the Activity Lifecycle](#)
- [Recreating an Activity](#)
- [Handling Runtime Changes](#)

## 2.3 P: Activities and Implicit Intents

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Create new project and layout](#)
- [Task 2. Implement open website](#)
- [Task 3. Implement open location](#)
- [Task 4. Implement share this text](#)
- [Task 5. Receive Implicit Intents](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

In a previous section you learned about *explicit* intents -- activating a specific activity in your app or a different app by sending an intent with the fully-qualified class name of that activity. In this section you'll learn more about *implicit* intents, and how you can use them to activate activities as well.

Implicit intents allow you to activate an activity if you know the action, but not the specific app or activity that will handle that action. For example, if you want your app to take a photo, or send email, or display a location on a map, you typically do not care which specific app or activity actually performs these actions.

Conversely, your activities can declare one or more intent filters in the Android manifest that advertise that activity's ability to accept implicit intents and to define the particular type of intents it will accept.

To match your request with a specific app installed on the device, the Android system matches your implicit intent with an activity whose intent filters indicate that they can perform that action. If there are multiple apps installed that match, the user is presented with an app chooser that lets them select which app they want to use to handle that intent.

In this practical you'll build an app that sends three implicit intents: to open a URL in a web browser, to open a location on a map, and to share a bit of text. Sharing -- sending a piece of information to other people through email or social media -- is a common and popular feature in many apps. For the sharing action we'll use the `ShareCompat.IntentBuilder` class, which makes it easy to build intents for sharing data.

Finally, we'll create a simple intent receiver app that accepts implicit intents for a specific action.

## What you should already KNOW

From the previous sections you should be familiar with:

- Creating and using activities.
- Creating and sending intents between activities.

## What you will LEARN

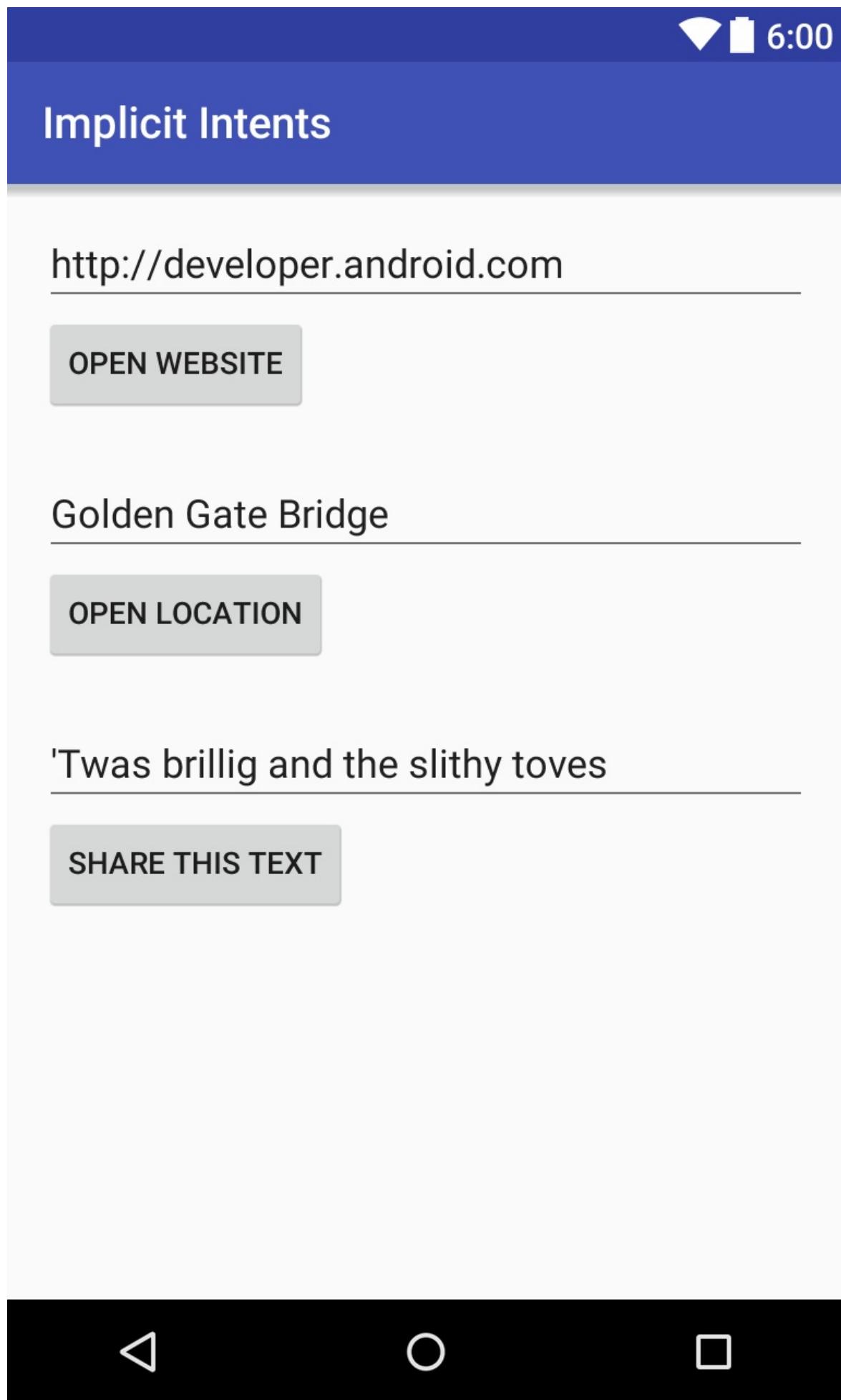
- How to create implicit intents, and the intent actions and categories you can use.
- Using the ShareCompat.IntentBuilder helper class to easily create implicit intents for sharing data.
- How to advertise that your app can accept implicit intents by declaring intent filters in the Andriod manifest

## What you will DO

- Create a new app to send implicit intents.
- Implement two implicit intents that open a web page and open a location on a map.
- Implement an action to share a snippet of text.
- Create a new app that can accept implicit intents for opening a web page.

## App Overview

In this section you'll create a new app with one activity and three options for actions: open a web site, open a location on a map, and share a snippet of text. All the text fields are editable (EditText) but contain default values.



# Task 1. Create new project and layout

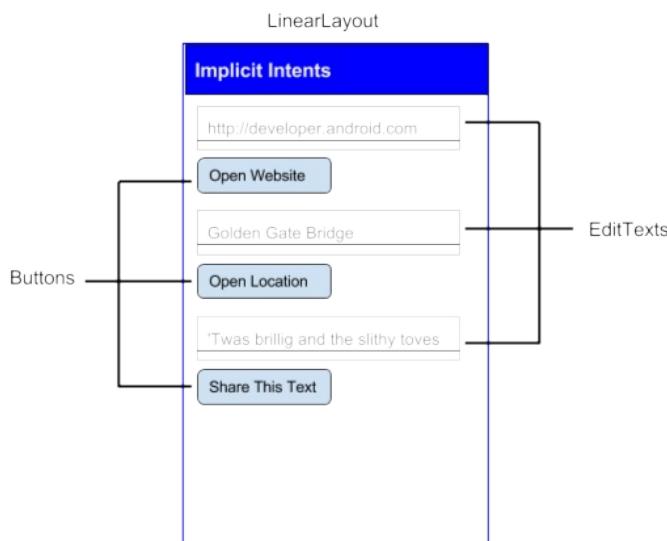
For this section you'll create a new project and app called **Implicit Intents** with a new layout.

## 1.1 Create the Project

1. Start Android Studio and create a new Android Studio project. Call your application "Implicit Intents."
2. Choose **Empty Activity** for the project template. Click **Next**.
3. Accept the default activity name (MainActivity). Make sure the **Generate Layout file** box is checked. Click **Finish**.

## 1.2 Create the Layout

In this task, create the layout for the app. Use a **LinearLayout**, three **Buttons**, and three **EditTexts**, like this:



1. Edit `res/values/strings.xml` to include these string resources:

```

<string name="edittext_uri">http://developer.android.com</string>
<string name="button_uri">Open Website</string>

<string name="edittext_loc">Golden Gate Bridge</string>
<string name="button_loc">Open Location</string>

<string name="edittext_share">'Twas brillig and the slithy toves</string>
<string name="button_share">Share This Text</string>

```

2. Change the default RelativeLayout to a Linear Layout. Add the android:orientation attribute and give it the value "vertical."

```

<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: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.android.implicitintents.MainActivity"
    android:orientation="vertical">

```

3. Remove the "Hello World" TextView.  
 4. Add an EditText and a Button to the layout for the Open Website function. Use these attribute values:

<b>Attribute (EditText)</b>	<b>Value (EditText)</b>
android:id	"@+id/website_edittext"
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:text	"@string/edittext_uri"
<b>Attribute (Button)</b>	<b>Value (Button)</b>
android:id	"@+id/open_website_button"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_marginBottom	"24dp"
android:text	"@string/button_uri"
android:onClick	"openWebsite"

5. Add a second EditText and a Button for the Open Website function. Use the same attributes to those in the previous step, but modify these attributes:

Attribute (EditText)	Value (EditText)
android:id	"@+id/location_edittext"
android:text	"@string/edittext_loc"
Attribute (Button)	Value (Button)
android:id	"@+id/open_location_button"
android:text	"@string/button_loc"
android:onClick	"openLocation"

6. Add a third EditText and a Button for the Share This function. Make these changes:

Attribute (EditText)	Value (EditText)
android:id	"@+id/share_edittext"
android:text	"@string/edittext_share"
Attribute (Button)	Value (Button)
android:id	"@+id/share_text_button"
android:text	"@string/button_share"
android:onClick	"shareText"

### Solution Code:

```

<?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: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.android.implicitintents.MainActivity"
    android:orientation="vertical">

    <EditText
        android:id="@+id/website_edittext"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/edittext_uri" />

    <Button
        android:id="@+id/open_website_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="24dp"
        android:onClick="openWebsite"
    >
```

```
        android:text="@string/button_uri" />

    <EditText
        android:id="@+id/location_edittext"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/edittext_loc" />

    <Button
        android:id="@+id/open_location_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="24dp"
        android:onClick="openLocation"
        android:text="@string/button_loc" />

    <EditText
        android:id="@+id/share_edittext"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/edittext_share" />

    <Button
        android:id="@+id/share_text_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="24dp"
        android:onClick="shareText"
        android:text="@string/button_share" />
</LinearLayout>
```

## Task 2. Implement open website

In this task you'll add the `onClick()` method for the first button in the layout ("Open Website.") This action uses an implicit intent to send the given URI to an activity that can handle that Implicit Intent (such as a web browser).

### 2.1 Define the `openWebsite` method

1. Open `MainActivity.java`.
2. Add a private variable at the top of the class to hold the `EditText` object for the web site URI.

```
private EditText mWebsiteEditText;
```

3. In the `onCreate()` method, use `findViewById()` to get a reference to the `EditText` instance and assign it to that private variable:

```
mWebsiteEditText = (EditText) findViewById(R.id.website_edittext);
```

4. Create a new method called `openWebsite()`, with this signature:

```
public void openWebsite(View view) { }
```

5. Get the string value of the `EditText`:

```
String url = mWebsiteEditText.getText().toString();
```

6. Encode and parse that string into a `Uri` object:

```
Uri webpage = Uri.parse(url);
```

7. Create a new Intent with `Intent.ACTION_VIEW` as the action and the URI as the data:

```
Intent intent = new Intent(Intent.ACTION_VIEW, webpage);
```

Note that this intent constructor is different from the one you used to create an explicit intent. In that constructor you specified the current context and a specific component (activity class) to send the intent. In this constructor you specify an action and the data for that action. Actions are defined by the `Intent` class and can include `ACTION_VIEW` (to view the given data), `ACTION_EDIT` (to edit the given data), or `ACTION_DIAL` (to dial a phone number). In this case the action is `ACTION_VIEW` because we want to open and view the web page specified by the `URI` in the `webpage` variable.

8. Use the `resolveActivity()` and the Android package manager to find an activity that can handle your implicit intent. Check to make sure the that request resolved successfully.

```
if (intent.resolveActivity(getApplicationContext()) != null) {  
}
```

This request that matches your intent action and data with the intent filters for installed applications on the device to make sure there is at least one activity that can handle your requests.

9. Inside the if-statement, call `startActivity()` to send the intent.

```
startActivity(intent);
```

10. Add an else block to print a log message if the intent could not be resolved.

```
} else {  
    Log.d("ImplicitIntents", "Can't handle this!");  
}
```

**Solution Code (not the entire class):**

```

public void openWebsite(View view) {
    // Get the URL text.
    String url = mWebsiteEditText.getText().toString();

    // Parse the URI and create the intent.
    Uri webpage = Uri.parse(url);
    Intent intent = new Intent(Intent.ACTION_VIEW, webpage);

    // Find an activity to hand the intent and start that activity.
    if (intent.resolveActivity(getApplicationContext()) != null) {
        startActivity(intent);
    } else {
        Log.d("ImplicitIntents", "Can't handle this intent!");
    }
}

```

## Task 3. Implement open location

In this task you'll add the `onClick()` callback for the second button in the UI ("Open Location.") This method is almost identical to `openWebsite`. The difference is the use of a geo URI to indicate a map location. You can use a geo URI with latitude and longitude, or use a query string for a general location. In this example we've used the latter.

### 3.1 Define the `openLocation` method

1. Open `MainActivity.java` (`java/com.example.android.implicitintents/MainActivity`).
2. Add a private variable at the top of the class to hold the `EditText` object for the location URI.

```
private EditText mLocationEditText;
```

3. In the `onCreate()` method, use `findViewById()` to get a reference to the `EditText` instance and assign it to that private variable:

```
mLocationEditText = (EditText) findViewById(R.id.location_edittext);
```

4. Create a new method called `openLocation` to use as the `onClick` method for the Open Location button. Use the same method signature as `openWebsite()`.
5. Get the string value of the `mLocationEditText` `EditText`.

```
String loc = mLocationEditText.getText().toString();
```

- Parse that string into a Uri object with a geo search query:

```
Uri addressUri = Uri.parse("geo:0,0?q=" + loc);
```

- Create a new Intent with Intent.ACTION\_VIEW as the action and loc as the data.

```
Intent intent = new Intent(Intent.ACTION_VIEW, addressUri);
```

- Resolve the intent and check to make sure the intent resolved successfully. If so, startActivity(), otherwise log an error message.

```
if (intent.resolveActivity(getApplicationContext()) != null) {
    startActivity(intent);
} else {
    Log.d("ImplicitIntents", "Can't handle this intent!");
}
```

### Solution Code (not the entire class):

```
public void openLocation(View view) {
    // Get the string indicating a location. Input is not validated; it is
    // passed to the location handler intact.
    String loc = mLocationEditText.getText().toString();

    // Parse the location and create the intent.
    Uri addressUri = Uri.parse("geo:0,0?q=" + loc);
    Intent intent = new Intent(Intent.ACTION_VIEW, addressUri);

    // Find an activity to handle the intent, and start that activity.
    if (intent.resolveActivity(getApplicationContext()) != null) {
        startActivity(intent);
    } else {
        Log.d("ImplicitIntents", "Can't handle this intent!");
    }
}
```

## Task 4. Implement share this text

Sharing actions are an easy way for users to share items in your app with social networks and other apps. Although you could build a share action in your own app using implicit intents, Android provides the [ShareCompat.IntentBuilder](#) helper class to make implementing sharing easy. You can use ShareCompat.IntentBuilder to build an intent and launch a chooser to let the user choose the destination app for sharing.

In this final task we'll implement sharing a bit of text in a text edit with the ShareCompat.IntentBuilder class.

## 4.1 Implement the shareText method

1. Open MainActivity.java.
2. Add a private variable at the top of the class to hold the EditText object for the web site URI.

```
private EditText mShareTextEdit;
```

3. In the onCreate() method, use findViewById() to get a reference to the EditText instance and assign it to that private variable:

```
mShareTextEdit = (EditText) findViewById(R.id.share_edittext);
```

4. Create a new method called shareThis() to use as the onClick method for the Share This Text button. Use the same method signature as openWebsite().
5. Get the string value of the mShareTextEdit EditText.

```
String txt = mShareTextEdit.getText().toString();
```

6. Define the mime type of the text to share:

```
String mimeType = "text/plain";
```

7. Call ShareCompat.IntentBuilder with these methods:

```
ShareCompat.IntentBuilder
    .from(this)
    .setType(mimeType)
    .setChooserTitle("Share this text with: ")
    .setText(txt)
    .startChooser();
```

This call to ShareCompat.IntentBuilder uses these methods:

Method	Description
from()	The activity that launches this share intent (this).
setType()	The MIME type of the item to be shared.
setChooserTitle()	The title that appears on the system app chooser.
setText()	The actual text to be shared
startChooser()	Show the system app chooser and send the intent.

This format, with all the builder's setter methods strung together in one statement, is an easy shorthand way to create and launch the intent. You can add any of the additional methods to this list.

### Solution Code (not the entire class):

```
public void shareText(View view) {
    String txt = mShareTextEditEditText.getText().toString();
    String mimeType = "text/plain";

    ShareCompat.IntentBuilder
        .from(this)
        .setType(mimeType)
        .setChooserTitle("Share this text with: ")
        .setText(txt)
        .startChooser();
}
```

## Task 5. Receive Implicit Intents

Up to this point in the chapter we've created apps that use both explicit and implicit intents to launch some other app's activity. In this task we'll look at the problem from the other way around: allowing an activity in your app to respond to implicit intents sent from some other app.

Activities in your app can always be activated from inside or outside your app with explicit intents. To allow an activity to receive implicit intents, you define an *intent filter* in your manifest to indicate which implicit intents your activity is interested in handling.

To match your request with a specific app installed on the device, the Android system matches your implicit intent with an activity whose intent filters indicate that they can perform that action. If there are multiple apps installed that match, the user is presented with an app chooser that lets them select which app they want to use to handle that intent.

When an app on the device sends an implicit intent, the Android system matches that intent's action and data with available activities that include the right intent filters. If your activity's intent filters match the intent, your activity can either handle the intent itself (if it is the only matching activity), or (if there are multiple matches) an app chooser appears to allow the user to pick which app they'd prefer to execute that action.

In this task you'll create a very simple app that receives implicit intents to open the URI for a web page. When activated by an implicit intent, that app displays the requested URI as a string in a TextView.

## 5.1 Create the Project & Layout

1. Start Android Studio and create a new Android Studio project.
2. Call your application "Implicit Intents Receiver."
3. Choose **Empty Activity** for the project template.
4. Accept the default activity name (MainActivity). Click **Next**.
5. Make sure the **Generate Layout file** box is checked. Click **Finish**.
6. Open `res/layout/activity_main.xml`.
7. Give the default ("Hello World") TextView these attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/text_uri_message"</code>
<code>android:layout_width</code>	<code>wrap_content</code>
<code>android:layout_height</code>	<code>wrap_content</code>
<code>android:textSize</code>	<code>"18sp"</code>
<code>android:textStyle</code>	<code>"bold"</code>

8. Delete the `android:text` attribute. There's no text in this TextView by default, but you'll add the URI from the intent in `onCreate()`.

## 5.2 Modify the Android Manifest

1. Open `manifests/AndroidManifest.xml`.
2. Note that the main activity already has this intent filter:

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

This intent filter, which is part of the default project manifest, indicates that this activity is the main entry point for your app (it has an intent action of "android.intent.action.MAIN"), and that this activity should appear as a top-level item in the launcher (its category is "android.intent.category.LAUNCHER" )

3. Add a second `<intent-filter>` tag inside `<activity>`, and include these elements :

```
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
<data android:scheme="http" android:host="developer.android.com" />
```

These lines define an intent filter for the activity, that is, the kind of intents that the

activity can handle. This intent filter declares these elements:

Filter Type	Value	Matches
action	"android.intent.action.VIEW"	All intents with view actions.
category	"android.intent.category.DEFAULT"	All implicit intents. This category must be included for your activity to receive <b>any</b> implicit intents.
category	"android.intent.category.BROWSABLE"	Requests for browsable links from web pages, email, or other sources.
data	android:scheme="http" android:host="developer.android.com"	URIs that contain a scheme of http AND a host name of developer.android.com.

Note that the data filter has a restriction on both the kind of links it will accept and the hostname for those URIs. If you'd prefer your receiver to be able to accept any links, you can leave the <data> element out altogether.

## Solution Code

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

    <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>

            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.BROWSABLE" />
                <data android:scheme="http"
                    android:host="developer.android.com" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

## 5.3 Process the Intent

In the `onCreate()` method for your activity, you process the incoming intent for any data or extras it includes. In this case incoming implicit intent has the URI stored in the Intent data.

1. Open `MainActivity.java`.
2. In the `onCreate()` method, get the incoming intent that was used to activate the activity:

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

3. Get the intent data. Intent data is always a `URI` object:

```
Uri uri = intent.getData();
```

4. Check to make sure the `uri` variable is not null. If that check passes, create a string from that `URI` object:

```
if (uri != null) {  
    String uri_string = "URI: " + uri.toString();  
}
```

5. Inside that same if block, get the text view for the message:

```
TextView textView = (TextView) findViewById(R.id.text_uri_message);
```

6. Also inside the if-block, set the text of that TextView to the URI:

```
textView.setText(uri_string);
```

7. Run the receiver app.

Running the app on its own shows a blank activity with no text. This is because the activity was activated from the system launcher, and not with an intent from another app.

8. Run the ImplicitIntents app, and click Open Website with the default URI.

An app chooser appears asking if you want to use the default browser or the ImplicitIntentsReceiver app. Choose "Just Once" for the receiver app. The ImplicitIntentsReceiver app launches and the message shows the URI from the original request.

9. Tap the back button and enter a different URI. Click Open Website.

The receiver app has a very restrictive intent filter that matches only exact URI protocol (http) and host (developer.android.com). Any other URI opens in the default web browser.

## Coding challenge

In the last section's challenge you created a shopping list app builder with two activities: one to display the list, and one to pick an item. Add an EditText and a Button to the shopping list activity to locate a particular store on a map.

**Note:** : All coding challenges are optional.

## Summary

- Implicit intents allow you to activate an activity if you know the action, but not the

specific app or activity that will handle that action.

- Actions that implicit intents can request might include taking a photo, playing a song, or display a location on a map.
- Activities that can receive implicit intents must define intent filters in their Android manifest that match one or more intent actions and categories.
- The Android system matches the content of an implicit intent and the intent filters of all available activities to determine which activity to activate. If there is more than one available activity, the system provides a chooser so the user can pick one.
- Create implicit intents in the sending app with an action and a URI representing the data to act on.
- To open a web URL, use an implicit intent with the intent action Intent.ACTION\_VIEW and the URL in the intent data.
- To open a location on a map, use an implicit intent with the intent action Intent.ACTION\_VIEW and the a location specified as a geo URI in the intent data.
- Use the Intent.resolveActivity() method to ensure that there is an activity that can handle your intent before you send the intent.
- Use startActivityForResult() or startActivityForResult() to send the implicit intent and activate an activity.
- The ShareCompat.IntentBuilder class makes it easy to build implicit intents for sharing data to social media or email.
- Create intent filters in the Android manifest with the <intent-filter> element. Intent filters can have actions, categories, and data. All the filters must match for an activity to be able to accept and handle implicit intents.
- Handle an incoming intent in your activity's onCreate() with the getIntent() method. Retrieve the URI from that intent's data and perform the requested action.

## Resources

- [Activity \(API Guide\)](#)
- [Activity \(API Reference\)](#)
- [Intents and Intent Filters \(API Guide\)](#)
- [Intent \(API Reference\)](#)
- [Uri](#)
- [Google Maps Intents](#)
- [ShareCompat.IntentBuilder \(API Reference\)](#)

# 3.1 P: Using the Debugger

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Create the SimpleCalc Project and App](#)
- [Task 2. Run SimpleCalc in the Debugger](#)
- [Task 3. Explore Debugger Features](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

In previous practicals you used the Log class to print information to the system log (logcat) when your app runs. Adding logging statements to your app is one way to find errors and improve your app's operation. Another way is to use the debugger built into Android Studio.

In this practical you'll learn how to debug your app in an emulator and on the device, set and view breakpoints, step through your code, and examine variables.

## What you should already KNOW

From the previous practicals you should be familiar with:

- Creating an Android Studio project, and working with EditText and Button views.
- Building and running your app in Android Studio, in both an emulator and on a device.
- Adding log statements and viewing the system log (logcat) in Android Monitor.

## What you will LEARN

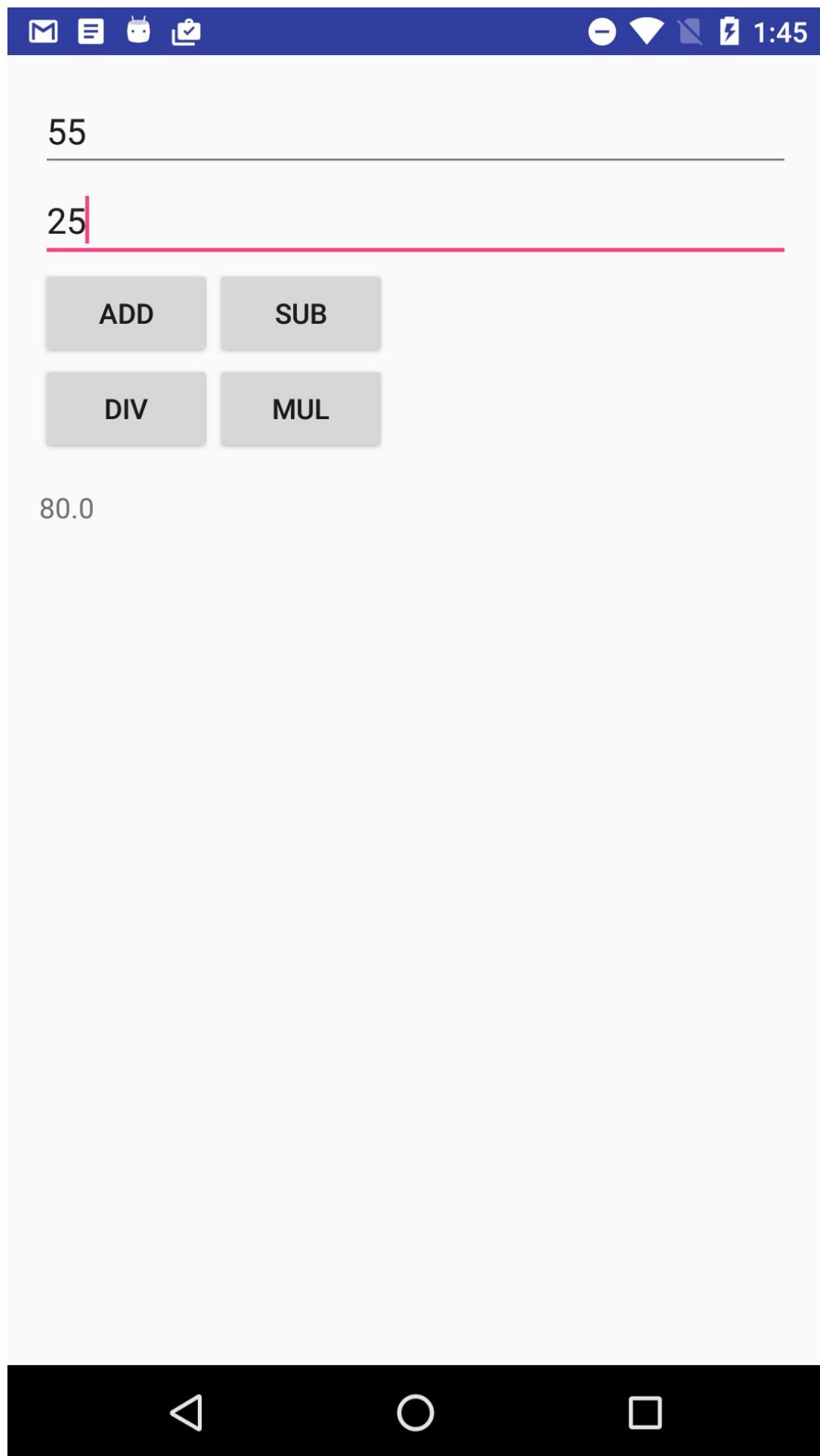
- How to run your app in debug mode in an emulator or on a device.
- How to step through the execution of your app.
- How to set and organize breakpoints.
- How to examine and modify variables in the debugger.

## What you will DO

- Build the SimpleCalc app.
- Set and view breakpoints in the code for SimpleCalc.
- Step through your code as it runs.
- Examine variables and evaluate expressions.
- Identify and fix problems in the sample app.

## App Overview

The SimpleCalc app has two edit texts and four buttons. When you enter two numbers and click a button, the app performs the calculation for that button and displays the result.



# Task 1. Create the SimpleCalc Project and App

For this practical you won't build the SimpleCalc app yourself. The complete project is available at . In this task you will open the SimpleCalc project into Android Studio and explore some of the app's key features.

## 1.1 Download and Open the SimpleCalc Project

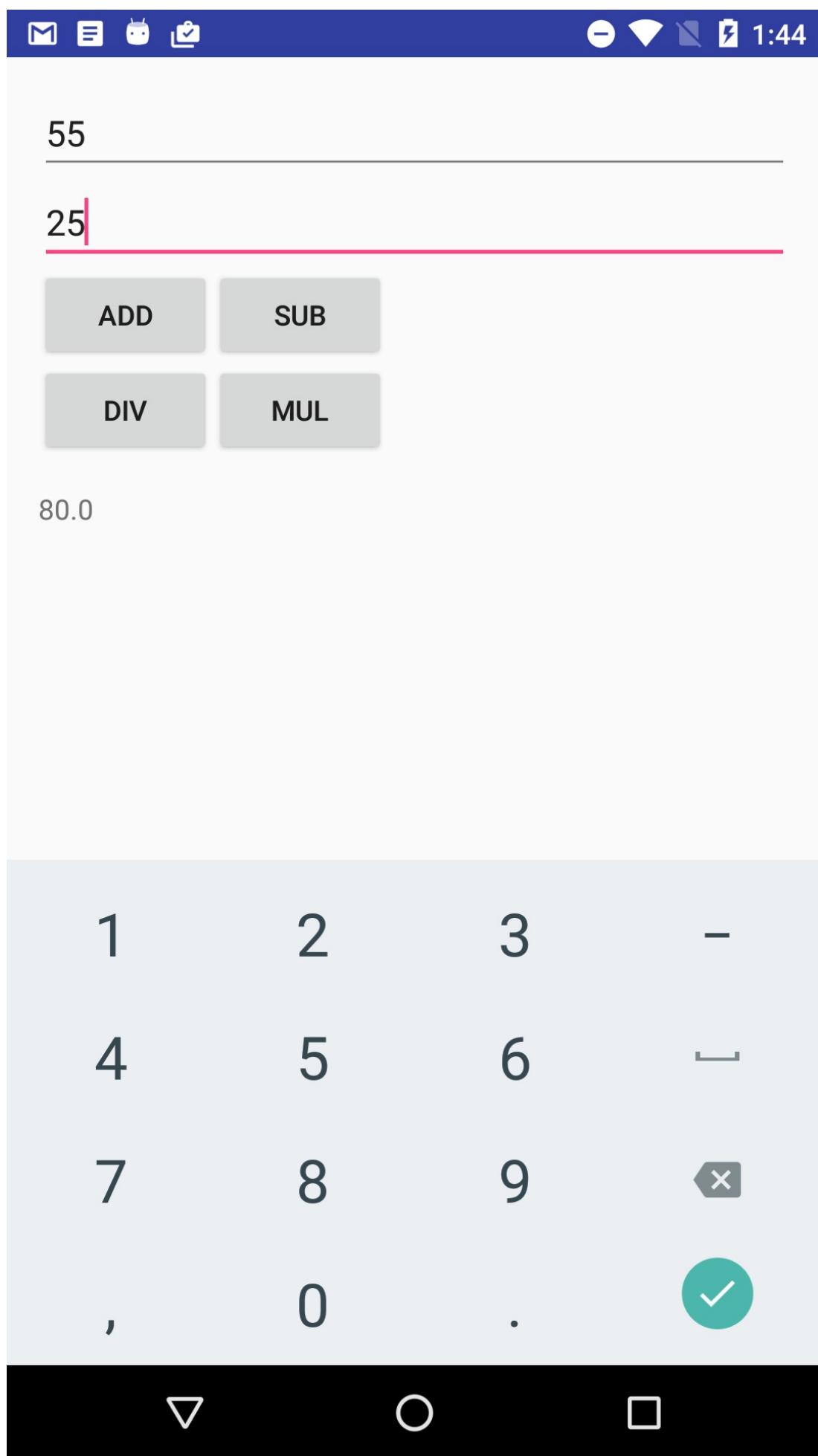
1. Download and unzip the file.
2. Start Android Studio and select File > Open.
3. Navigate to the folder for SimpleCalc, select that folder file, and click **OK**.

The SimpleCalc project builds. Open the project view if it is not already open.

**Warning:** : This app contains errors that you will find and fix. If you run the app on a device or emulator you might run into unexpected behavior and even crashes.

## 1.2 Explore the Layout

1. Open `res/layout/activity_main.xml` .
2. Preview the layout in the Layout Editor.
3. Examine the layout code and design and note the following:
  - The layout contains two EditTexts for the input, four Button views for the calculations, and one TextViews to display the result.
  - Each calculation button has its own onClick handler (onAdd, OnSub, and so on.)
  - The TextView for the result does not have any text in it by default.
  - The two EditText views have the property `android:inputType` and the value `"numberDecimal"` . This property indicates that the EditText only accepts numbers as input. The keyboard that appears on screen will only contain numbers. You will learn more about input types for EditTexts in a later practical.



A screenshot of a mobile application interface for a calculator. At the top, there is a blue header bar with various icons: a mail icon, a document icon, a settings icon, a battery icon, and the time '1:44'. Below the header, there are two input fields. The first input field contains the number '55'. The second input field contains the number '25', with a red cursor line indicating it is active or selected. Below these input fields are four grey buttons labeled 'ADD', 'SUB', 'DIV', and 'MUL', representing addition, subtraction, division, and multiplication respectively. In the main body of the screen, the result '80.0' is displayed. At the bottom, there is a numeric keypad grid. The grid includes digits 1 through 9, a decimal point '.', a comma ',', and a multiplier symbol '×'. To the right of the grid is a green circular button with a white checkmark. Below the keypad is a black navigation bar with three white icons: a downward-pointing triangle, a circle, and a square.

1	2	3	-
4	5	6	[ ]
7	8	9	×
,	0	.	✓

▽ ○ □

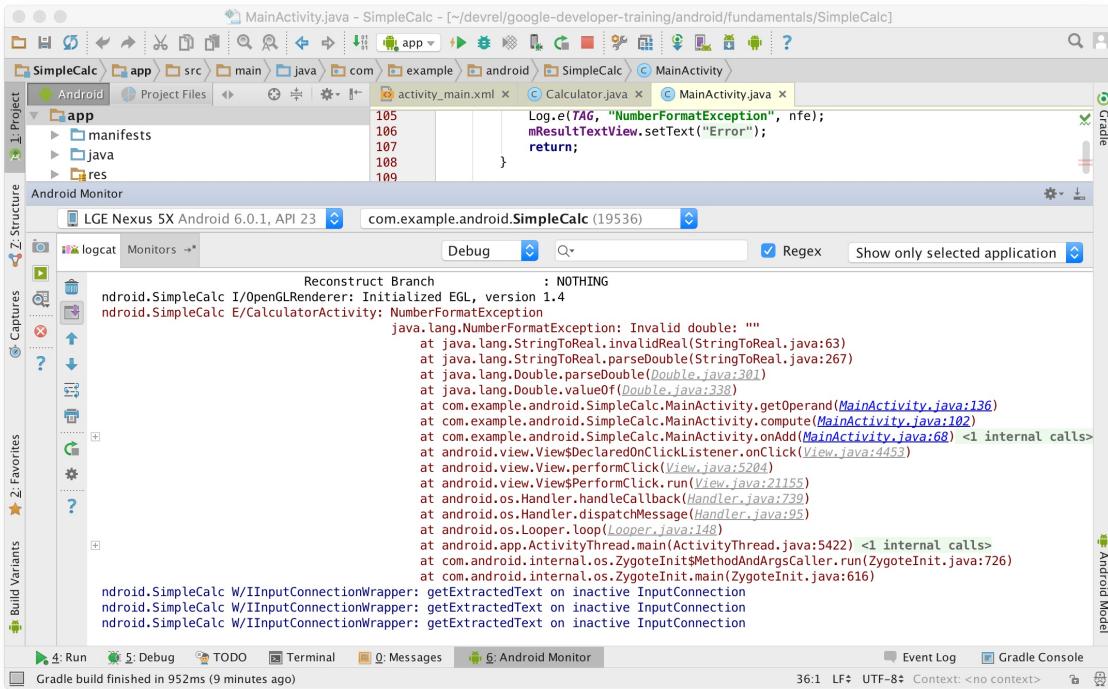
## 1.3 Explore the app code

1. Expand the app/java folder in the Android project view. Note that in addition to the MainActivity class, this project also includes a utility Calculator class.
2. Open Calculator (java/com.example.android.simplecalc/Calculator.java). Examine the code and note these things:
  - The operations the calculator can perform are defined by the Operator enum.
  - All of the operation methods are public.
3. Open MainActivity (java/com.example.android.simplecalc/MainActivity). Examine the code and note these things:
  - All of the onClick handlers call the private compute() method, with the operation name as one of the values from the Calculator.Operator enumeration.
  - The compute() method calls the private method getOperand() (which in turn calls getOperandText()) to retrieve the number values from the EditTexts.
  - The compute() method then uses a switch on the operand name to call the appropriate method in the Calculator class.
  - The calculation methods in the Calculator class perform the actual arithmetic and return a value.
  - The last part of the compute() method updates the TextView with the result of the calculation.
4. Run the app. Try these things:
  - Enter both integer and floating-point values for the calculation.
  - Enter floating-point values with large decimal fractions (for example, 1.6753456)
  - Divide a number by zero.
  - Leave one or both of the EditText views empty, and try any calculation.
5. Examine the stack trace in Android Studio when the app reports an error.

If the stack trace is not visible, click the Android Monitor button at the bottom of the Android Studio, and then click logcat.

If one or both the EditText views in SimpleCalc is empty, the app reports "Error" and the system log displays the state of the execution stack at the time the app produced the error. The stack trace usually provides important information about why an error or

happened. Examine the stack trace and try to figure out why the error occurred (but don't fix it yet.)



## Task 2. Run SimpleCalc in the Debugger

In this task you'll get an introduction to the debugger in Android Studio, and learn how to run your app in debug mode.

### 2.1 Start and Run your app in debug mode

1. In Android Studio, select **Run > Debug app** or click the **Debug** icon in the toolbar.

If your app is already running, you will be asked if you want to restart your app in debug mode. Click **Restart app**.

Android Studio builds and runs your app on the emulator or on the device. Debugging is the same in either case. While Android Studio is initializing the debugger, you may see a message that says "Waiting for debugger" on the device before you can use your app.

If the **Debug** view does not automatically appear in Android Studio, click the **Debug** tab at the bottom of the screen, and then the **Debugger** tab.

2. Open the **MainActivity.java** file and click in the fourth line of the **compute()** method (the line just after the **try** statement).

- Click in the left gutter of the editor window at that line, next to the line numbers. A red dot appears at that line, indicating a breakpoint.

You can also use **Run > Toggle Line Breakpoint** or Control-F8 (Command-F8 on OS X) to set a breakpoint at a line.

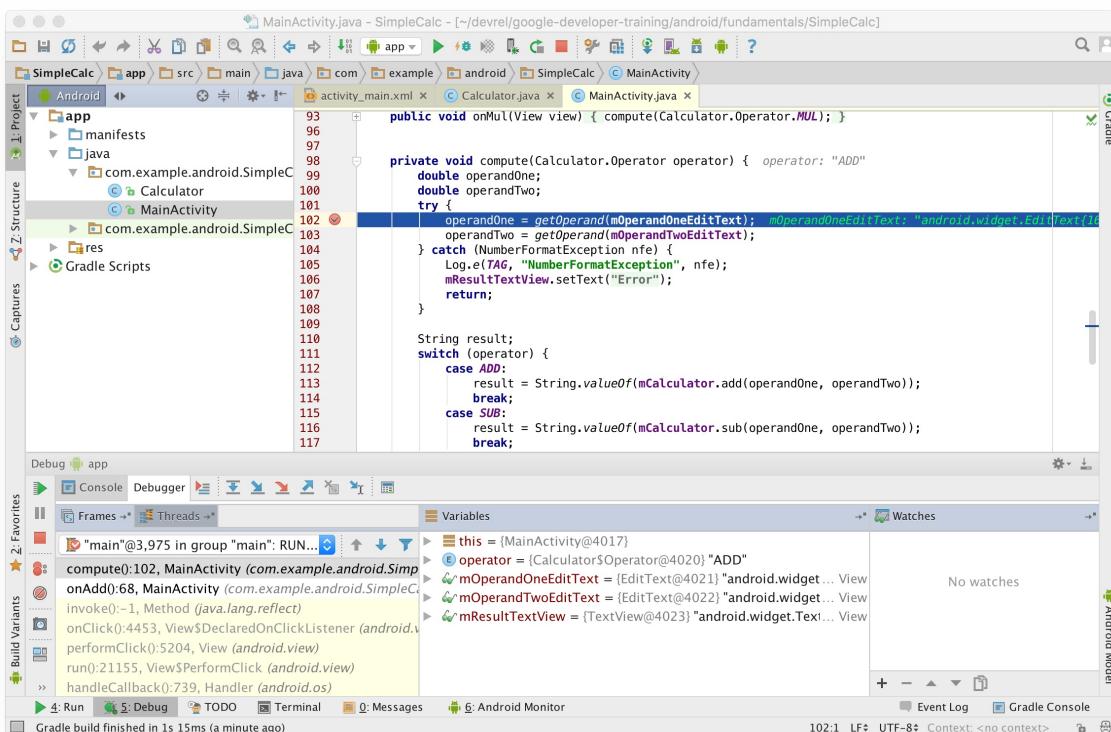
```

98     private void compute(Calculator.Operator operator) {
99         double operandOne;
100        double operandTwo;
101       try {
102           operandOne = getOperand(mOperandOneEditText);
103           operandTwo = getOperand(mOperandTwoEditText);
104       } catch (NumberFormatException nfe) {
105           Log.e(TAG, "NumberFormatException", nfe);
106           mResultTextView.setText("Error");
107       }
108
109       String result;
110       switch (operator) {
111

```

- In the SimpleCalc app on a device, enter numbers in the EditText views and click one of the calculate buttons.

The execution of your app stops when it reaches the breakpoint you set, and the debugger shows the current state of your app at that breakpoint.



- Examine the the Debug window. It includes these parts:
- Frames panel:** shows the current execution stack for a given thread. The execution stack shows each class and method that have been called in your app and in the

Android runtime, with the most recent method at the top. Threads appear in a drop down menu. Note that your app is currently running in the main thread, and that the app is executing the `compute()` method in `MainActivity`.

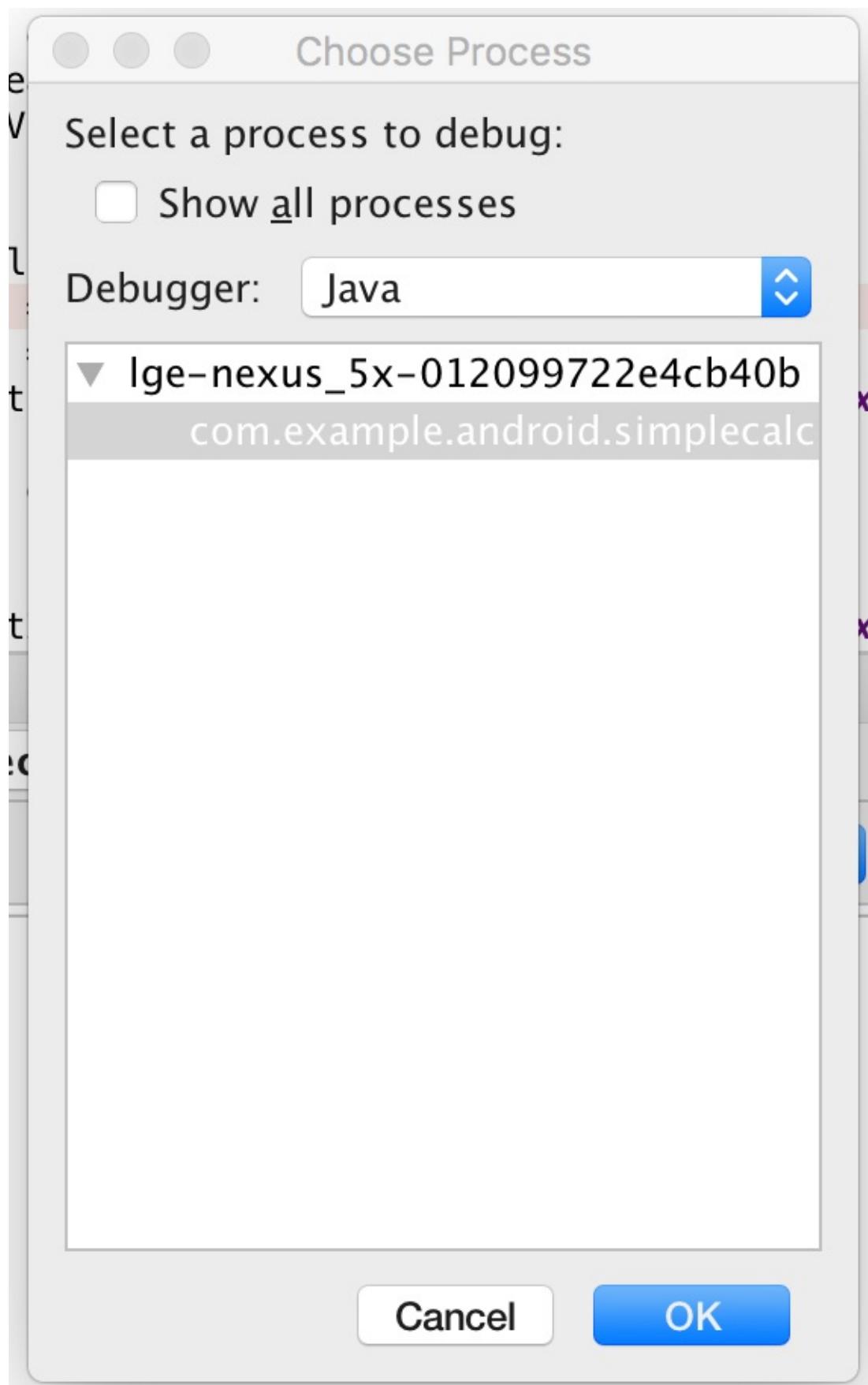
7. **Variables panel:** displays the variables in the current execution scope and their values. At this stage of your app's execution, the available variables are `this` (for the activity), `operator` (the operator name from `Calculator.Operator` the method was called from), as well as the global variables for the `EditTexts` and the `TextView`. Each variable in this panel has a disclosure triangle to allow you to view the properties of the objects contained in those variables. Try expanding a variable to explore its properties.
8. **Watches panel:** displays the values for any variable watches you have set. Watches allow you to keep track of a specific variable in your program, and see how that variable changes as your program runs.
9. Resume your app's execution with **Run > Resume Program** or click the **Resume**  icon on the left side of the debugger window.

The SimpleCalc app continues running, and you can interact with the app until the next time code execution arrives at the breakpoint.

## 2.2 Debug a running app

If your app is already running on a device or emulator, and you decide you want to debug that app, you can switch an already running app to debug mode.

1. Run the SimpleCalc app normally, with the **Run**  icon.
2. Select **Run > Attach debugger to Android process** or click the **Attach**  icon in the toolbar.
3. Select your app's process from the dialog that appears. Click **OK**.



The

Debug window appears, and you can now debug your app as if you had started it in debug mode.

**Note:** If the Debug window does not automatically appear, click the Debug tab at the

bottom of the screen, and then the Debugger tab.

## Task 3. Explore Debugger Features

In this task we'll explore the various features in the Android Studio debugger, including executing your app line by line, working with breakpoints, and examining variables.

### 3.1 Step through your app's execution

After a breakpoint, you can use the debugger to execute each line of code in your app one at a time, and examine the state of variables as the app runs.

1. Debug your app in Android Studio, with the breakpoint you set in the last task.
2. In the app, enter numbers in both EditText views and click the Add button.

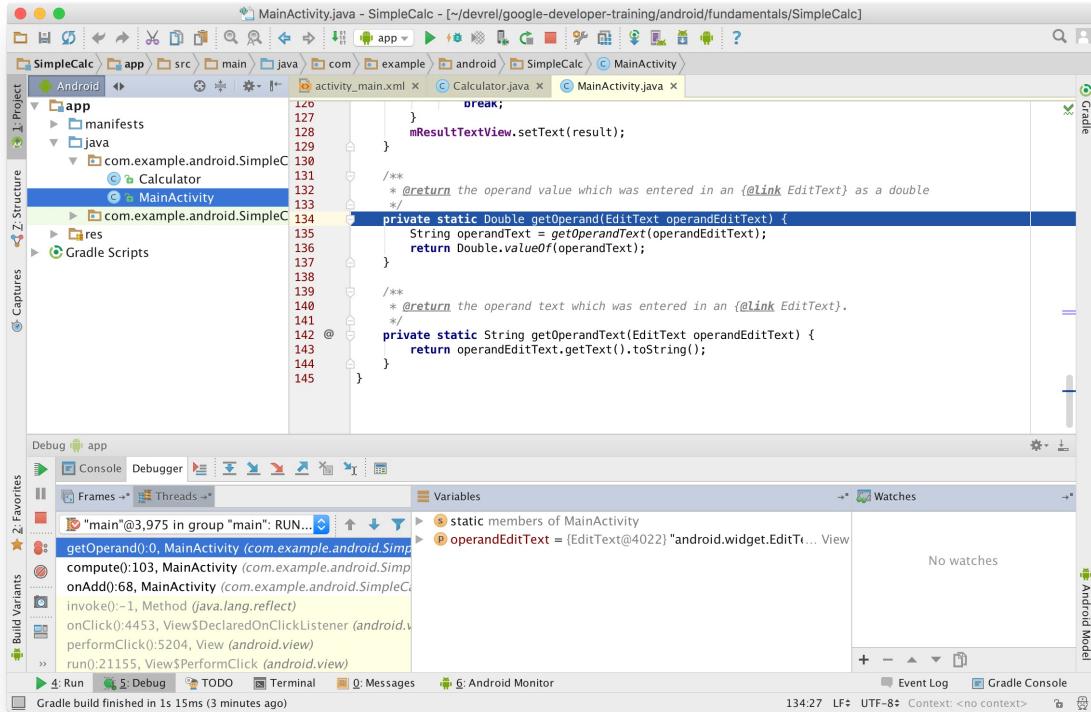
Your app's execution stops at the breakpoint you set earlier, and the debugger shows the current state of the app. The current line is highlighted in your code.

3. Click the **Step Over**  button at the top of the debugger window.

The debugger executes the current line in the compute() method (where the breakpoint is, the assignment for operandOne), and the highlight moves to the next line in the code (the assignment for operandTwo). The Variables panel updates to reflect the new execution state, and the current values of variables also appears after each line of your source code in italics.

You can also use **Run > Step Over**, or F8, to step over your code.

4. At the next line (the assignment for operandTwo), click the **Step Into**  icon. Step Into jumps into the execution of a method call in the current line (versus just executing that method and remaining on the same line). In this case, because that assignment includes a call to getOperand(), the debugger scrolls the MainActivity code to that method definition.



You can also use **Run > Step Into**, or F7, to step into a method.

5. Click **Step Over** to run each of the lines in `getOperand()`. Note that when the method completes the debugger returns you to the point where you first stepped into the method, and all the panels update with the new information.
6. Use **Step Over** twice to move the execution point to the first line inside the case statement for ADD.
7. Click **Step Into** . The debugger executes the appropriate method defined in the `Calculator` class, opens the `Calculator.java` file, and scrolls to the execution point in that class. Again, the various panels update to reflect the new state.
8. Use the **Step Out** icon to execute the remainder of that calculation method and pop back out to the `compute()` method in `MainActivity`. You can then continue debugging the `compute()` method from where you left off. You can also use **Run > Step Out** or Shift-F8 to step out of a method execution.

## 3.2 Work with Breakpoints

Use breakpoints to indicate where in your code you want to interrupt your app's execution to debug that portion of that app.

1. Find the breakpoint you set in the last task at the start of the `compute()` method in `MainActivity`.
2. Add a breakpoint to the start of the switch statement.

1. Find the breakpoint you set in the last task at the start of the compute() method in MainActivity.
2. Add a breakpoint to the start of the switch statement.
3. Right-click on that new breakpoint and enter the following test in the Condition field:

```
(operandOne == 42) || (operandTwo == 42)
```

4. Click **Done**.

This second breakpoint is a *conditional* breakpoint. The execution of your app will only stop at this breakpoint if the test in the condition is true. In this case, the expression is only true if one or the other operands you entered is 42. You can enter any Java expression as a condition as long as it returns a boolean.

5. Run your app in debug mode (**Run > Debug**), or click **Resume**  if it is already running. In the app, enter two numbers other than 42 and click the Add button. Execution halts at the first breakpoint in the compute() method.
6. Click **Resume** to continue debugging the app. Observe that execution did not stop at your second breakpoint, because the condition was not met.
7. Right click the first breakpoint and uncheck **Enabled**. Click **Done**. Observe that the breakpoint icon now has a green dot with a red border.

Disabling a breakpoint enables you to temporarily "mute" that breakpoint without actually removing it from your code. If you remove a breakpoint altogether you also lose any conditions you created for that breakpoint, so disabling it may often be a better choice.

You can also mute all breakpoints in your app at once with the **Mute Breakpoints**  icon.

8. In the app, enter 42 in the first EditText and click any button. Observe that the conditional breakpoint at the switch statement halts execution (the condition was met.)
9. Click the View Breakpoints icon on the left edge of the debugger window. The Breakpoints window appears.

The Breakpoints window enables you to view all the breakpoints in your app, enable or disable individual breakpoints, and add additional features of breakpoints including conditions, dependencies on other breakpoints, and logging.

10. Click **Done** to close the breakpoints window.

### 3.3 Examine and modify variables

2. In the app, enter two numbers, one of them 42, and click the Add button.

The first breakpoint in compute() is still muted. Execution stops at the second breakpoint (at the switch statement), and the debugger appears.

3. Observe in the Variables panel that the operandOne and operandTwo variables have the values you entered into the app.
4. Observe that the this variable is a MainActivity object. Click the disclosure arrow to view the member variables of that object.
5. Right-click the operandOne variable in the Variables panel, and select **Set Value**. You can also use F2.
6. Change the value of operandOne to 10 and press Return.
7. Modify the value of operandTwo to 10 in the same way and press Return.
8. Click the Resume icon to continue running your app. Observe that the result in the app is now 20, based on the variable values you changed in the debugger.
9. In the app, click the Add button. Execution halts at the breakpoint.
10. Click the Evaluate Expression  icon, or select **Run > Evaluate Expression**. The Evaluate Expression window appears. You can also right-click on any variable and choose Evaluate Expression.

Use Evaluate Expression to explore the state of variables and objects in your app, including calling methods on those objects. You can enter any code into this window.

11. Type mOperandOneEditText.getHint(); into the Expression window and click Evaluate.
12. The Evaluate Expression window updates with the result of that expression. The hint for this EditText is the string "Type Operand 1", as was originally defined in the XML for that EditText.

Note that the result you get from evaluating an expression is based on the app's current state. Depending on the values of the variables in your app at the time you evaluate expressions, you may get different results. Note also that if you use Evaluate Expression to change the values of variables or object properties, you change the running state of the app.

13. Click **Close** to hide the Evaluate Expression window.

## Coding challenge

In Task 1.3, you tried running the SimpleCalc app with no values in either of the EditText views, resulting in an error message. Use the debugger to step through the execution of the code and determine precisely why this error occurs. Fix the bug that causes this error.

In Task 1.3, you tried running the SimpleCalc app with no values in either of the EditText views, resulting in an error message. Use the debugger to step through the execution of the code and determine precisely why this error occurs. Fix the bug that causes this error.

## Conclusion

In this chapter, you learned about the Android Studio debugger, and in particular:

- How to run your app in debug mode.
- How to step through the execution of your app.
- How to set, mute and manage breakpoints.
- How to view and change the values of variables as your app runs.

## Resources

- [Debug Your App \(Android Studio User Guide\)](#)
- [Debugging and Testing in Android Studio \(video\)](#)

## 3.2 P: Testing Apps With Unit Tests

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Explore and run CalculatorTest in Android Studio](#)
- [Task 2. Add more unit tests in CalculatorTest](#)
- [Coding challenges](#)
- [Summary](#)
- [Resources](#)

Even if you have an app that compiles and runs and looks the way you want it to on different devices, it's hard to make sure that your app will behave the way you expect it to in every situation, especially as that app grows and becomes more complex. Even if you try to manually test your app every time you make a change—a tedious prospect all on its own—you'll likely miss something or just not anticipate what another user might do with your app to cause it to fail.

Writing and running tests is a critical part of the software development process. "Test-Driven Development" (TDD) is a popular software development philosophy that places tests at the core of all software development for an application or service.

Testing your code can help you catch bugs early on in development and improve the robustness of your code as your app gets larger. With tests in your code you can exercise small portions of your app in isolation, and in an automated and repeatable manner. The code you write to test your app doesn't end up in the production version of your app; it lives only on your development machine, alongside your app's code in Android Studio.

Android Studio and the Android Testing Support Library support several different kinds of tests and testing frameworks. Two of the simplest forms of testing Android Studio supports are local unit tests and instrumented tests.

*Local unit tests* are tests that are compiled and run entirely on your local machine with the Java Virtual Machine (JVM). Use local unit tests to test the parts of your app (such as the internal logic) that do not need access to the Android framework or an Android device or emulator, or those for which you can create fake (mock) objects that pretend to behave like the framework equivalents. Unit tests are written with JUnit, a common unit testing framework for Java.

*Instrumented tests* are tests that run on an Android device or emulator. These tests have access to the Android framework and to [Instrumentation](#) information such as the app's [Context](#). You can use instrumented tests for unit testing, user interface (UI) testing, or making sure the components of your app interact correctly. Most commonly, however, you use instrumentation for UI testing, which allows you to test that your app behaves correctly when a user interacts with your app's activities or enters a specific input.

In this practical you'll explore Android Studio's built-in functionality for testing, and learn how to write and run unit tests. For user interface testing, Android uses the Espresso framework, which enables you to write automated UI tests. You'll learn about Espresso in a later practical.

## What you should already KNOW

From the previous practicals you should be familiar with:

- How to create projects in Android Studio.
- The major components of your Android Studio project (manifest, resources, Java files, gradle files).
- How to build and run apps.

## What you will LEARN

- How organizing and running tests works in Android Studio
- What a unit test is, and how to write unit tests for your code.
- How to create and run local unit tests in Android Studio.

## What you will DO

- Run the initial tests in the SimpleCalc app.
- Add more tests to the SimpleCalc app.
- Run those unit tests to see the results.

## App Overview

This practical uses the same SimpleCalc app from the last section. You can modify that app in place, or [copy your project](#) into a new app.

# Task 1. Explore and run CalculatorTest in Android Studio

You both write and run your tests (both unit tests and instrumented tests) inside Android Studio, alongside the code for your app. Every new Android project includes very basic sample classes for testing that you can extend or replace for your own uses.

In this task we'll return to the SimpleCalc app, which includes a basic unit testing class.

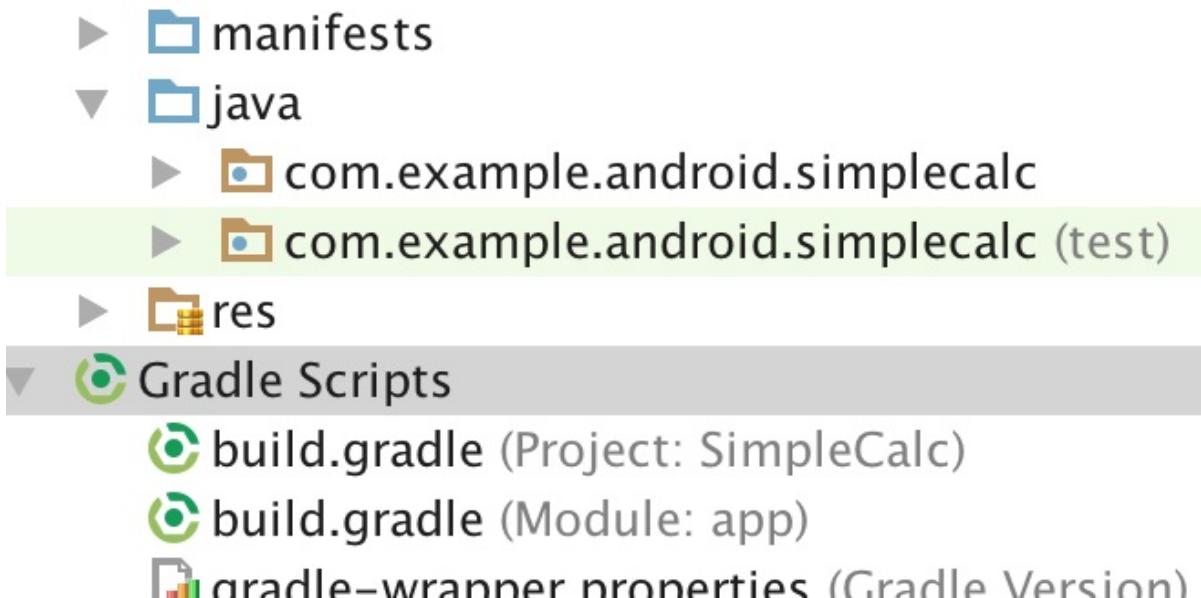
## 1.1 Explore source sets and CalculatorTest

*Source sets* collect related code in your project for different build targets or other "flavors" for your app. When Android Studio creates your project, it creates three source sets:

- The main source set, for your app's code and resources.
- The **test** source set, for your app's local unit tests.
- The **androidTest** source set, for Android instrumented tests.

In this task you'll explore how source sets are displayed in Android Studio, examine the gradle configuration for testing, and run the unit tests for the SimpleCalc app. You'll use the androidTest source set in more detail in a later practical.

1. Open the SimpleCalc project in Android Studio if you have not already done so. If you don't have SimpleCalc you can find it at this [download link](#).
2. Open the Project view, and expand the **app** and **java** folders.



### 3. Expand the **com.android.example.simplecalc (test)** folder.

This folder is where you put your app's local unit tests. Android Studio creates a sample test class for you in this folder for new projects, but for SimpleCalc the test class is called `CalculatorTest`.

4. Open **CalculatorTest.java**.
5. Examine the code and note the following:
  6. The only imports are from the `org.junit`, `org.hamcrest`, and `android.test` packages. There are no dependencies on the Android framework classes here.
  7. The `@RunWith(JUnit4.class)` annotation indicates the runner that will be used to run the tests in this class. A test runner is a library or set of tools that enables testing to occur and the results to be printed to a log. For tests with more complicated setup or infrastructure requirements (such as Espresso) you'll use different test runners. For this example we're using the basic JUnit4 test runner.
  8. The `@SmallTest` annotation indicates that all the tests in this class are unit tests that have no dependencies, and run in milliseconds. The `@SmallTest`, `@MediumTest`, and `@LargeTest` annotations are conventions that make it easier to bundle groups of tests into suites of similar functionality.
  9. The `setUp()` method is used to set up the environment before testing, and includes the `@Before` annotation. In this case the setup creates a new instance of the `Calculator` class and assigning it to the `mCalculator` member variable.
  10. The `addTwoNumbers()` method is an actual test, and is annotated with `@Test`. Only methods in a test class that have an `@Test` annotation are considered tests to the test runner. Note that by convention test methods do not include the word "test."
  11. The first line of `addTwoNumbers()` calls the `add()` method from the `Calculator` class. Note that you can only test methods that are public or package-protected. In this case the `Calculator` is a public class with public methods, so all is well.

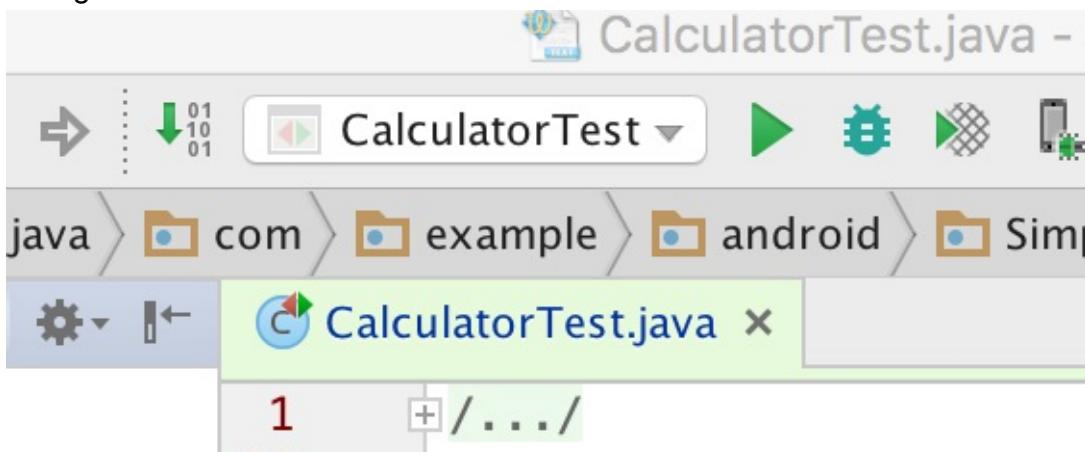
- runner. Note that by convention test methods do not include the word "test."
11. The first line of `addTwoNumbers()` calls the `add()` method from the `Calculator` class. Note that you can only test methods that are public or package-protected. In this case the `Calculator` is a public class with public methods, so all is well.
  12. The second line is the assertion for the test. Assertions are expressions that must evaluate to true for the test to pass. In this case the assertion is that the result you got from the `add` method ( $1 + 1$ ) matches the given number 2. You'll learn more about how to create assertions later in this practical.

## 1.2 Run tests in Android Studio

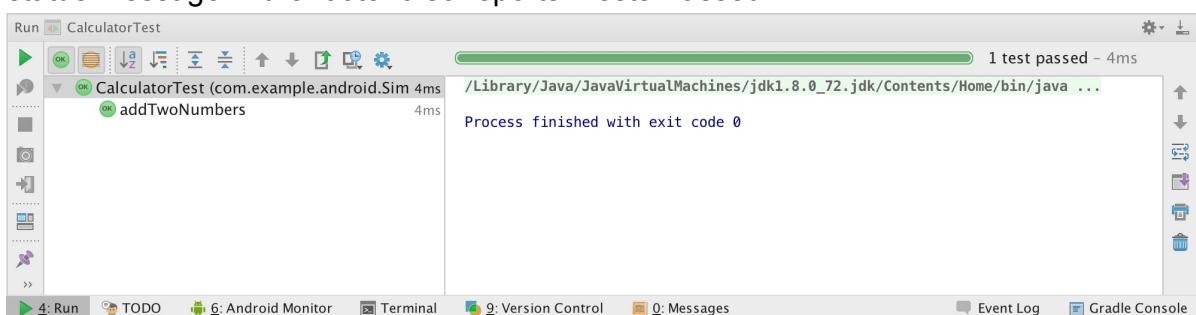
In this task you'll run the unit tests in the test folder and view the output for both successful and failed tests.

1. In the project view, right-click the `CalculatorTest` class and select **Run 'CalculatorTest'**.

The project builds, if necessary, and the testing view appears at the bottom of the screen. At the top of the screen, the dropdown (for available run configurations) also changes to **CalculatorTest**.



All the tests in the `CalculatorTest` class run, and if those tests are successful, the progress bar at the top of the view turns green. (In this case, there is currently only the one test.) A status message in the footer also reports "Tests Passed."



2. In the `CalculatorTest` class, change the assertion in `addTwoNumbers()` to:

The test runs again as before, but this time the assertion fails (3 is not equal to 1 + 1.) The progress bar in the run view turns red, and the testing log indicates where the test (assertion) failed and why.

1. Change the assertion in `addTwoNumbers()` back to the correct test and run your tests again to ensure they pass.
2. In the run configurations dropdown, select **app** to run your app normally.

## Task 2. Add more unit tests in CalculatorTest

With unit testing you take a small bit of code in your app such as a method or a class, isolate it from the rest of your app, and write tests to make sure that one small bit works in the way you expect. Typically unit tests call a method with a variety of different inputs, and verify that that method does what you expect and returns what you expect it to return.

In this task you'll learn more about how to construct unit tests. You'll write additional unit tests for the methods in the Calculator utility methods in the SimpleCalc app, and run those tests to make sure they produce the output you expect.

**Note:** Unit testing, test-driven development and the JUnit 4 API are all large and complex topics and outside the scope of this course. See the [Resources](#) for links to more information.

### 2.1 Add more tests for the add() method

Although it is impossible to test every possible value that the `add()` method may ever see, it's a good idea to test for input that might be unusual. For example, consider what happens if the `add()` method gets arguments:

- With negative operands.
- With floating-point numbers.
- With exceptionally large numbers.
- With operands of different types (a float and a double, for example)
- With an operand that is zero.
- With an operand that is infinity.

The base SimpleCalc app provides a single unit test for you (`addTwoNumbers()`), which you ran in the last section. In this task we'll add more unit tests for the `add()` method to test different kinds of inputs.

1. Add a new method to `CalculatorTest` called `addTwoNumbersNegative()`. Use this skeleton:

The base SimpleCalc app provides a single unit test for you (`addTwoNumbers()`), which you ran in the last section. In this task we'll add more unit tests for the `add()` method to test different kinds of inputs.

1. Add a new method to `CalculatorTest` called `addTwoNumbersNegative()`. Use this skeleton:

```
@Test  
public void addTwoNumbersNegative() {  
}
```

This test method has a similar structure to `addTwoNumbers`: it is a public method, with no parameters, that returns void. It is annotated with the `@Test` annotation, which indicates it is a single unit test.

Note that this is a new method in the `CalculatorTest` class that is just an additional test for the `add()` method. Why not just add more assertions to `addTwoNumbers`? Grouping more than one assertion into a single method can make your tests harder to debug if only one assertion fails, and obscures the tests that do succeed. The general rule for unit tests is to provide a test method for every individual assertion.

2. Run all tests in `CalculatorTests`, as before.

In the test window both `addTwoNumbers` and `addTwoNumbersNegative` are listed as available (and passing) tests in the left panel. The `addTwoNumbersNegative` test still passes even though it doesn't contain any code -- a test that does nothing is still considered a successful test.

3. Add a line to invoke the `add()` method in the `Calculator` class with a negative operand.

```
double resultAdd = mCalculator.add(-1d, 2d);
```

The "d" notation after each operand indicates that these are numbers of type double. Since the `add()` method is defined with double parameters, floats or ints will also work. Indicating the type explicitly enables you to test other types separately, if you need to.

4. Add an assertion with `assertThat()`.

```
assertThat(resultAdd, is(equalTo(1d)));
```

The `assertThat()` method is a JUnit4 assertion that claims the expression in the first argument is equal to the one in the second argument. Older versions of JUnit used more specific assertion methods (`assertEquals()`, `assertNull()`, `assertTrue()`), but `assertThat()` is a more flexible, more debuggable and often easier to read format.

In this case the assertion is that the result of the add() operation (-1 + 2) is equal to 1.

5. Add a new unit test to CalculatorTest for floating-point numbers:

```
@Test
public void addTwoNumbersFloats() {
    double resultAdd = mCalculator.add(1.111f, 1.111d);
    assertThat(resultAdd, is(equalTo(2.222d)));
}
```

Again, a very similar test to the previous test method, but with one argument to add() that is explicitly type float rather than double. The add() method is defined with parameters of type double, so you can call it with a float type, and that number is promoted to a double.

6. Run all tests in CalculatorTests, as before.

7. Click **Run**  to run all the tests again.

This time the test failed, and the progress bar is red. This is the important part of the error message:

```
java.lang.AssertionError:
Expected: is <2.222>
but: was <2.2219999418258665>
```

Arithmetic with floating-point numbers is inexact, and the promotion resulted in a side effect of additional precision. The assertion in the test is technically false: the expected value is not equal to the actual value.

The question here is: when you have a precision problem with promoting float arguments is that a problem with your code, or a problem with your test? In this particular case both input arguments to the add() method from the Calculator app will always be type double, so this is an arbitrary and unrealistic test. But if your app was written such that that input to the add() method could be either double or float and you only care about *some* precision, you need to provide some wiggle room to the test so that "close enough" counts as a success.

8. Change the assertThat() method to use the closeTo matcher:

```
assertThat(resultAdd, is(closeTo(2.222, 0.01)));
```

For this test, rather than testing for exact equality you can test for equality within a specific delta. In this case the closeTo() matcher method takes two arguments: the expected value and the amount of delta. Here that delta is just two decimal points of precision.

```
assertThat(resultAdd, is(closeTo(2.222, 0.01)));
```

For this test, rather than testing for exact equality you can test for equality within a specific delta. In this case the `closeTo()` matcher method takes two arguments: the expected value and the amount of delta. Here that delta is just two decimal points of precision.

## 2.2 Add unit tests for the other calculation methods

Use what you learned in the previous task to fill out the unit tests for the `Calculator` class.

1. Add a unit test called `subTwoNumbers()` that tests the `sub()` method.
2. Add a unit test called `subWorksWithNegativeResults()` that tests the `sub()` method where the given calculation results in a negative number.
3. Add a unit test called `mulTwoNumbers()` that tests the `mul()` method.
4. Add a unit test called `mulTwoNumbersZero()` that tests the `mul` method with at least one argument as zero.
5. Add a unit test called `divTwoNumbers()` that tests the `div()` method with two non-zero arguments.
6. **Challenge:** Add a unit test called `divByZero()` that tests the `div()` method with a second argument of 0. Hint: Try this in the app first to see what the result is.

**Solution Code:**

```

@Test
public void addTwoNumbers() {
    double resultAdd = mCalculator.add(1d, 1d);
    assertThat(resultAdd, is(equalTo(2d)));
}

@Test
public void addTwoNumbersNegative() {
    double resultAdd = mCalculator.add(-1d, 2d);
    assertThat(resultAdd, is(equalTo(1d)));
}

@Test
public void addTwoNumbersFloats() {
    double resultAdd = mCalculator.add(1.111f, 1.111d);
    assertThat(resultAdd, is(closeTo(2.222, 0.01)));
}

@Test
public void subTwoNumbers() {
    double resultSub = mCalculator.sub(1d, 1d);
    assertThat(resultSub, is(equalTo(0d)));
}

@Test
public void subWorksWithNegativeResult() {
    double resultSub = mCalculator.sub(1d, 17d);
    assertThat(resultSub, is(equalTo(-16d)));
}

@Test
public void mulTwoNumbers() {
    double resultMul = mCalculator.mul(32d, 2d);
    assertThat(resultMul, is(equalTo(64d)));
}

@Test
public void divTwoNumbers() {
    double resultDiv = mCalculator.div(32d, 2d);
    assertThat(resultDiv, is(equalTo(16d)));
}

@Test
public void divTwoNumbersZero() {
    double resultDiv = mCalculator.div(32d, 0);
    assertThat(resultDiv, is(equalTo(Double.POSITIVE_INFINITY)));
}

```

## Coding challenges

1. Dividing by zero is always worth testing for, because it a special case in arithmetic. If you try to divide by zero in the current version of the SimpleCalc app, it behaves the way Java is defined: Dividing a number by returns the "Infinity" constant (Double.POSITIVE\_INFINITY). Dividing 0 by 0 returns the not a number constant

- (Double.NaN). Although these values are correct for Java, they're not necessarily useful values for the user in the app itself. How might you change the app to more gracefully handle divide by zero? To accomplish this challenge, start with the test first -- consider what the right behavior is, and then write the tests as if that behavior already existed. Then change or add to the code so that it makes the tests come up green.
2. Sometimes it's difficult to isolate a unit of code from all of its external dependencies. Rather than artificially organize your code in complicated ways just so it can be more easily tested, you can use a mocking framework to create fake ("mock") objects that pretend to be dependencies. Research the [Mockito](#) framework, and learn how to set it up in Android Studio. Write a test class for the calcButton() method in SimpleCalc, and use Mockito to simulate the Android context in which your tests will run.

## Summary

In this practical you explored Android Studio's built-in features for running local unit tests, and wrote a handful of your own unit tests for the SimpleCalc program. From here you'll build on your knowledge of testing in an upcoming practical when you'll learn about Android instrumented tests and the Espresso UI testing framework.

## Resources

- [Best Practices for Testing](#)
- [Getting Started with Testing](#)
- [Building Local Unit Tests](#)
- [JUnit 4 Home Page](#)
- [JUnit 4 API Reference](#)
- [Mockito Home Page](#)
- [Android Testing Support - Testing Patterns \(video\)](#)
- [Android Testing Codelab](#)
- [Android Tools Protip: Test Size Annotations](#)
- [The Benefits of Using assertThat over other Assert Methods in Unit Tests](#)

# 3.3 P: Using The Android Support Libraries

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Set up your project to use support libraries](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

The Android SDK includes several libraries collectively called the Android support library. These libraries provide a number of features that are not built into the Android framework, including:

- Backward-compatible versions of framework components: The support library allows apps running on older versions of the Android platform to support features made available on newer versions of the platform. For example, older devices may not have access to features such as fragments, action bars, or the material design elements. Using the support library provides support for the widest possible set of devices -- when in doubt, always use the support library.
- Additional layout and user interface elements: The support library includes views and layouts that can be useful for your app, but are not included in the standard Android framework. For example, the [RecyclerView](#) view that you will use in a later chapter is part of the support library.
- Support for different device form factors, such as TV or wearables: For example, the [Leanback](#) library includes components specific to app development on TV devices.
- Design support: The [design support library](#) includes components to support material design elements in your app, including floating action buttons (FAB).
- Various other features such as palette support, annotations, percentage-based layout dimensions, and preferences.

When you use the Android support libraries for backward-compatibility, you do not need to check for the Android SDK level or build number in your code or create different versions of your app based on the Android version. The support library manages those system checks

for you and picks the best possible implementation -- the one provided by the framework itself, or an implementation the library itself provides.

In a previous chapter you configured your project to use the RecyclerView support library, and learned how to use the RecyclerView classes in your code. In this chapter you'll learn more about how to use the support libraries in your project.

## What you should already KNOW

From the previous practicals you should be familiar with:

- How to create projects in Android Studio.
- The major components of your Android Studio project (manifest, resources, Java files, gradle files).

## What you will LEARN

- How to verify that the the Android support libraries are available in Android Studio.
- How to find the dependency name for the library you're interested in.
- How to add support library dependencies to your app's gradle build files.
- How to indicate support library classes in your app.
- Where to find more information on the Android support libraries.

## What you will DO

- Verify the Android support libraries are available in Android Studio.
- Find a dependency statement for a support library.
- Modify the build files of your app to include a dependency.
- Explore the support library resources.

## App Overview

This practical does not use a specific app. You can use any app, such as the Hello Toast app that you built in an earlier chapter.

## Task 1. Set up your project to use support libraries

The Android support libraries are downloaded as part of the Android SDK, and available in the Android SDK manager. In Android Studio, you'll use the Android Support Repository—the local repository for the support libraries—to get access to the library from within your gradle build files. In this task you'll verify that the Android Support Repository is downloaded and available for your projects.

## 1.1 Verify that the Android Support Library is available

1. In Android Studio, select **Tools > Android > SDK Manager**, or click the SDK Manager icon.

The SDK Manager preference pane appears.

2. Click the **SDK Tools** tab, and look for Android Support Repository in the list.
3. If **Installed** appears in the Status column, you're all set. Click **Cancel**.
4. If **Not installed** appears, or an update is available, click the checkbox next to Android Support Repository. A download icon should appear next to the checkbox. Click **OK**.  
**Note:** You may also see "Android Support Library" in the SDK Tools list. This is an older version of the support libraries that is now obsolete. (You don't need to remove it if it exists.)
5. Click **OK** again, and then **Finish** when the support repository has been installed.

## 1.2 Find a library dependency statement

To provide access to a support library from your project, you must add that library to your gradle build file as a dependency. Dependency statements have a specific format that includes the name and version number of the library.

1. Visit the [Support Library Features](#) page on developer.android.com.
2. Find the library you're interested in on that page, for example, the [Design Support Library](#) for material design support.
3. Copy the dependency statement shown at the end of the section. For example, the dependency for the design support library looks like this:

```
com.android.support:design:23.3.0
```

Note that the number at the end of the line may vary.

## 1.3 Add the dependency to your build.gradle file

The gradle scripts for your project manage how your app is built, including specifying the dependencies your app has on other libraries. To add a support library to your project you modify your gradle build files to include a dependency to that library.

1. In Android Studio, make sure the **Project** pane is open and the **Android** tab is clicked.
2. Expand **Gradle Scripts**, if necessary, and open the **build.gradle (Module: app)** file.

Note that build.gradle for the overall project (build.gradle (Project: app\_name)) is a different file from the build.gradle for the app module.

3. Locate the `dependencies` section of build.gradle, near the end of the file.

The dependencies section for a new project already includes dependencies for JUnit and the v7 AppCompat support library. Those dependencies look like this:

```
testCompile 'junit:junit:4.12'  
compile 'com.android.support:appcompat-v7:23.1.1'
```

4. Add a dependency for the support library that includes the statement you copied in the previous task. For example, a complete dependency on the design support library looks like this:

```
compile 'com.android.support:design:23.3.0'
```

5. Update the version number, if necessary.

If the version number you specified is lower than the currently available library version number, Android Studio will warn you and tell you of the new version ("a newer version of `com.android.support:design` is available"). Edit the version number to the updated version.

6. Click **Sync Now** to sync your updated gradle files with the project, if prompted.

## Coding challenge

Start with the Hello Toast app, and modify it to include the following features provided by support libraries:

- Change the layout to use percentage-based dimensions.
- Add a navigation drawer with several options named for colors in the material design palette. ("Red", "Green", "Teal," and so on)
- Change the color of the background (currently yellow) to the color option the user picked in the nav drawer.

- (Extra Credit) Change the toast to be a Snackbar. Use a Coordinator Layout to adjust the FAB's position when the Snackbar is displayed (the Snackbar should not overlay the FAB).

## Summary

In this chapter, you learned about the Android support libraries, and in particular:

- How to ensure Android Studio has access to the Android support repository.
- Where to find the gradle dependency statements for various support libraries.
- How to add those support libraries to your app's build.gradle file.

## Resources

- [Android Support Library](#) (introduction)
- [Support Library Setup](#)
- [Support Library Features](#)
- [API Reference](#) (all packages that start with android.support)

# 4.1 P: Using Keyboards, Input Controls, Alerts, and Pickers

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App Overview
- Task 1: Experiment with text entry keyboard attributes
- Task 2: Change the keyboard type
- Coding Challenge
- Task 3: Add a spinner input control for selecting a phone label
- Task 4: Use a dialog for an alert requiring a decision
- Task 5: Use a picker for user input
- Summary
- Resources

You can customize input methods to make entering data easier for users. In this practical, you'll learn how to use different on-screen keyboards and controls for user input, to show an alert message that users can interact with, and to provide interface elements for selecting a time and date.

## What you should already KNOW

- Creating an Android Studio project from a template and generating the main layout.
- Running apps on the emulator or a connected device.
- Making a copy of an app project, and renaming the app.
- Creating and editing UI elements using the Layout Editor and XML code.
- Accessing UI elements from your code using `findViewById()`.
- Converting the text in a view to a string using `getText().toString()`.
- Handling a button click.
- Displaying a toast message.
- Starting an activity with another app using an implicit intent.
- Using an adapter to connect your data to a view, such as the RecyclerView in a previous lesson.

## What you will LEARN

- Changing the input methods to enable spelling suggestions, auto-capitalization, and password obfuscation.
- Changing the generic on-screen keyboard to a phone keypad or other specialized keyboards.
- Adding a spinner input control to show a dropdown menu with values, from which the user can select one.
- Adding an alert with OK and Cancel for a user decision.
- Using date and time pickers and recording the selections.

## What you will DO

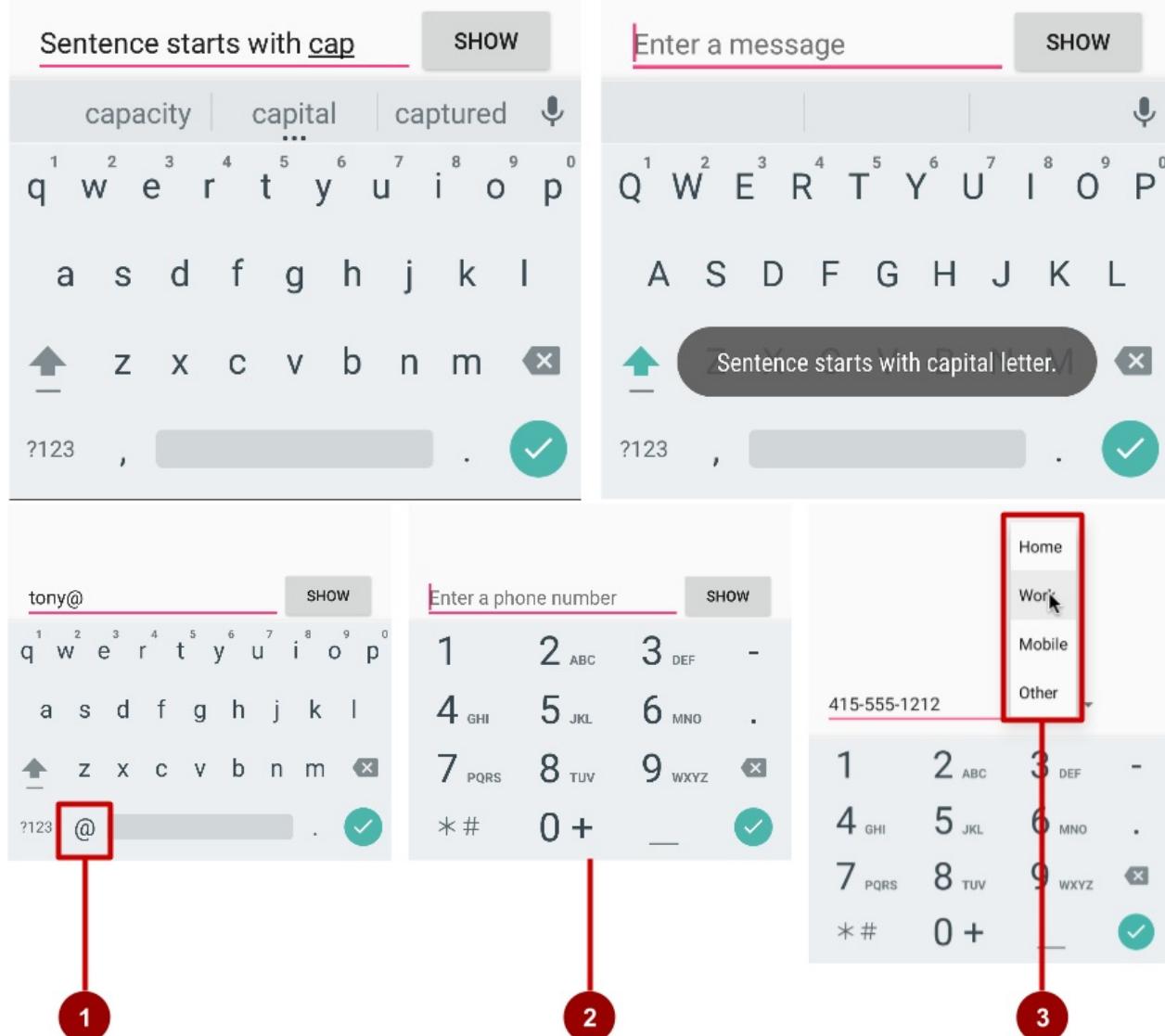
- Create new Android Studio projects to show keyboards, a spinner, an alert, and time and date pickers.
- Provide spelling suggestions when a user enters text, and automatically capitalize new sentences, by experimenting with the input method.
- Experiment with the input type attribute to change the on-screen keyboard to a special keyboard for entering email addresses, and then to a numeric keypad to force numeric entry.
- Add a spinner input control for the phone number field for selecting one value from a set of values.
- Create a new project with an alert dialog to notify the user to make a decision, such as OK or Cancel.
- Add the date picker and time picker to the new project, and use listeners to record the user's selection.

## Apps Overview

In this practical, you'll create and build a new app called **Keyboard Samples** for experimenting with the `android:inputType` attribute for the `EditText` UI element. You will change the keyboard so that it suggests spelling corrections and capitalizes each new sentence. To keep the app simple, you'll display the entered text in a `toast` message.

You will also change the keyboard to one that offers the "@" symbol in a prominent location for entering email addresses, and to a phone keypad for entering phone numbers. As a challenge, you will implement a listener for the action key in the keyboard in order to send an implicit intent to another app to dial the phone number.

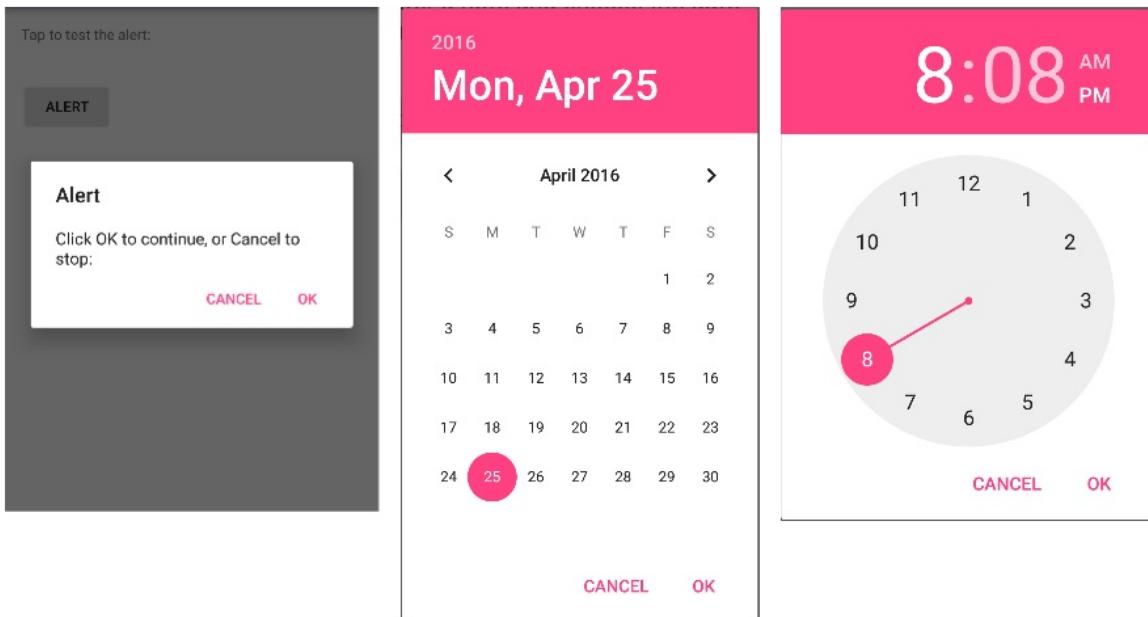
You will then copy the app to create Phone Number Spinner that offers a spinner input control for selecting the label (Home, Work, Other, Custom) for the phone number.



The figure above shows the following:

1. The email keyboard with the "@" symbol in an easy-to-find location
2. The phone keypad
3. The spinner

You'll also create Alert Sample to experiment with an alert dialog, and Date Time Pickers to experiment with a date picker and a time picker and use the selections in your app.



## Task 1. Experiment with text entry keyboard attributes

Touching an `EditText` editable text field places the cursor in the text field and automatically displays the on-screen keyboard. You will change attributes of the text entry field so that the keyboard suggests spelling corrections while you type, and automatically starts each new sentence with capital letters. For example:

- `android:inputType="textCapSentences"` : Sets the keyboard to capital letters at the beginning of sentences.
- `android:inputType="textAutoCorrect"` : Sets the keyboard to show automatic spelling corrections as you enter characters.
- `android:inputType="textMultiLine"` : Enables the Return key on the keyboard to end lines and create new blank lines without closing the keyboard.
- `android:inputType="textPassword"` : Sets the characters the user enters into dots to conceal the entered password.

Android Project: KeyboardSamples

### 1.1 Create the main layout and the `showText` method

You will add a Button, and change the TextView element to an `EditText` element so that the user can enter text.

1. Create a new project called **Keyboard Samples**, and choose the Empty Activity template.
2. Open the `activity_main.xml` layout file in order to edit the XML code.
3. Add a Button above the existing `TextView` element with the following attributes:

Button Attribute	New Value
<code>android:id</code>	<code>"@+id/button_main"</code>
<code>android:layout_width</code>	<code>"wrap_content"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:layout_alignParentBottom</code>	<code>"true"</code>
<code>android:layout_alignParentRight</code>	<code>"true"</code>
<code>android:onClick</code>	<code>"showText"</code>
<code>android:text</code>	<code>"Show"</code>

4. Extract the string resource for the `android:text` attribute value to create and entry for it in `strings.xml`: Place the cursor on **"Show"**, press Alt-Enter (Option-Enter on the Mac), and select **Extract string resources**. Then edit the Resource name for the string value to **show**.

**Why:** The string resource assignments are stored in the `strings.xml` file (under `app > res > values`). You can edit this file to change the string assignments so that the app can be localized with a different language.

1. Change the existing `TextView` element to an `EditText` element with the following attributes:

EditText Attribute	TextView Old Value	EditText New Value
<code>android:id</code>		<code>"@+id/editText_main"</code>
<code>android:layout_width</code>	<code>"wrap_content"</code>	<code>"match_parent"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>	<code>"wrap_content"</code>
<code>android:layout_alignParentBottom</code>		<code>"true"</code>
<code>android:layout_toLeftOf</code>		<code>"@+id/button_main"</code>
<code>android:hint</code>		<code>"Enter a message"</code>
<code>android:text</code>	<code>"Hello World!"</code>	(Remove this attribute)

2. Extract the string resource for the `android:hint` attribute value "Enter a message" to enter.
3. Open `MainActivity` and enter the following `showText` method, which retrieves the information entered into the `EditText` element and shows it in a `toast` message:

```
public void showText(View view) {  
    EditText editText = (EditText) findViewById(R.id.editText_main);  
    if (editText != null) {  
        String showString = editText.getText().toString();  
        Toast.makeText(this, showString, Toast.LENGTH_SHORT).show();  
    }  
}
```

4. Open `strings.xml` (in `app > res > values`), and edit the `app_name` value to "**Keyboard Samples**" (with a space between "Keyboard" and "Samples").
5. Run the app and examine how the keyboard works.

Tapping the **Show** button shows a toast message of the text entry.

To close the on-screen keyboard, tap the down-pointing arrow in the bottom row of icons.

In the standard keyboard layout, a checkmark icon appears in the lower right corner of the keypad, known as the Return (or Enter) key, to enter a new line. With the default attributes for the `EditText` element, tapping the Return key adds another line of text. In the next section, you will change the keyboard so that it capitalizes sentences as you type. As a result of setting the `android:inputType` attribute, the default attribute for the Return key changes to shift focus away from the `EditText` element and close the keyboard.

## 1.2 Set the keyboard to capitalize sentences

1. Add the `android:inputType` attribute to the `EditText` element using the `textCapSentences` value to set the keyboard to capital letters at the beginning of a sentence, so that users can automatically start a sentence with a capital letter:

```
    android:inputType="textCapSentences"
```

2. Run your app.

Capital letters will now appear on the keyboard at the beginning of sentences. When you tap the Return key on the keyboard, the keyboard closes and your text entry is finished. You can still tap the text entry field to add more text or edit the text. Tap **Show** to show the text in a toast message.

For details about the `android:inputType` attribute, see [Specifying the Input Method Type](#).

## 1.3 Set the keyboard to hide a password when entering it

1. Change the `EditText` element to use the `textPassword` value for the `android:inputType` attribute.
2. Change the `android:hint` to "Enter your password".
3. Run the app.

The characters the user enters turn into dots to conceal the entered password. For help, see [Text Fields](#).

## Task 2. Change the keyboard type

Every text field expects a certain type of text input, such as an email address, phone number, password, or just plain text. It's important to specify the input type for each text field in your app so that the system displays the appropriate soft input method, such as:

- The standard on-screen keyboard for plain text
- The keyboard for an email address which includes the "@" symbol in a prominent location
- The phone keypad for a phone number

### 2.1 Use an email keyboard

Modify the main activity's `EditText` element to show an email keyboard rather than a standard keyboard:

1. In the `EditText` element in the **activity\_main.xml** layout file, change the `android:inputType` attribute to the following:

```
    android:inputType="textEmailAddress"
```
2. Change the `android:hint` attribute to "Enter an email address".
3. Extract the string resource for the `android:hint` value to `enter_email`.
4. Run the app. Tapping the field brings up the on-screen email keyboard with the "@" symbol located next to the space key.

### 2.2 Use a phone keypad

Modify the main activity's `EditText` element to show a phone keypad rather than a standard keyboard:

1. In the `EditText` element in the **activity\_main.xml** layout file, change the

`android:inputType` attribute to the following:

```
    android:inputType="phone"
```

1. Change the `android:hint` attribute to "Enter a phone number".
2. Extract the string resource for the `android:hint` value to `enter_phone`.
3. Run the app.

Tapping the field now brings up the on-screen phone keypad in place of the standard keyboard.

**Note:** When running the app on the emulator, the field will still accept text rather than numbers if you type on the computer's keyboard. However, when run on the device, the field only accepts the numbers of the keypad.

```
<a id="codingchallenge1" />
```

## Coding challenge (optional)

You can also perform an action directly from the keyboard and replace the "return" key with an action key, such as for dialing a phone number. For this challenge, use the `android:imeOptions` attribute for the `EditText` component with the `actionSend` value:

```
    android:imeOptions="actionSend"
```

In the `onCreate()` method for this main activity, you can use `setOnEditorActionListener()` to set the listener for the `EditText` view to detect if the key is pressed:

```
EditText editText = (EditText) findViewById(R.id.editText_main);
if (editText != null)
    editText.setOnEditorActionListener(new TextView.OnEditorActionListener() {
        ...
    })
```

For help setting the listener, see "Specifying the Input Action" in [Handling Keyboard Input](#) and "Specifying Keyboard Actions" in [Text Fields](#).

You can use the `IME_ACTION_SEND` constant in the `EditorInfo` class to respond to the pressed key. In the example below, the key is used to call a method (`dialNumber()`) to dial the phone number:

```

@Override
public boolean onEditorAction(TextView textView, int actionId, KeyEvent keyEvent) {
    boolean mHandled = false;
    if (actionId == EditorInfo.IME_ACTION_SEND) {
        dialNumber();
        mHandled = true;
    }
    return mHandled;
}

```

To finish the challenge, create the `dialNumber()` method, which uses an implicit intent with `ACTION_DIAL` to pass the phone number to another app that can dial the number. It should look like this:

```

private void dialNumber() {
    EditText editText = (EditText) findViewById(R.id.editText_main);
    String mPhoneNum = null;
    if (editText != null) mPhoneNum = "tel:" + editText.getText().toString();
    Log.d(TAG, "dialNumber: " + mPhoneNum);
    Intent intent = new Intent(Intent.ACTION_DIAL);
    intent.setData(Uri.parse(mPhoneNum));
    if (intent.resolveActivity(getApplicationContext()) != null) {
        startActivity(intent);
    } else {
        Log.d("ImplicitIntents", "Can't handle this!");
    }
}

```

## Solution Code

To see the code for this challenge:

KeyboardDialPhone:

## Task 3. Add a spinner input control for selecting a phone label

Input controls are the interactive components in your app's user interface. Android provides a wide variety of controls you can use in your UI, such as buttons, seek bars, checkboxes, zoom buttons, toggle buttons, spinners, and many more. (For more information about input controls, see [Input Controls](#).)

A *spinner* provides a quick way to select one value from a set. Touching the spinner displays a drop-down list with all available values, from which the user can select one. If you are providing only two or three choices, you might want to use radio buttons for the choices if

you have room in your layout for them; however, with more than three choices, a spinner works very well, and takes up little room in your layout. On the other hand, if you have a long list of choices, a spinner may be too cumbersome and extend beyond your layout, forcing the user to scroll it.

For more information about spinners, see [Spinners](#).

To provide a way to select a label for a phone number (such as Home, Work, Mobile, and Other), you can add a spinner to the layout to appear right next to the phone number field.

### 3.1 Create a new project and modify the main activity's layout

1. Copy the **KeyboardSamples** project folder, rename it to **PhoneNumberSpinner**, and refactor it. (See the [Appendix](#) for instructions on copying a project.)
2. After refactoring, change the `<string name="app_name">` value in the **strings.xml** file (within **app > res > values**) to **Phone Number Spinner** (with spaces) as the app's name.
3. Open the **activity\_main.xml** layout file.
4. Enclose the `EditText` and `Button` elements within a `LinearLayout` with a horizontal orientation, placing the `EditText` element above the `Button` : [NESTED LINEAR LAYOUT]

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="@dimen/activity_vertical_margin"
    android:orientation="horizontal">
    <EditText
        ...
    <Button
        ...
    </LinearLayout>
```

5. Make the following changes to the `EditText` and `Button` elements (changes in bold):

<b>EditText Attribute</b>	<b>Value</b>
<code>android:id</code>	" <code>@+id/editText_main</code> "
<code>android:layout_width</code>	" <code>wrap_content</code> "
<code>android:layout_height</code>	" <code>wrap_content</code> "
<code>android:layout_alignParentBottom</code>	(Remove this attribute)
<code>android:inputType</code>	" <code>phone</code> "
<code>android:hint</code>	"Enter phone number"

Button Attribute	Value
android:id	"@+id/button_main"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
<b>android:layout_alignParentBottom</b>	<b>(Remove this attribute)</b>
<b>android:layout_alignParentRight</b>	<b>(Remove this attribute)</b>
android:onClick	"showText"
android:text	"Show"

6. Add a Spinner element between the `EditText` element and the `Button` element:

```
<Spinner
    android:id="@+id/label_spinner"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
</Spinner>
```

The Spinner element provides the drop-down list. In the next practical you will add code that will fill the spinner list with values. The layout code for the `EditText`, `Spinner`, and `Button` elements should now look like this: ````xml`

1. Add another `LinearLayout` with a horizontal orientation to enclose two `TextView` elements side-by-side – a text description, and a text field to show the phone number and the phone label – and align the `LinearLayout` to the parent's bottom:

```
```xml
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:layout_alignParentBottom="true">

    <TextView
        ...
    <TextView
        ...
    </LinearLayout>
```

1. Add the following `TextView` elements within the `LinearLayout`:

TextView Attribute	Value
android:id	"@+id/title_phonelabel"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:text	"Phone Number: "

TextView Attribute	Value
android:id	"@+id/text_phonelabel"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:text	"Nothing entered."

2. Check your layout by clicking the Preview tab on the right side of the layout window.

You should now have a screen with the phone entry field at the top on the left, a skeletal spinner next to the field, and the **Show** button on the right. At the bottom should appear the text “Phone Number:” followed by “Nothing entered.”

3. Extract your strings into string resources: Place the cursor on the hard-coded string, press Alt-Enter (Option-Enter on the Mac), and select **Extract string resources**. Then edit the Resource name for the string value. Extract as follows:

Element	String	String resource
EditText	"Enter phone number"	"@string/hint_phonenumber"
Button	"Show"	"@string/show_button"
TextView	"Phone Number: "	"@string/phonenumber_label"
TextView	"Nothing entered."	"@string/nothing_entered"

## 3.2 Add code to activate the spinner and its listener

The choices for this phone label spinner are predetermined, so you can use a simple text array defined in strings.xml to hold the values for it, and use an [ArrayAdapter](#) to assign the array to the spinner.

An adapter connects your data — in this case, the array of spinner items — to the spinner view. The pattern is similar to using an adapter to connect data to the RecyclerView, as shown in a previous lesson.

To activate the spinner and its listener, implement the `AdapterView.OnItemSelectedListener` interface, which requires also adding the `onItemSelected()` and `onNothingSelected()` callback methods.

1. Open `strings.xml` to define the selectable values (**Home**, **Work**, **Mobile**, and **Other**) for the spinner as the string array `labels_array` :

```
<string-array name="labels_array">
    <item>Home</item>
    <item>Work</item>
    <item>Mobile</item>
    <item>Other</item>
</string-array>
```

2. To define the selection callback for the spinner, change your `MainActivity` class to implement the `AdapterView.OnItemSelectedListener` interface as shown:

```
public class MainActivity extends AppCompatActivity implements
    AdapterView.OnItemSelectedListener {
```

As you type **AdapterView**. in the above statement, Android Studio automatically imports the `AdapterView` widget. This line should appear in your block of import statements:

```
`import android.widget.AdapterView;
```

After typing **OnItemSelectedListener** in the above statement, a red light bulb appears in the left margin.

1. Click the bulb and choose **Implement methods**. The `onItemSelected()` and `onNothingSelected()` methods, which are required for `OnItemSelectedListener`, should already be highlighted, and the “Insert @Override” option should be checked. Click **OK**.

This step automatically adds empty `onItemSelected()` and `onNothingSelected()` callback methods to the bottom of the `MainActivity` class.

2. Instantiate a spinner object using the `spinner` element in the layout (`label_spinner`), and set its listener (`spinner.setOnItemSelectedListener`) during the `onCreate()` method — that is, when the main screen is created. Add the code to the `onCreate()` method:

```
// Create the spinner.
Spinner spinner = (Spinner) findViewById(R.id.label_spinner);
if (spinner != null) {
    spinner.setOnItemSelectedListener(this);
}
```

3. Continuing to edit the `onCreate()` method, add a statement that creates the `ArrayAdapter` with the string array (`labels_array`) using the simple spinner layout for

```
each item ( layout.simple_spinner_item ):

    // Create ArrayAdapter using the string array and default spinner layout.
    ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
        R.array.labels_array, android.R.layout.simple_spinner_item);
    // Specify the layout to use when the list of choices appears.
```

4. Specify the layout for the spinner's choices to be `simple_spinner_dropdown_item`, and then apply the adapter to the spinner:

```
adapter.setDropDownViewResource
    (android.R.layout.simple_spinner_dropdown_item);
// Apply the adapter to the spinner.
if (spinner != null) {
    spinner.setAdapter(adapter);

}
```

**Tip:** The `simple_spinner_item` and `simple_spinner_dropdown_item` layouts are the default pre-defined layouts you should use unless you want to define your own layouts for the items in the spinner and the spinner's appearance.

### 3.3 Add code to respond to the user's selections

When the user selects a choice from the drop-down list, the `Spinner` object receives an on-item-selected event. When you implemented the `AdapterView.OnItemSelectedListener` interface in the previous step, you added empty `onItemSelected()` and `onNothingSelected()` callback methods. The former is called when an item is selected, and you will use it to retrieve the selected item in the spinner menu, using `getItemAtPosition`, and assign the item to a string you will call `mSpinnerLabel`. To do this, you must first declare `mSpinnerLabel`.

1. Declare the `mSpinnerLabel` string at the beginning of the `MainActivity` class definition:

```
public class MainActivity extends AppCompatActivity implements AdapterView.OnItemSelectedListener {
    private String mSpinnerLabel = "";
```

2. Add code to the empty `onItemSelected()` callback method, as shown below, to retrieve the user's selected item using `getItemAtPosition`, and assign it to `mSpinnerLabel`:

```
public void onItemSelected(AdapterView<?> adapterView, View view, int
    i, long l) {
    mSpinnerLabel = adapterView.getItemAtPosition(i).toString();
}
```

3. Add code to the empty `onNothingSelected()` callback method, as shown below, to display a logcat message if nothing is selected:

```
public void onNothingSelected(AdapterView<?> adapterView) {  
    Log.d(TAG, "onNothingSelected: ");  
}
```

The TAG in the above statement is in red because it hasn't been defined.

4. Extract the string resource for `"onNothingSelected: "` to `nothing_selected`.
5. Click **TAG**, click the red light bulb, and choose **Create constant field 'TAG'** from the pop-up menu. Android Studio adds the following under the `MainActivity` class declaration:

```
private static final String TAG = ;
```

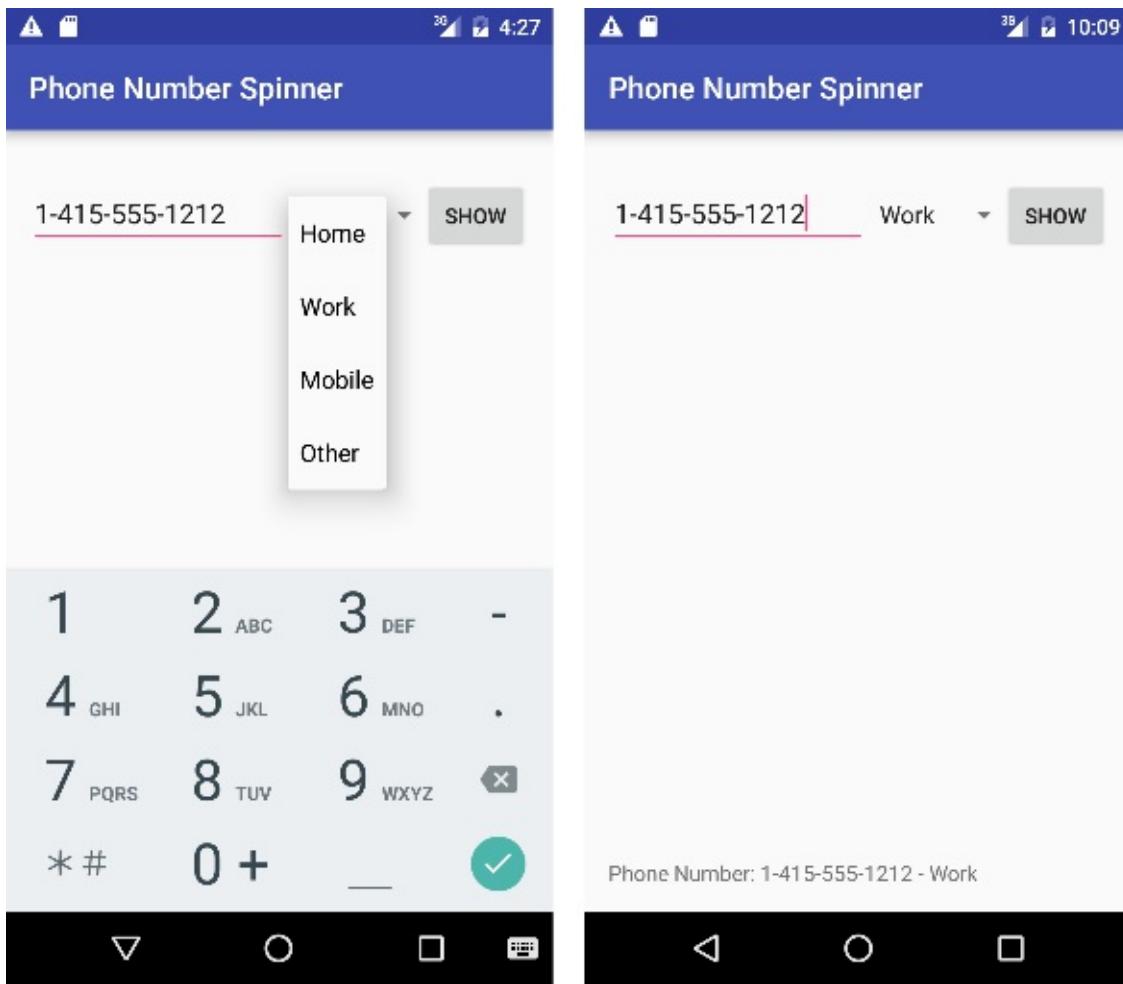
6. Add `MainActivity.class.getSimpleName()` to use the simple name of the class for TAG:

```
private static final String TAG = MainActivity.class.getSimpleName();
```

7. Change the `String showString` statement in the `showText` method to show both the entered string and the selected spinner item (`mSpinnerLabel`):

```
String showString = (editText.getText().toString() + " - " + mSpinnerLabel);
```

8. Run the app.



## Solution Code

PhoneNumberSpinner:

## Task 4. Use a dialog for an alert requiring a decision

You can provide a dialog for an alert to require users to make a decision. A *dialog* is a window that appears on top of the display or fills the display, interrupting the flow of activity.

For example, an alert dialog might require the user to click **Continue** after reading it, or give the user a choice to agree with an action by clicking a positive button (such as **OK** or **Accept**), or to disagree by clicking a negative button (such as **Cancel**). In Android, you use the [AlertDialog](#) subclass of the [Dialog](#) class to show a standard dialog for an alert.

**Tip:** Use dialogs sparingly as they interrupt the user's work flow. Read the [Dialogs design guide](#) for best design practices, and [Dialogs](#) in the Android developer documentation for code examples.

**Tip:** Use dialogs sparingly as they interrupt the user's work flow. Read the [Dialogs design guide](#) for best design practices, and [Dialogs](#) in the Android developer documentation for code examples.

In this practical, you will add a button to trigger a standard alert dialog. In a real world app, you might trigger an alert dialog based on some condition, or based on the user tapping something.

Android Project: AlertSample

## 4.1 Create a new project with a layout to show an alert dialog

In this exercise, you'll build an alert with OK and Cancel buttons, which will be triggered by the user clicking a button.

1. Create a new project called **Alert Sample** based on the Empty Activity template.
2. Open the `activity_main.xml` layout, and make the following changes:

TextView Attribute	Value
<code>android:id</code>	<code>"@+id/top_message"</code>
<code>android:text</code>	"Tap to test the alert."

3. Extract the `android:text` string above into the resource `tap_test` to make it easier to translate.
4. Add a `Button` with the following attributes:

Button Attribute	Value
<code>android:id</code>	<code>"@+button1"</code>
<code>android:layout_width</code>	<code>wrap_content</code>
<code>android:layout_height</code>	<code>wrap_content</code>
<code>android:layout_below</code>	<code>"@+id/top_message"</code>
<code>android:layout_marginTop</code>	<code>"36dp"</code>
<code>android:text</code>	"Alert"
<code>android:onClick</code>	<code>"onClickShowAlert"</code>

5. Extract the `android:text` string above into the resource `alert_button` to make it easier to translate.
6. Extract the dimension value for `android:layout_marginTop` the same way: Place the cursor on `"36dp"`, press Alt-Enter (Option-Enter on the Mac), and select **Extract dimension resource**. Then edit the Resource name for the value to

**Why:** The dimension resource assignments are stored in the `dimens.xml` file (under `app > res > values > dimens`). You can edit this file to change the assignments so that the app can be changed for different display sizes.

## 4.2 Add an alert dialog to the main activity

The *builder* design pattern makes it easy to create an object from a class that has a lot of required and optional attributes and would therefore require a lot of parameters to build. Without this pattern, you would have to create constructors for combinations of required and optional attributes; with this pattern, the code is easier to read and maintain. For more information about the builder design pattern, see [Builder pattern](#).

The builder class is usually a static member class of the class it builds. Thus, you use `AlertDialog.Builder`, which is a subclass of the `AlertDialog` class, to build a standard alert dialog, using `setTitle` to set its title, `setMessage` to set its message, and `setPositiveButton` and `setNegativeButton` to set its buttons.

To make the alert, you need to make an object of `AlertDialog.Builder`, which is a subclass of `AlertDialog`. You will add the `onClickShowAlert()` method, which makes this object as its first order of business.

**Note:** To keep this example simple to understand, the alert dialog is created in the `'onClickShowAlert()'` method. This occurs only if the `'onClickShowAlert()'` method is called, which is what happens when the user clicks the button. This means the app builds a new dialog every time the button is clicked. In a real world app, you may want to build the dialog once in the `'onCreate()'` method, and then invoke the dialog in the `'onClickShowAlert()'` method.

1. Add the `onClickShowAlert()` method to `MainActivity.java` as follows:

```
public void onClickShowAlert(View view) {  
    AlertDialog.Builder alertDialog = new AlertDialog.Builder(MainActivity.this);
```

**Note:** If `AlertDialog.Builder` is not recognized as you enter it, click the red bulb icon, and choose the support library version for importing into your activity.

2. Set the title and the message inside the alert dialog:

```
// Set the dialog title.  
alertDialog.setTitle("Alert");  
// Set the dialog message.  
alertDialog.setMessage("Click OK to continue, or Cancel to stop:");
```

3. Extract the title and message into string resources. The previous lines of code should now be:

3. Extract the title and message into string resources. The previous lines of code should now be:

```
// Set the dialog title.  
 alertDialog.setTitle(R.string.alert_title);  
 // Set the dialog message.  
 alertDialog.setMessage(R.string.alert_message);
```

4. Add the OK button to the alert with `setPositiveButton()` and using `onClickListener()`:

```
// Add the buttons.  
 alertDialog.setPositiveButton("OK", new DialogInterface.OnClickListener() {  
     public void onClick(DialogInterface dialog, int which) {  
         // User clicked OK button.  
         Toast.makeText(getApplicationContext(), "Pressed OK",  
             Toast.LENGTH_SHORT).show();  
     }  
 });
```

You set the positive (**OK**) and negative (**Cancel**) buttons using the `setPositiveButton()` and `setNegativeButton()` methods. After the user taps the **OK** button in the alert, you can grab the user's selection and use it in your code. In this example, you display a toast message if the **OK** button is clicked.

5. Extract the string resource for `"OK"` and for `"Pressed OK"`. The statement should now be:

```
// Add the buttons.  
 alertDialog.setPositiveButton(R.string.ok, new  
     DialogInterface.OnClickListener() {  
         public void onClick(DialogInterface dialog, int which) {  
             // User clicked OK button.  
             Toast.makeText(getApplicationContext(), R.string.pressed_ok,  
                 Toast.LENGTH_SHORT).show();  
         }  
     });
```

6. Add the Cancel button to the alert with `setNegativeButton()` and `onClickListener()`, display a toast message if the button is clicked, and then cancel the dialog:

```
alertDialog.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {  
    public void onClick(DialogInterface dialog, int which) {  
        // User cancelled the dialog.  
        Toast.makeText(getApplicationContext(), "Pressed Cancel",  
            Toast.LENGTH_SHORT).show();  
    }  
});
```

```

        alertDialog.setNegativeButton(R.string.cancel, new
            DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            // User cancelled the dialog.
            Toast.makeText(getApplicationContext(), R.string.pressed_cancel,
                Toast.LENGTH_SHORT).show();
        }
    });
}

```

## 8. Add show to display the alert dialog:

```

// Show the AlertDialog.
alertDialog.show();

```

**Tip:** To learn more about onClickListener and other listeners, see [User Interface: Input Events](#).

The following is the entire `onClickShowAlert()` method:

## Solution code

```

public void onClickShowAlert(View view) {
    // Build the alert dialog.
    AlertDialog.Builder alertDialog = new AlertDialog.Builder(MainActivity.this);
    // Set the dialog title.
    alertDialog.setTitle(R.string.alert_title);
    // Set the dialog message.
    alertDialog.setMessage(R.string.alert_message);
    // Add the buttons.
    alertDialog.setPositiveButton(R.string.ok, new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            // User clicked OK button.
            Toast.makeText(getApplicationContext(), R.string.pressed_ok,
                Toast.LENGTH_SHORT).show();
        }
    });
    alertDialog.setNegativeButton(R.string.cancel, new
        DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int which) {
        // User cancelled the dialog.
        Toast.makeText(getApplicationContext(), R.string.pressed_cancel,
            Toast.LENGTH_SHORT).show();
    }
});
    // Show the AlertDialog.
    alertDialog.show();
}

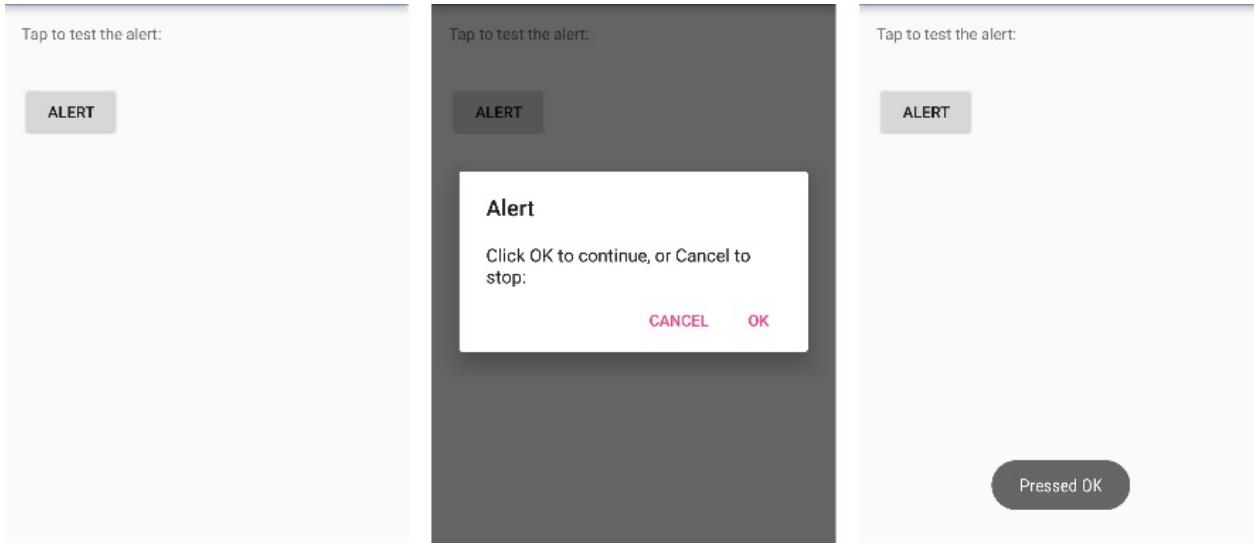
```

```

public void onClickShowAlert(View view) {
    // Build the alert dialog.
    AlertDialog.Builder alertDialog = new AlertDialog.Builder(MainActivity.this);
    // Set the dialog title.
    alertDialog.setTitle(R.string.alert_title);
    // Set the dialog message.
    alertDialog.setMessage(R.string.alert_message);
    // Add the buttons.
    alertDialog.setPositiveButton(R.string.ok, new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            // User clicked OK button.
            Toast.makeText(getApplicationContext(), R.string.pressed_ok,
                    Toast.LENGTH_SHORT).show();
        }
    });
    alertDialog.setNegativeButton(R.string.cancel, new
            DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            // User cancelled the dialog.
            Toast.makeText(getApplicationContext(), R.string.pressed_cancel,
                    Toast.LENGTH_SHORT).show();
        }
    });
    // Show the AlertDialog.
    alertDialog.show();
}

```

Run the app. You should be able to tap the **Alert** button to test the alert dialog. The dialog shows **OK** and **Cancel** buttons, and a toast message appears showing which one you pressed.



## Task 5. Use a picker for user input

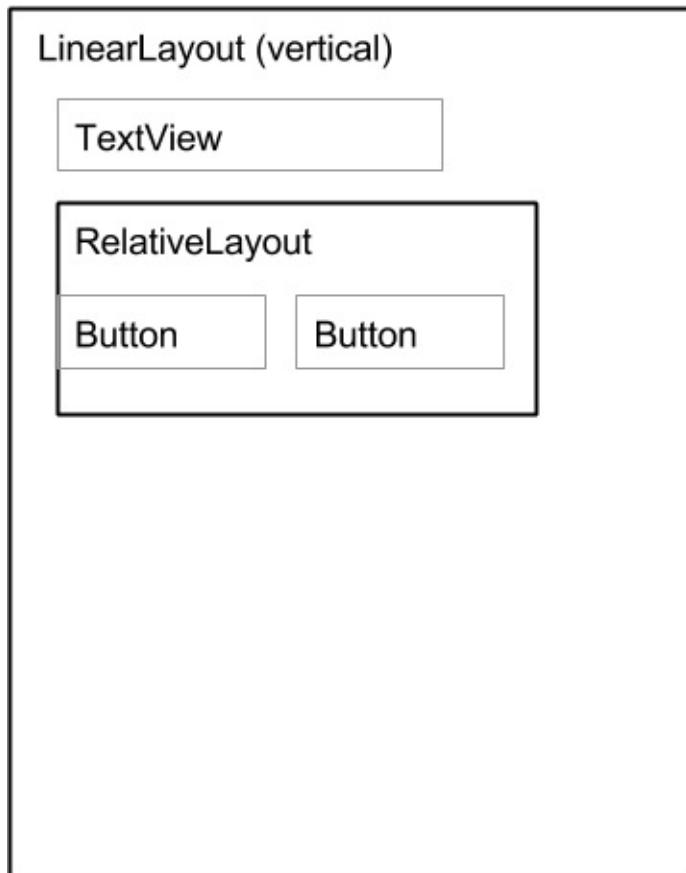
**Tip:** Another benefit of using fragments for the pickers is that you can implement different layout configurations, such as a basic dialog on handset-sized displays or an embedded part of a layout on large displays.

Android Project:

## 5.1 Create the main activity layout

To start this task, create the main activity layout to provide buttons to access the time and date pickers. Refer to the XML layout code below:

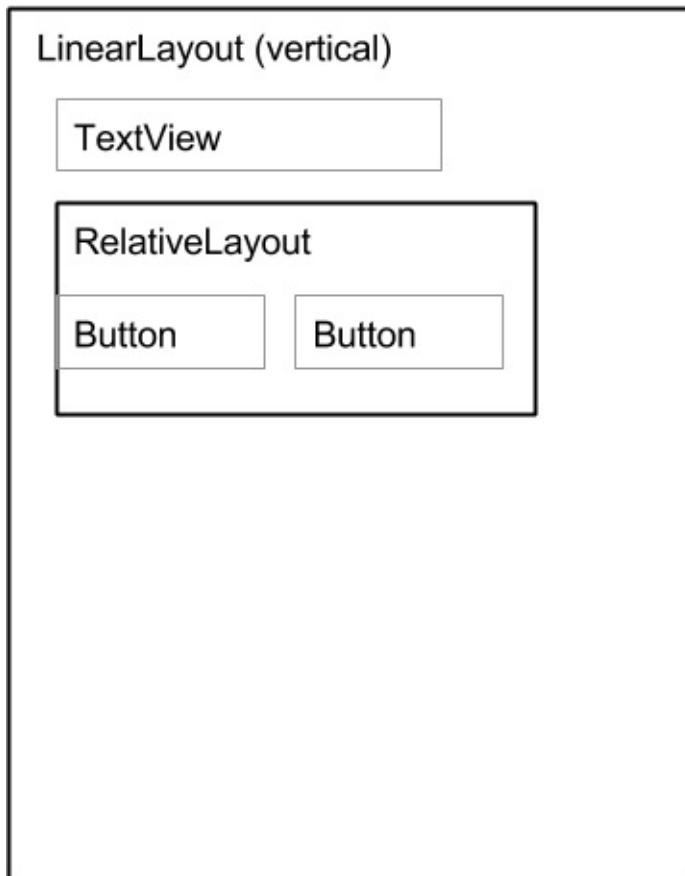
1. Start a new project called **Date Time Pickers** using the Empty Activity template.
2. Open **activity\_main.xml** to edit the layout code.
3. Change the `RelativeLayout` parent to be `LinearLayout` and add `android:orientation="vertical"` to orient the layout vertically. Don't worry about the appearance of the layout yet. The goal is to use a layout that embeds a `RelativeLayout`



within the `LinearLayout`:

4. Change the first `TextView` element's text to "**Choose the date and time:**" and enter a text size of **20sp**, then extract the text to the string resource `choose_datetime`.

TextView Attribute	Old Value	New Value
<code>android:textSize</code>		"20sp"
<code>android:text</code>	"Hello World"	"@string/choose_datetime"



within the LinearLayout:

4. Change the first `TextView` element's text to "**Choose the date and time:**" and enter a text size of **20sp**, then extract the text to the string resource `choose_datetime`.

TextView Attribute	Old Value	New Value
<code>android:textSize</code>		"20sp"
<code>android:text</code>	"Hello World"	"@string/choose_datetime"

5. Extract the `android:textSize` dimension to `text_size`.
6. Add a `RelativeLayout` child inside the `LinearLayout` to contain the `Button` elements, and accept the "match parent" default width and height.
7. Add the first `Button` element within the `RelativeLayout` with the following attributes:

First Button Attribute	Value
<code>android:layout_width</code>	"wrap_content"
<code>android:layout_height</code>	"wrap_content"
<code>android:id</code>	"@+id/button_date"
<code>android:layout_marginTop</code>	"12dp"
<code>android:text</code>	"Date"
<code>android:onClick</code>	"showDatePickerDialog"

Don't worry that the `showDatePickerDialog` reference is in red. The method hasn't been defined yet — you define it later.

```
<?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:orientation="vertical"
    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=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="20sp"
        android:text="@string/choose_datetime" />

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/button_date"
            android:layout_marginTop="12dp"
            android:text="@string/date_button"
            android:onClick="showDatePickerDialog"/>

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/button_time"
            android:layout_marginTop="12dp"
            android:layout_alignBottom="@+id/button_date"
            android:layout_toRightOf="@+id/button_date"
            android:text="@string/time_button"
            android:onClick="showTimePickerDialog"/>

    </RelativeLayout>
</LinearLayout>
```

```
<?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:orientation="vertical"
    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=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="20sp"
        android:text="@string/choose_datetime" />

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/button_date"
            android:layout_marginTop="12dp"
            android:text="@string/date_button"
            android:onClick="showDatePickerDialog"/>

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/button_time"
            android:layout_marginTop="12dp"
            android:layout_alignBottom="@+id/button_date"
            android:layout_toRightOf="@+id/button_date"
            android:text="@string/time_button"
            android:onClick="showTimePickerDialog"/>

    </RelativeLayout>
</LinearLayout>
```

After adding the empty `onDataSet()` method, Android Studio automatically adds the following in the `import` block at the top:

```
import android.widget.DatePicker;
```

1. Replace `onCreateView()` with `onCreateDialog()`, and remove the empty public constructor for `DatePickerFragment`. Annotate the `onCreateDialog()` method with `@NonNull`, and add the following code to `onCreateDialog()` to initialize the `year`, `month`, and `day` from `Calendar`, and return the dialog and these values to the main activity.

```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    // Use the current date as the default date in the picker.
    final Calendar c = Calendar.getInstance();
    int year = c.get(Calendar.YEAR);
    int month = c.get(Calendar.MONTH);
    int day = c.get(Calendar.DAY_OF_MONTH);

    // Create a new instance of DatePickerDialog and return it.
    return new DatePickerDialog(getActivity(), this, year, month, day);
}
```

2. To make the code more readable, change the `onDataSet()` method's parameters from `int i`, `int i1`, and `int i2` to `int year`, `int month`, and `int day`:

```
public void onDataSet(DatePicker view, int year, int month, int day)
```

## Solution code for DatePickerFragment

```
```java
public class DatePickerFragment extends DialogFragment
```

```
    implements DatePickerDialog.OnDateSetListener {
```

```
        @Override
        public Dialog onCreateDialog(Bundle savedInstanceState) {
```

After adding the empty `onDataSet()` method, Android Studio automatically adds the following in the `import` block at the top:

```
import android.widget.DatePicker;
```

1. Replace `onCreateView()` with `onCreateDialog()`, and remove the empty public constructor for `DatePickerFragment`. Annotate the `onCreateDialog()` method with `@NonNull`, and add the following code to `onCreateDialog()` to initialize the `year`, `month`, and `day` from `Calendar`, and return the dialog and these values to the main activity.

```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    // Use the current date as the default date in the picker.
    final Calendar c = Calendar.getInstance();
    int year = c.get(Calendar.YEAR);
    int month = c.get(Calendar.MONTH);
    int day = c.get(Calendar.DAY_OF_MONTH);

    // Create a new instance of DatePickerDialog and return it.
    return new DatePickerDialog(getActivity(), this, year, month, day);
}
```

2. To make the code more readable, change the `onDataSet()` method's parameters from `int i`, `int i1`, and `int i2` to `int year`, `int month`, and `int day`:

```
public void onDataSet(DatePicker view, int year, int month, int day)
```

## Solution code for DatePickerFragment

```
```java
public class DatePickerFragment extends DialogFragment
```

```
    implements DatePickerDialog.OnDateSetListener {
```

```
        @Override
        public Dialog onCreateDialog(Bundle savedInstanceState) {
```

```
<a id="task5steps3" />

### **5.3 Create a new fragment for the time picker**

Add a fragment to the DateTimePickers project for the time picker:
1. Select **MainActivity** again.
1. Choose **File > New > Fragment > Fragment (Blank)**, and name the fragment **TimeP
ickerFragment**. Uncheck all three options so you do *not* create a layout XML, do *no
t* include fragment factory methods, and do *not* include interface callbacks. Click *F
inish* to create the fragment.
1. Open **TimePickerFragment** and follow the same procedures as with `DatePickerFrag
ment`, implementing the `onTimeSet()` blank method, replacing `onCreateView()` with `o
nCreateDialog()`, and removing the empty public constructor for `TimePickerFragment`.  
It performs the same tasks as the `DatePickerFragment`, but with time values:
* It extends `DialogFragment` and implements `TimePickerDialog.OnTimeSetListener
` to create a standard time picker with a listener. See [Picker](http://developer.andr
oid.com/guide/topics/ui/controls/pickers.html) for more information about extending Di
alogFragment for a time picker.
* It uses the `onCreateDialog()` method to initialize the `hour` and `minute` fr
om `Calendar`, and returns the dialog and these values to the main activity using the
24-hour date format.
* It also defines the empty `onTimeSet()` method for you to add code to use the
`hourOfDay` and `minute` the user selects.
<div class="note">
<strong>Note:</strong>As you make the changes, Android Studio automatically adds the f
ollowing in the `import` block at the top:
```java
import android.app.TimePickerDialog.OnTimeSetListener;
import android.support.v4.app.DialogFragment;
import android.app.TimePickerDialog;
import android.widget.TimePicker;
import java.util.Calendar;
```

</div>

## Solution code for TimePickerFragment

```
<a id="task5steps3" />

### **5.3 Create a new fragment for the time picker**

Add a fragment to the DateTimePickers project for the time picker:
1. Select **MainActivity** again.
1. Choose **File > New > Fragment > Fragment (Blank)**, and name the fragment **TimePickerFragment**. Uncheck all three options so you do *not* create a layout XML, do *not* include fragment factory methods, and do *not* include interface callbacks. Click *Finish* to create the fragment.
1. Open **TimePickerFragment** and follow the same procedures as with `DatePickerFragment`, implementing the `onTimeSet()` blank method, replacing `onCreateView()` with `onCreateDialog()`, and removing the empty public constructor for `TimePickerFragment`. It performs the same tasks as the `DatePickerFragment`, but with time values:


- It extends `DialogFragment` and implements `TimePickerDialog.OnTimeSetListener` to create a standard time picker with a listener. See [Picker](http://developer.android.com/guide/topics/ui/controls/pickers.html) for more information about extending DialogFragment for a time picker.
- It uses the `onCreateDialog()` method to initialize the `hour` and `minute` from `Calendar`, and returns the dialog and these values to the main activity using the 24-hour date format.
- It also defines the empty `onTimeSet()` method for you to add code to use the `hourOfDay` and `minute` the user selects.


<div class="note">
<strong>Note:</strong>As you make the changes, Android Studio automatically adds the following in the `import` block at the top:
```java
import android.app.TimePickerDialog.OnTimeSetListener;
import android.support.v4.app.DialogFragment;
import android.app.TimePickerDialog;
import android.widget.TimePicker;
import java.util.Calendar;

```

&lt;/div&gt;

## Solution code for TimePickerFragment

```

public class MainActivity extends AppCompatActivity {

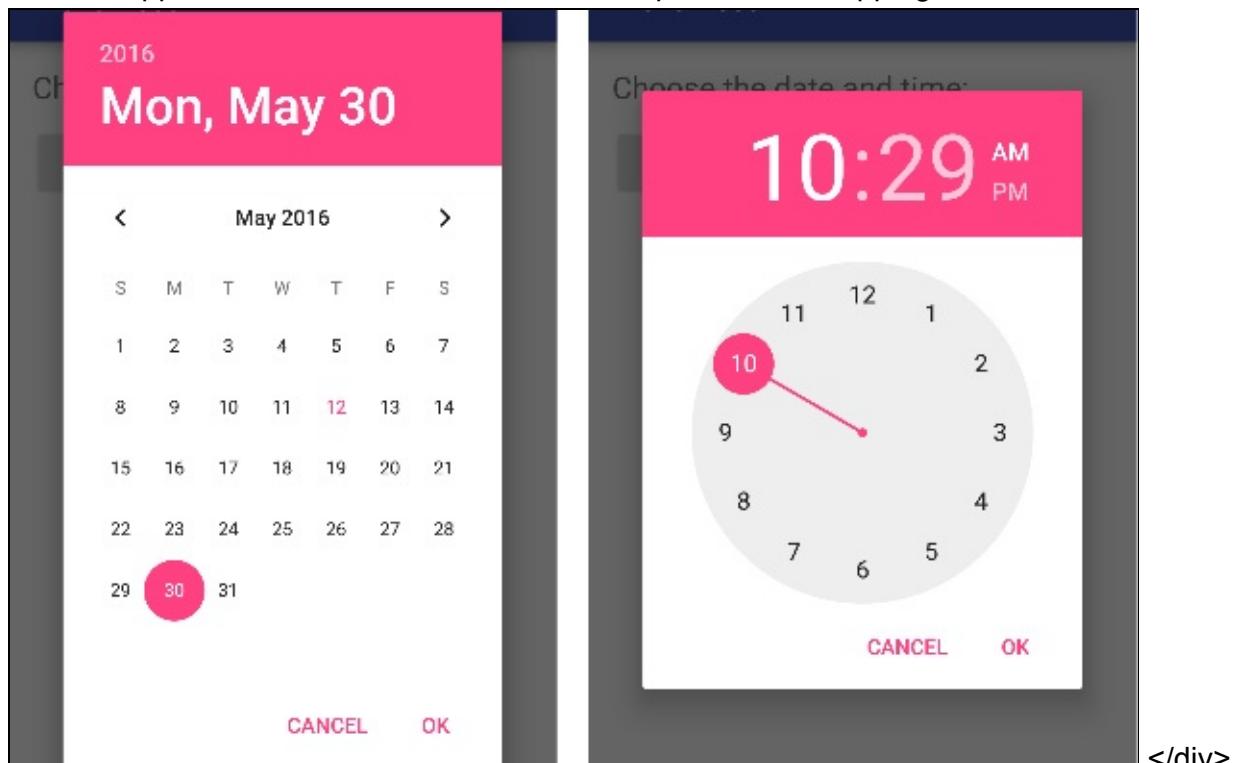
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void showDatePickerDialog(View v) {
        DialogFragment newFragment = new DatePickerFragment();
        newFragment.show(getSupportFragmentManager(),
            getString(R.string.date_picker));
    }

    public void showTimePickerDialog(View view) {
        DialogFragment newFragment = new TimePickerFragment();
        newFragment.show(getSupportFragmentManager(),
            getString(R.string.time_picker));
    }
}

```

Run the app. You should see the date and time pickers after tapping the buttons.



## 5.5 Use the chosen date and time

In this exercise you'll convert the user-chosen date to a string, and pass the string back to the main activity. You'll do the same thing for the user-selected time. You then use the `onFinishDateDialog()` and `onFinishTimeDialog()` methods in the main activity to take the

```

public class MainActivity extends AppCompatActivity {

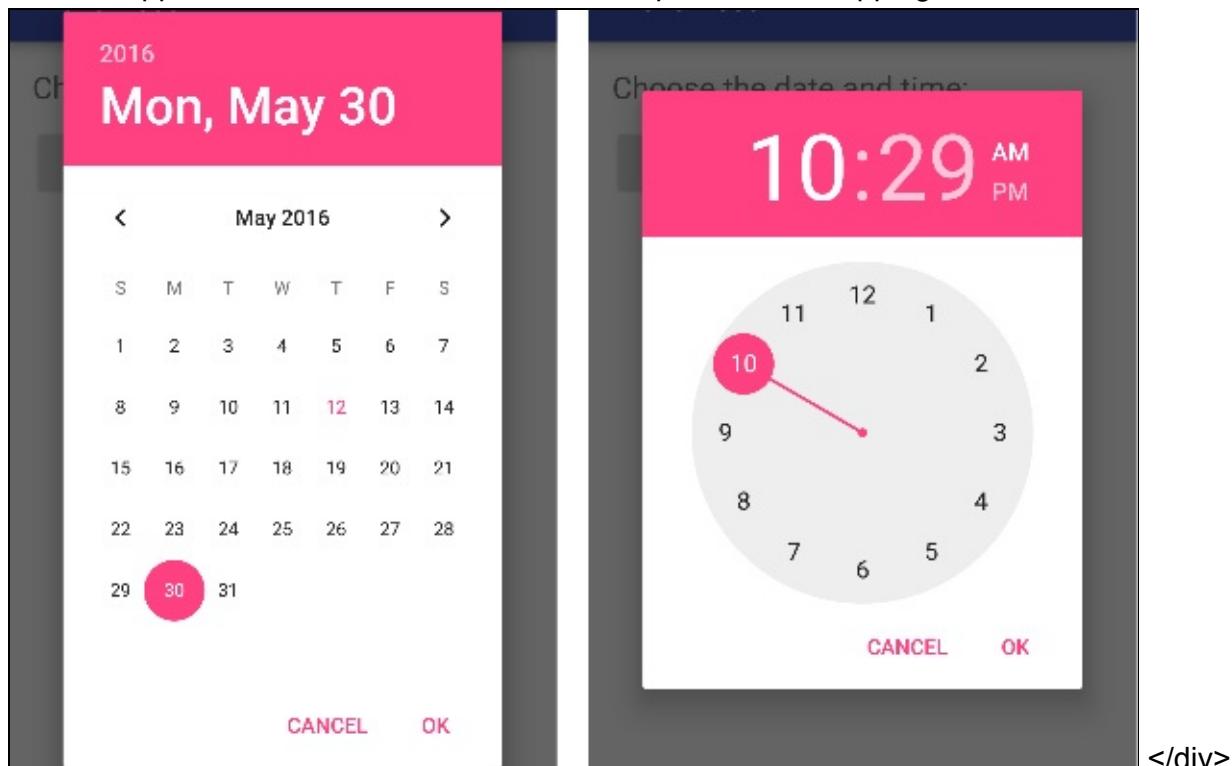
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void showDatePickerDialog(View v) {
        DialogFragment newFragment = new DatePickerFragment();
        newFragment.show(getSupportFragmentManager(),
            getString(R.string.date_picker));
    }

    public void showTimePickerDialog(View view) {
        DialogFragment newFragment = new TimePickerFragment();
        newFragment.show(getSupportFragmentManager(),
            getString(R.string.time_picker));
    }
}

```

Run the app. You should see the date and time pickers after tapping the buttons.



&lt;/div&gt;

## 5.5 Use the chosen date and time

In this exercise you'll convert the user-chosen date to a string, and pass the string back to the main activity. You'll do the same thing for the user-selected time. You then use the `onFinishDateDialog()` and `onFinishTimeDialog()` methods in the main activity to take the

```
public void onTimeSet(TimePicker view, int hourOfDay, int minute) {  
  
    // Do something with the time chosen by the user  
  
}
```

### New code:</code>

```
public void onTimeSet(TimePicker view, int hourOfDay, int minute) { **  
  
    // Convert the time elements to strings.  
    String hour_string = Integer.toString(hourOfDay);  
    String minute_string = Integer.toString(minute);  
    String timeMessage = (hour_string + ":" + minute_string);  
    MainActivity activity = (MainActivity) getActivity();  
    activity.onFinishTimeDialog(timeMessage);  
}
```

**Note:**The `onFinishTimeDialog()` method is in red because it has not yet been created in `MainActivity`.

1. Create string resources in strings.xml:

```
<string name="date">Date: "</string>  
<string name="time">Time: "</string>
```

Be sure to include the space and colon in the two string values.

2. Add the `onFinishDateDialog()` and `onFinishTimeDialog()` methods in the main activity to take the date or time message and show it in a toast message:
- ```
java public void  
onFinishDateDialog(String message) {
```

```
    Toast.makeText(this, getString(R.string.date) + message,  
        Toast.LENGTH_SHORT).show();
```

```
}
```

```
public void onFinishTimeDialog(String message) { Toast.makeText(this,  
    getString(R.string.time) + message, Toast.LENGTH_SHORT).show(); } ``
```

1. You can now run the app. After selecting the date or time, the date or time appears in a toast message.

```
public void onTimeSet(TimePicker view, int hourOfDay, int minute) {  
  
    // Do something with the time chosen by the user  
  
}
```

### New code:</code>

```
public void onTimeSet(TimePicker view, int hourOfDay, int minute) { **  
  
    // Convert the time elements to strings.  
    String hour_string = Integer.toString(hourOfDay);  
    String minute_string = Integer.toString(minute);  
    String timeMessage = (hour_string + ":" + minute_string);  
    MainActivity activity = (MainActivity) getActivity();  
    activity.onFinishTimeDialog(timeMessage);  
}
```

**Note:**The `onFinishTimeDialog()` method is in red because it has not yet been created in `MainActivity`.

1. Create string resources in strings.xml:

```
<string name="date">"Date: "</string>  
<string name="time">"Time: "</string>
```

Be sure to include the space and colon in the two string values.

2. Add the `onFinishDateDialog()` and `onFinishTimeDialog()` methods in the main activity to take the date or time message and show it in a toast message:
- ```
java public void  
onFinishDateDialog(String message) {
```

```
    Toast.makeText(this, getString(R.string.date) + message,  
        Toast.LENGTH_SHORT).show();
```

```
}
```

```
public void onFinishTimeDialog(String message) { Toast.makeText(this,  
    getString(R.string.time) + message, Toast.LENGTH_SHORT).show(); } ``
```

1. You can now run the app. After selecting the date or time, the date or time appears in a toast message.

- Setting up XML layout attributes to control the keyboard for an `EditText` element:
  - Using the `textAutoCorrect` value for the `android:inputType` attribute to change the keyboard so that it suggests spelling corrections.
  - Using the `textCapSentences` value for the `android:inputType` attribute to start each new sentence with a capital letter.
  - Using the `textPassword` value for the `android:inputType` attribute to hide a password when entering it.
  - Using the `textEmailAddress` value for the `android:inputType` attribute to show an email keyboard rather than a standard keyboard.
  - Using the `phone` value for the `android:inputType` attribute to show a phone keypad rather than a standard keyboard.
  - Challenge: Using the `android:imeOptions` attribute with the `actionSend` value to perform an action directly from the keyboard and replace the Return key with an action key, such as an implicit intent to another app to dial a phone number.
- Using a `Spinner` input control to provide a drop-down menu, and writing code to control it:
  - Using an  `ArrayAdapter` to assign an array of text values as the spinner menu items.
  - Implementing the `AdapterView.OnItemSelectedListener` interface, which requires also adding the `onItemSelected()` and `onNothingSelected()` callback methods to activate the spinner and its listener.
  - Using the `onItemSelected()` callback method to retrieve the selected item in the spinner menu using `getItemAtPosition`.
- Using `AlertDialog.Builder`, a subclass of `AlertDialog`, to build a standard alert dialog, using `setTitle` to set its title, `setMessage` to set its message, and `setPositiveButton` and `setNegativeButton` to set its buttons.
- Using the standard date and time pickers:
  - Adding a fragment for a date picker, and extending the `DialogFragment` class to implement `DatePickerDialog.OnDateSetListener` for a standard date picker with a listener.
  - Adding a fragment for a time picker, and extending the `DialogFragment` class to implement `TimePickerDialog.OnTimeSetListener` for a standard time picker with a listener.
  - Implementing the `onDateSet()`, `onTimeSet()`, and `onCreateDialog()` methods.
  - Using the `onFinishDateDialog()` and `onFinishTimeDialog()` methods to retrieve the selected date and time.

## Resources

- Setting up XML layout attributes to control the keyboard for an `EditText` element:
  - Using the `textAutoCorrect` value for the `android:inputType` attribute to change the keyboard so that it suggests spelling corrections.
  - Using the `textCapSentences` value for the `android:inputType` attribute to start each new sentence with a capital letter.
  - Using the `textPassword` value for the `android:inputType` attribute to hide a password when entering it.
  - Using the `textEmailAddress` value for the `android:inputType` attribute to show an email keyboard rather than a standard keyboard.
  - Using the `phone` value for the `android:inputType` attribute to show a phone keypad rather than a standard keyboard.
  - Challenge: Using the `android:imeOptions` attribute with the `actionSend` value to perform an action directly from the keyboard and replace the Return key with an action key, such as an implicit intent to another app to dial a phone number.
- Using a `Spinner` input control to provide a drop-down menu, and writing code to control it:
  - Using an  `ArrayAdapter` to assign an array of text values as the spinner menu items.
  - Implementing the `AdapterView.OnItemSelectedListener` interface, which requires also adding the `onItemSelected()` and `onNothingSelected()` callback methods to activate the spinner and its listener.
  - Using the `onItemSelected()` callback method to retrieve the selected item in the spinner menu using `getItemAtPosition`.
- Using `AlertDialog.Builder`, a subclass of `AlertDialog`, to build a standard alert dialog, using `setTitle` to set its title, `setMessage` to set its message, and `setPositiveButton` and `setNegativeButton` to set its buttons.
- Using the standard date and time pickers:
  - Adding a fragment for a date picker, and extending the `DialogFragment` class to implement `DatePickerDialog.OnDateSetListener` for a standard date picker with a listener.
  - Adding a fragment for a time picker, and extending the `DialogFragment` class to implement `TimePickerDialog.OnTimeSetListener` for a standard time picker with a listener.
  - Implementing the `onDateSet()`, `onTimeSet()`, and `onCreateDialog()` methods.
  - Using the `onFinishDateDialog()` and `onFinishTimeDialog()` methods to retrieve the selected date and time.

## Resources



# 4.2 P: Using an Options Menu and Radio Buttons

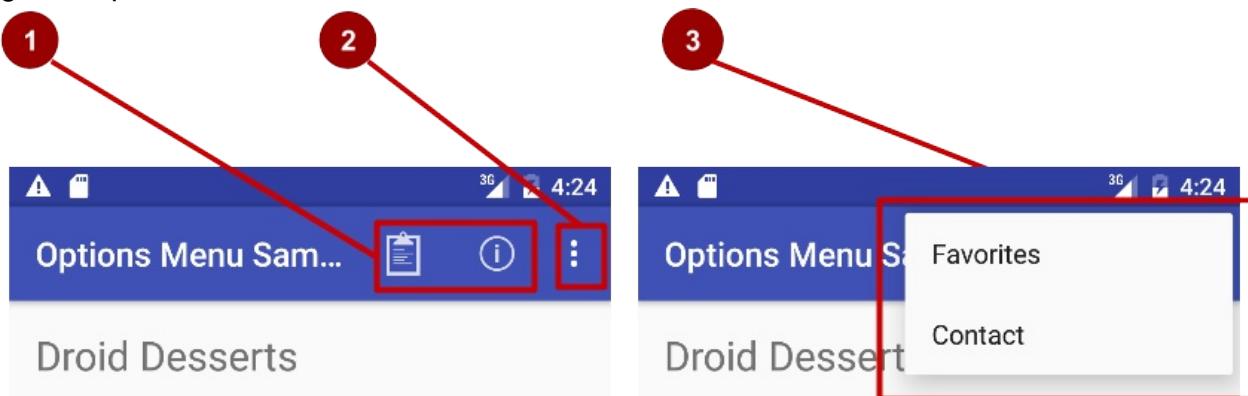
## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App Overview
- Task 1: Add items to the options menu
- Task 2: Add icons for menu items and images for the layout
- Coding challenge #1 (optional)
- Task 3: Handle the selected menu item
- Task 4: Add radio buttons, images buttons, and up navigation
- Coding challenge #2
- Summary
- Resources

In this practical you'll learn about setting up the options menu in your app, and adding radio buttons for the user to select one item from a set of items.

The options menu is the primary collection of menu items for an activity. You can use the options menu for navigation to other activities, such as placing an order, or for actions that have a global impact on the app, such as changing settings.

Options menu items appear in the action overflow popup menu (see figure below). However, you can place some items as icons — as many as can fit — in the top bar, which is called the *app bar*. It's a dedicated space at the top of each screen that is generally persistent throughout the app's screens. Using the app bar makes your app consistent with other Android apps, allowing users to quickly understand how to operate your app and have a great experience.



In the above figure:

1. First two options menu items appearing as icons in the app bar.
2. The action overflow button to show more options menu items.
3. More options menu items appearing in the overflow pop-up.

**Tip:** To provide a familiar and consistent user experience, you should use the Menu APIs to present user actions and other options in your activities. See [Menus](#) for details.

## What you should already KNOW

From the previous chapters, you should be familiar with how to do the following:

- Creating and running apps in Android Studio.
- Creating and editing UI elements using the Layout Editor, entering XML code directly, and accessing elements from your Java code.
- Adding onClick functionality to a button.

## What you will LEARN

- Adding menu items to the options menu.
- Adding icons for items in the options menu.
- Setting menu items to show in the action bar.
- Adding the event handlers for menu item clicks.
- Adding a set of radio buttons with handlers.

## What you will DO

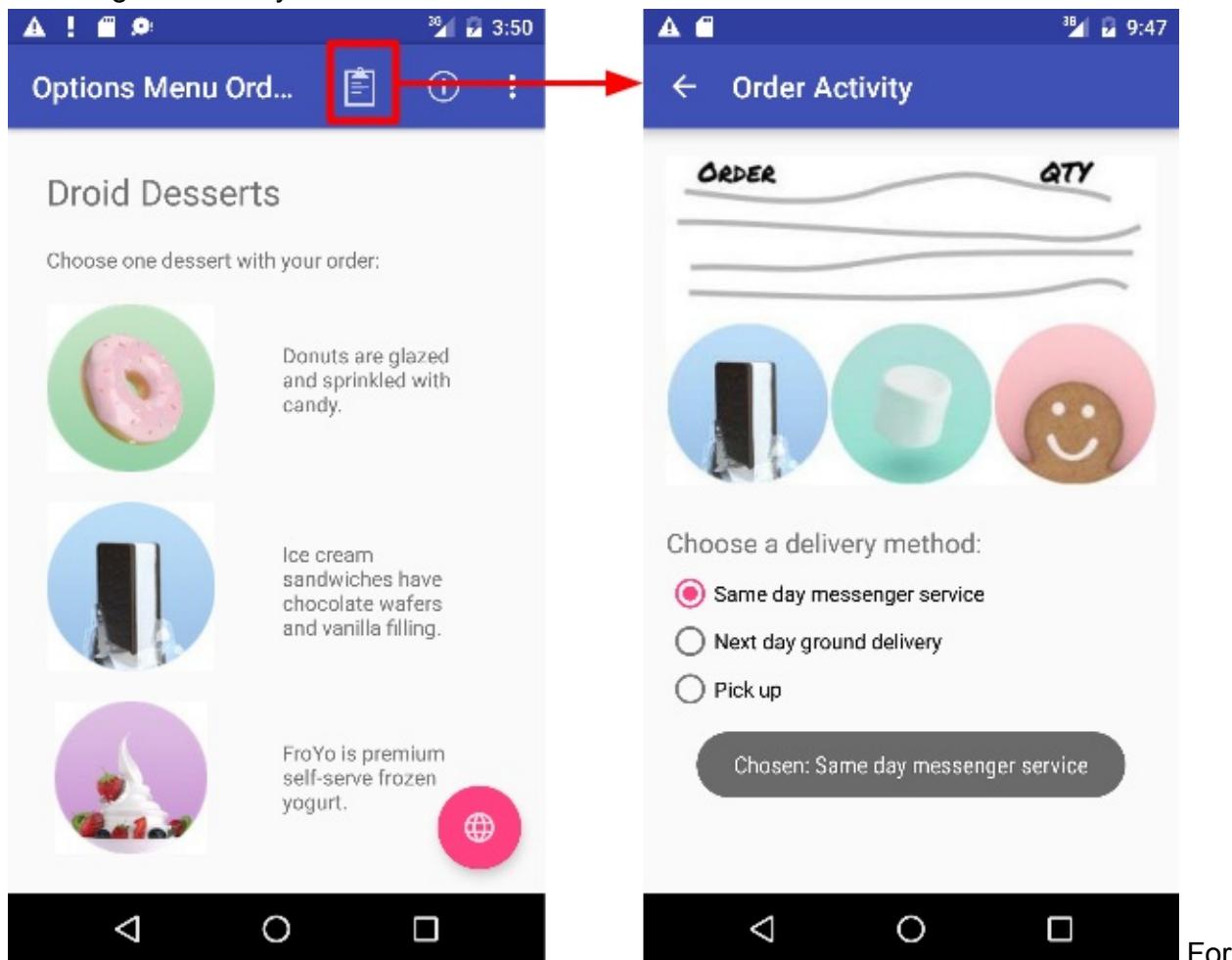
- Start a new project based with the Basic Activity template that offers an options menu.
- Add menu items to the options menu.
- Add icons for menu items to appear in the action bar.
- Connect menu item clicks to event handlers that process the click events.
- Add a second activity with radio buttons to make a choice.

## App Overview

You will build an app called Options Menu Sample that has an app bar at the top of the display containing an action overflow menu. You will learn how to modify the options menu, add icons for some of the items, and write code that determines whether to show the icon for

the menu item in the app bar at the top of the screen, or show the item in the action overflow menu — depending on the screen size.

Continuing this exercise and the “menu” theme, you will create a prototype of an app for placing orders for food items. You will use *placeholder* images, also known as “mock-ups”, that stand in for functions you haven’t developed yet, such as listing food items with “Add to order” buttons, so that the app appears to be a prototype of a full app. You will then connect a new activity to the Order menu choice, and create radio buttons on the Order screen for choosing the delivery method.



For this exercise you are using the default format for the app bar. To read more about design considerations for using the app bar, see “[App Bar](#)” in the Material Design Spec.

Android project: OptionsMenuSample

## Task 1. Add items to the options menu

After starting a new project based on the Basic Activity template, you’ll modify the “Hello World” TextView with appropriate text for this example, and add menu items to the options menu in the app bar at the top of the screen.

## 1.1 Start the new project and examine the app bar code

1. Start a new Android Studio project with the app name **Options Menu Sample**. Choose the **Basic Activity** template, accept the default settings for the main activity, and click **Finish**.

The project opens with two layouts in the **res > layout** folder: **activity\_main.xml**, and **content\_main.xml**.

2. Open **content\_main.xml** to see what it does. It defines a `RelativeLayout` with the layout behavior set to `@string/appbar_scrolling_view_behavior`, which controls the scrolling behavior of the app bar's options menu. This string, defined in the **values.xml** file (which is generated by Android Studio and should not be edited), is

```
"android.support.design.widget.AppBarLayout$ScrollingViewBehavior" .
```

For more about scrolling behavior, see the [Android Design Support Library blog entry](#) in the Android Developers Blog. For design practices involving scrolling menus, see [Scrolling Techniques](#) in the Material Design spec.

3. In **content\_main.xml**, extract the `"Hello World"` string in the `TextView` to use the `intro_text` resource name, and then open **strings.xml** and redefine the resource to use some descriptive text, such as **Droid Desserts** for a dessert-ordering app:

```
<string id="intro_text">Droid Desserts</string>
```

4. Open **activity\_main.xml** to see the main layout, which uses a `CoordinatorLayout` viewgroup with the `AppBarLayout` class.

`AppBarLayout` is a vertical `LinearLayout` which uses a `Toolbar` widget to implement an app bar — a section at the top of the display that may display the activity title, application-level navigation, and other interactive items, and includes scrolling gestures.

The `Toolbar` widget within this layout has the id `toolbar`:

```
<android.support.design.widget.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/AppTheme.AppBarOverlay">

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay" />

</android.support.design.widget.AppBarLayout>
```

For more details about the `AppBarLayout` class, see [AppBarLayout](#) in the Android Developer Reference. For more details about toolbars, see [Toolbar](#) in the Android Developer Reference.

The `activity_main.xml` layout also uses an `include layout` statement to include the entire layout defined in `content_main.xml`. This separation of layout definitions makes it easier to change the layout's *content* (for now, just the `TextView` field), apart from the layout's toolbar definition and coordinator layout. This is a best practice for separating your content (which may need to be translated) from the format of your layout.

5. Run the app. Notice the bar at the top of the screen showing the name of the app. It also shows the *action overflow* button (three vertical dots) on the right side. Tap the overflow button to see the menu, with **Settings** as the menu option.
6. Examine the `AndroidManifest.xml` file. The `.MainActivity` activity is set to use the `NoActionBar` theme to prevent the app from using the native `ActionBar` class attributes for the app bar:

```
android:theme="@style/AppTheme.NoActionBar"
```

The `NoActionBar` theme is defined in the `styles.xml` file (expand `app > res > values > styles.xml` to see it). Styles are covered in another lesson, but you can see that the `NoActionBar` theme sets the `windowActionBar` attribute to `false` (no window action bar), and the `windowNoTitle` attribute to `true` (no title).

Why? The reason these values are set is because you are defining the app bar in your layout with `AppBarLayout`, which would conflict with the default settings. You need to essentially disable the `ActionBar`'s window if you want to replace it with a Toolbar.

1. Look at the `MainActivity.java` file, which extends `AppCompatActivity` and starts with the `onCreate()` method.

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);  
    setSupportActionBar(toolbar);  
    ...
```

The activity's `onCreate()` method calls the activity's `setSupportActionBar()` method, and passes `toolbar` to it, setting the `toolbar` defined in `activity_main.xml` as the app bar for the activity.

For best practices about adding an app bar to your app, see “[Adding the App Bar](#)” in Best Practices for User Interface.

## 1.2 Add more menu items to the options menu

Android provides a standard XML format to define menu items. Instead of building a menu in your activity's code, you can define a menu and all its items in an XML menu resource. You can then inflate the menu resource (load it as a `Menu` object) in your activity or fragment.

In keeping with the “menu” idea, you will design this app as a restaurant app to show options for ordering food, checking the status of your order, showing favorite foods, and contacting the restaurant. You will change the options menu to have four options: Contact (replacing Settings), Order, Status, and Favorites.

1. Take a look at `menu_main.xml` (expand `res > menu` in the Project view). It defines menu items with the `<item> </item>` tag within the `<menu> </menu>` block. The only menu item provided from the template is Settings, which is defined as:

[CODE\_HIGHLIGHT = "XML"] `<item`

```
    android:id="@+id/action_settings"
    android:orderInCategory="100"
    android:title="@string/action_settings"
    app:showAsAction="never" />
```

1. Open `strings.xml` to see that the resource name `action_settings` has string value of `Settings` : `<string name="action_settings">Settings</string>`
2. Go back to `menu_main.xml`, and change the following attributes of the Settings item to make it the Contact item:

Attribute	Value
<code>android:id</code>	<code>"@+id/action_contact"</code>
<code>android:title</code>	<code>"Contact"</code>
<code>app:showAsAction</code>	<code>"never"</code>

3. Extract the hard-coded string `"Contact"` into the string resource `action_contact`.
4. Add a new Order menu item using the `<item> </item>` tag within the `<menu> </menu>` block, and give the item the following attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/action_order"</code>
<code>android:orderInCategory</code>	<code>"10"</code>
<code>android:title</code>	<code>"Order"</code>
<code>app:showAsAction</code>	<code>"never"</code>

The `android:orderInCategory` attribute specifies the order in which the menu items

appear in the menu, with the lowest number appearing higher in the menu. The Contact item is set to 100, which is a big number in order to specify that it shows up at the bottom rather than the top. You set the Order item to 10, which puts it above Contact, and leaves plenty of room in the menu for more items.

5. Extract the hard-coded string "order" into the string resource `action_order`.
6. Add two more menu items the same way with the following attributes:

Status Item Attribute	Value
<code>android:id</code>	"@+id/action_status"
<code>android:orderInCategory</code>	"20"
<code>android:title</code>	"Status"
<code>app:showAsAction</code>	"never"

Favorites Item Attribute	Value
<code>android:id</code>	"@+id/action_favorites"
<code>android:orderInCategory</code>	"40"
<code>android:title</code>	"Favorites"
<code>app:showAsAction</code>	"never"

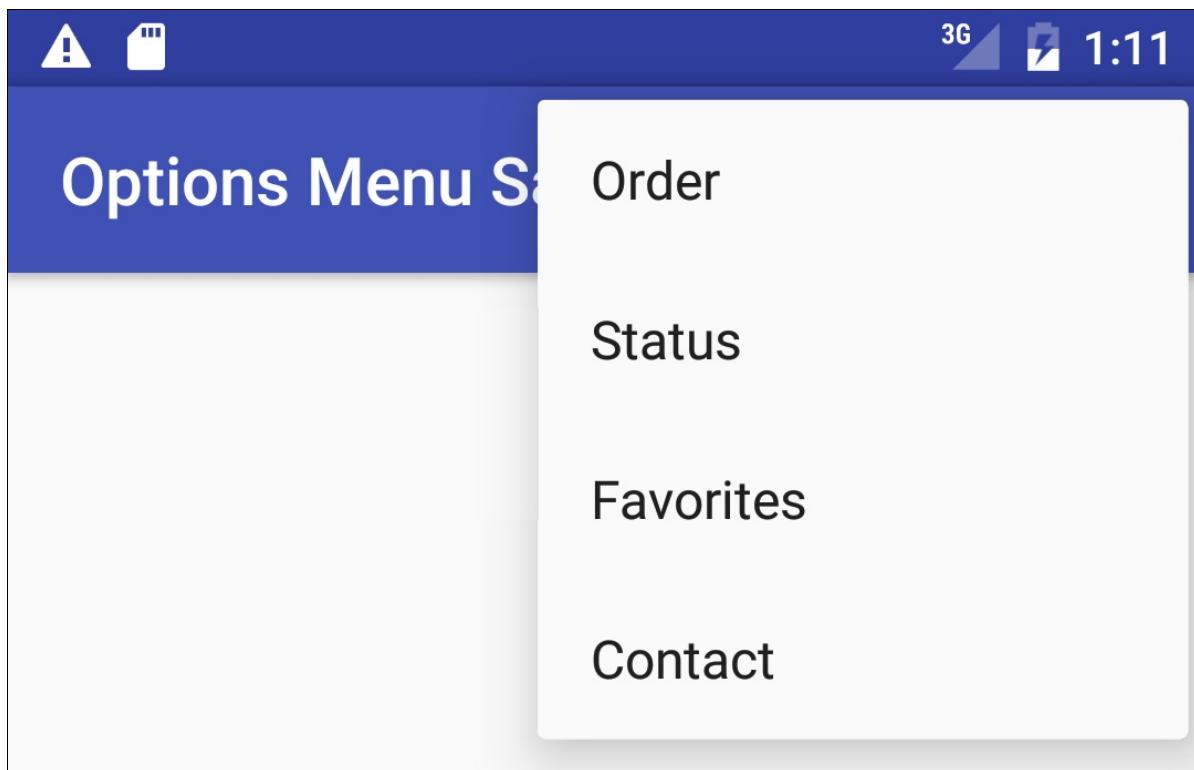
7. Extract "Status" into the resource `action_status`, and "Favorites" into the resource `action_favorites`.
8. You will display a toast message with an action message depending on which option menu item the user selects. Add the following string names and values `in strings.xml` for these messages:

```
<string name="action_order_message">You selected Order.</string>
<string name="action_status_message">You selected Status.</string>
<string name="action_favorites_message">You selected Favorites.</string>
<string name="action_contact_message">You selected Contact.</string>
```

9. Open `MainActivity.java`, and change the `if` statement in the `onOptionsItemSelected()` method replacing the id `action_settings` with the new id `action_order`:

```
if (id == R.id.action_order)
```

Run the app, and tap the action overflow icon to see the options menu. You will soon add callbacks to respond to items selected from this menu.



Notice the order of items in the options menu. You used the `android:orderInCategory` attribute to specify the priority of the menu items in the menu: The Order item is 10, followed by Status (20) and Favorites (40), and Contact is last (100). The following table shows the priority of items in the menu:

Menu Item	orderInCategory attribute
Order	10
Status	20
Favorites	40
Contact	100

## Task 2. Add icons for menu items and images for the layout

Whenever possible, you want to show the most frequently used actions using icons in the app bar so the user can click them without having to first click the overflow icon. In this task, you'll add icons for some of the menu items, and show some of menu items in the app bar at the top of the screen as icons.

In this example, the Order and Status actions are considered the most frequently used. Favorites is occasionally used, and Contact is the least frequently used. You can set icons for these actions, and specify the following:

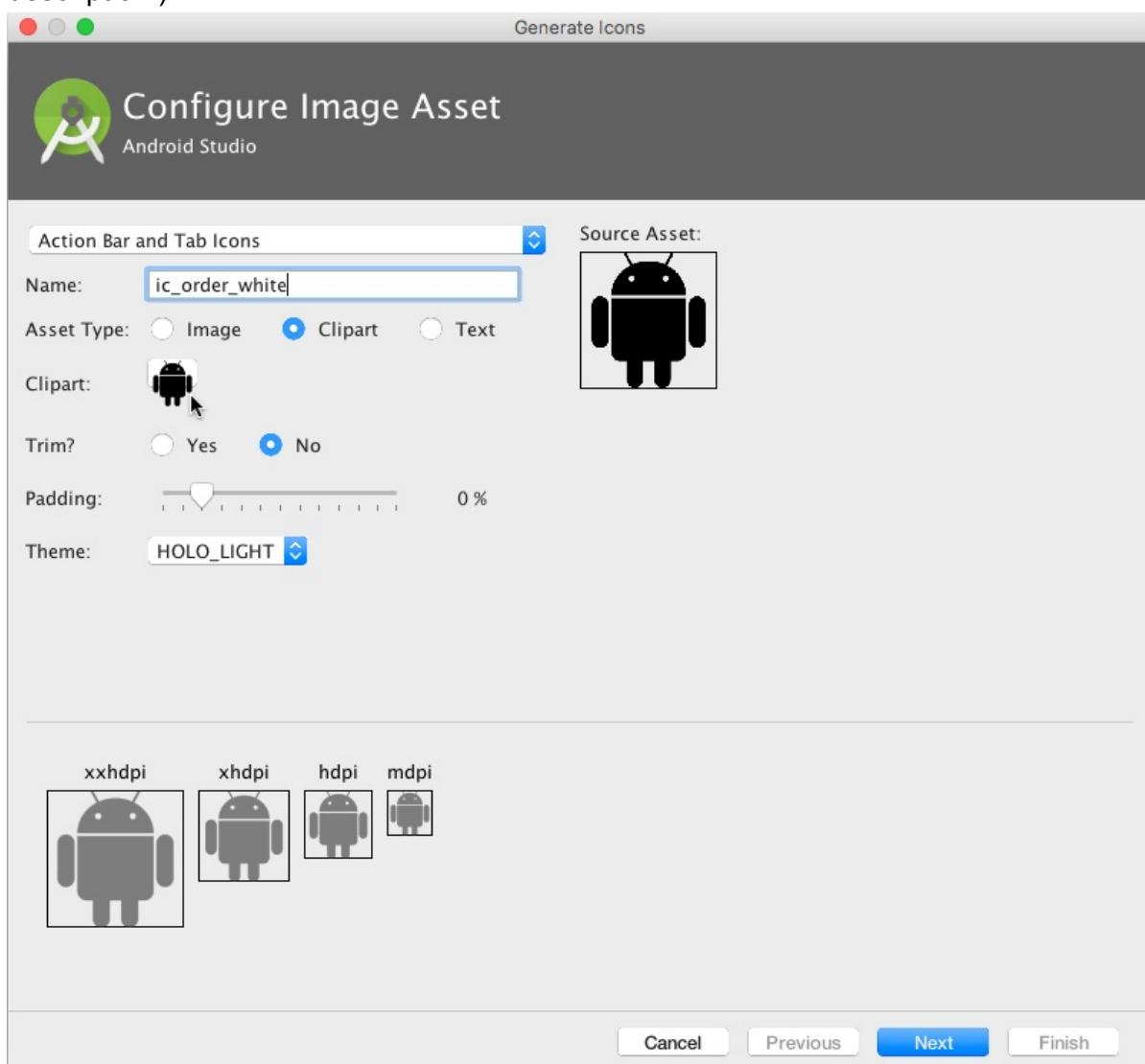
- Order and Status should always be shown in the app bar.

- Favorites should be shown in the app bar if it will fit; if not, it should appear in the overflow menu.
- Contact should not appear in the app bar; it should *only* appear in the overflow menu.

## 2.1 Add icons for menu items

To specify icons for actions, you need to first add the icons as image assets to the **drawable** folder.

1. Expand **res** in the Project view, and right-click (or Command-click) **drawable**.
2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.
3. Choose **Action Bar and Tab Items** in the drop-down menu.
4. Change **ic\_action\_name** to **ic\_order\_white** (for the Order action). The Configure Image Asset screen should look as follows (see [Image Asset Studio](#) for a complete description.)



5. Click the clipart image (the Android logo) to select a clipart image as the icon. A page of icons appears. Click the icon you want to use for the Order action (for example, the

- clipboard icon may be appropriate).
6. Choose **HOLO\_DARK** from the Theme drop-down menu. This sets the icon to be white against a dark-colored (or black) background. Click **Next**.
  7. Click **Finish** in the Confirm Icon Path dialog.
  8. Repeat the above steps for the Status and Favorites icons, naming them **ic\_status\_white** and **ic\_favorites\_white** respectively. You may want to use the circled-i icon for Status (typically used for Info), and the heart icon for Favorites.

## 2.2 Show the menu items as icons in the app bar

To show menu items as icons in the action bar, use the `app:showAsAction` attribute in **menu\_main.xml**. The following values for the attribute specify whether or not the action should appear in the action bar as an icon:

- "always" : Always appears in the action bar.
- "ifRoom" : Appears in the action bar if there is room.
- "never" : Never appears in the action bar; its text appears in the overflow menu.
- Open **menu\_main.xml** again, and add the following attributes to the Order, Status, and Favorites items so that the first two (Order and Status) always appear, and the Favorites item appears only if there is room for it:

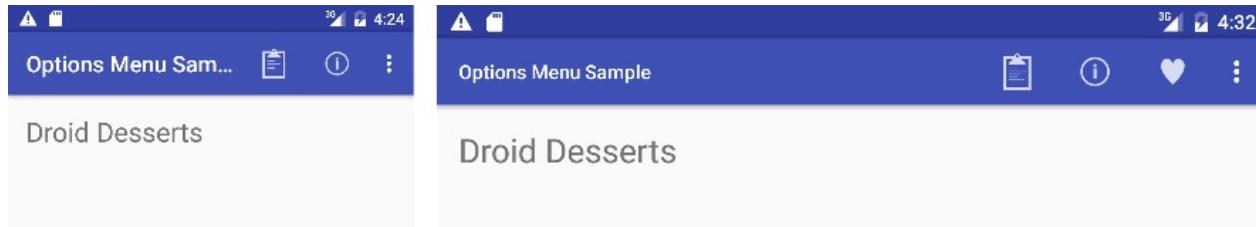
<b>Order Item Attribute</b>	<b>Old Value</b>	<b>New Value</b>
android:icon		"@drawable/ic_order_white"
app:showAsAction	"never"	"always"

<b>Status Item Attribute</b>	<b>Old Value</b>	<b>New Value</b>
android:icon		"@drawable/ic_status_white"
app:showAsAction	"never"	"always"

<b>Favorites Item Attribute</b>	<b>Old Value</b>	<b>New Value</b>
android:icon		"@drawable/ic_favorites_white"
app:showAsAction	"never"	"ifRoom"

- Run the app. You should now see at least two icons in the app bar: Order and Status. If your device or the emulator is displaying in vertical orientation, the Favorites and Contact options appear in the overflow menu.
- Rotate your device to the horizontal orientation, or if you're running in the emulator, click the **Rotate Left** or **Rotate Right** icons to rotate the display into the horizontal orientation. You should then see all three icons in the app bar: Order, Status, and Favorites.

**Tip:** How many actions will fit in the app bar? It depends on the orientation and the size of the device screen. Action buttons may not occupy more than half of the main app bar's width.



## 2.3 Add images to the layout to simulate a dessert app

1. Open `content_main.xml`, and change the `TextView` in the layout to use a larger text size of `24sp` and padding of `10dp`, and add the `android:id` attribute with the id `textintro`.
2. Extract the dimension resource for the `android:padding` attribute to the resource name `padding_regular`, and the `android:textsize` attribute to the resource name `text_heading`.
3. Add another `TextView` under the `textintro` `TextView` with the following attributes:

TextView Attribute	Value
<code>android:layout_width</code>	"wrap_content"
<code>android:layout_height</code>	"wrap_content"
<code>android:padding</code>	"@dimen/padding_regular"
<code>android:id</code>	"@+id/choose_dessert"
<code>android:layout_below</code>	"@+id/textintro"
<code>android:text</code>	"Choose one dessert with your order."

4. Extract the string resource for the `android:text` attribute to the resource name `choose`.
5. Copy the images into the project. The images named `donut_circle.jpg`, `froyo_circle.jpg`, and `icecream_circle.jpg` are provided with the app. To add the images, close the project, copy the image files into the drawable folder (`project_name > app > src > main > res > drawable`), and reopen the project.
6. Open `content_main.xml` file again and add an `ImageView` to the layout for the donut under the `TextView` to show the first image, using the following attributes:

<b>ImageView Attribute for donut</b>	<b>Value</b>
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:padding	"@dimen/padding_regular"
android:id	"@+id/donut"
android:layout_below	"@id/choose_dessert"
android:contentDescription	"Donuts are glazed and sprinkled with candy."
android:src	"@drawable/donut_circle"

7. Extract the  `android:contentDescription`  attribute value to the string resource  `donut`  .
8. Add a  `TextView`  that will appear next to the donut as a description, with the following attributes:

<b>TextView Attribute</b>	<b>Value</b>
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:padding	"35dp"
android:layout_below	"@+id/choose_dessert"
android:layout_toRightOf	"@id/donut"
android:text	"Donuts are glazed and sprinkled with candy."

9. Extract the  `android:text`  attribute value to the string resource  `donut_description`  .
10. Extract the dimension resource for the  `android:padding`  attribute to the resource name  `padding_wide`  .
11. Add a second  `ImageView`  to the layout for the ice cream sandwich, using the following attributes:

<b>ImageView Attribute for ice_cream</b>	<b>Value</b>
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:padding	"@dimen/padding_regular"
android:id	"@+id/ice_cream"
android:layout_below	"@id/donut"
android:contentDescription	"Ice cream sandwich"
android:src	"@drawable/icecream_circle"

12. Extract the  `android:contentDescription`  attribute value to the string resource

```
ice_cream_sandwich .
```

13. Add a `TextView` that will appear next to the ice cream sandwich as a description, with the following attributes:

<b>TextView Attribute</b>	<b>Value</b>
<code>android:layout_width</code>	"wrap_content"
<code>android:layout_height</code>	"wrap_content"
<code>android:padding</code>	"@dimen/padding_wide"
<code>android:layout_below</code>	"@+id/donut"
<code>android:layout_toRightOf</code>	"@id/ice_cream"
<code>android:text</code>	"Ice cream sandwiches have chocolate wafers and vanilla filling."

14. Extract the `android:text` attribute to the string resource `ice_cream_description`.  
 15. Add a third `ImageView` to the layout for the froyo, using the following attributes:

<b>ImageView Attribute for ice_cream</b>	<b>Value</b>
<code>android:layout_width</code>	"wrap_content"
<code>android:layout_height</code>	"wrap_content"
<code>android:padding</code>	"@dimen/padding_regular"
<code>android:id</code>	"@+id/froyo"
<code>android:layout_below</code>	"@id/ice_cream"
<code>android:contentDescription</code>	"Froyo"
<code>android:src</code>	"@drawable/froyo_circle"

16. Extract the `android:contentDescription` attribute value to the string resource `froyo`.  
 17. Add a `TextView` that will appear next to the froyo as a description, with the following attributes:

<b>TextView Attribute</b>	<b>Value</b>
<code>android:layout_width</code>	"wrap_content"
<code>android:layout_height</code>	"wrap_content"
<code>android:padding</code>	"@dimen/padding_wide"
<code>android:layout_below</code>	"@+id/ice_cream"
<code>android:layout_toRightOf</code>	"@id/froyo"
<code>android:text</code>	"FroYo is premium self-serve frozen yogurt."

18. Extract the `android:text` attribute to the string resource `froyo_description`.  
 19. Run the app.

The following screen layout should appear, and you can click the options menu. In the next section you will add handlers for the options menu items.

▲ ! ⌂ ⌚ 3G 2:02

## Options Menu Sam...

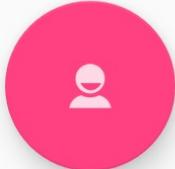
Droid Desserts

Choose one dessert with your order:

 Donuts are glazed and sprinkled with candy.

 Ice cream sandwiches have chocolate wafers and vanilla filling.

 FroYo is premium self-serve frozen yogurt.



◀ ○ □

## Coding challenge #1 (optional)

When you click the floating action button with the email icon that appears at the bottom of the screen, the code in `MainActivity` displays a drawer that opens and closes, called a *Snackbar*. A Snackbar provides feedback about an operation — it shows a brief message at the bottom of the screen on a smartphone, or in the lower left corner on larger devices.

Look at how other apps implement the floating action button. For example, the Gmail app provides a floating action button to create a new email message, and the Contacts app provides one to create a new contact. For more information, see [Snackbar](#).

Now that you know how to add icons for menu items, use the same technique to add another icon, and assign that icon to the floating action button, replacing the email icon. For example, you might want the floating action button to start a chat session; in which case you might want to use an icon showing a human.

**Hint:** The floating action button is defined in `activity_main.xml`.

While adding the icon, also change the text that appears in the Snackbar after tapping the floating action button. You will find this text in the `Snackbar.make` statement in the main activity. Extract the string resource for this text to be `snackbar_text`.

## Task 3. Handle the selected menu item

In this task, you'll add a method to display a message about which menu item is tapped, and use the `onOptionsItemSelected()` method to determine which menu item was tapped.

### 3.1 Create a method to display the menu choice

1. Open `MainActivity`.
2. Add the following method for displaying a toast message:

```
public void displayToast(String message) {  
    Toast.makeText(getApplicationContext(), message, Toast.LENGTH_SHORT).show();
```

The `displayToast()` method gets the `message` from the appropriate string (such as `action_contact_message`).

### 3.2 Use the `onOptionsItemSelected` event handler

The `onOptionsItemSelected()` method handles selections from the options menu if you don't specify a method with each item using an `onClick` attribute in the menu resource file.

You will add a `switch case` block to determine which menu item was selected, and what message to create for each selected item. You could implement event handlers rather than messages that perform actions, such as starting another activity, as shown later in this lesson.

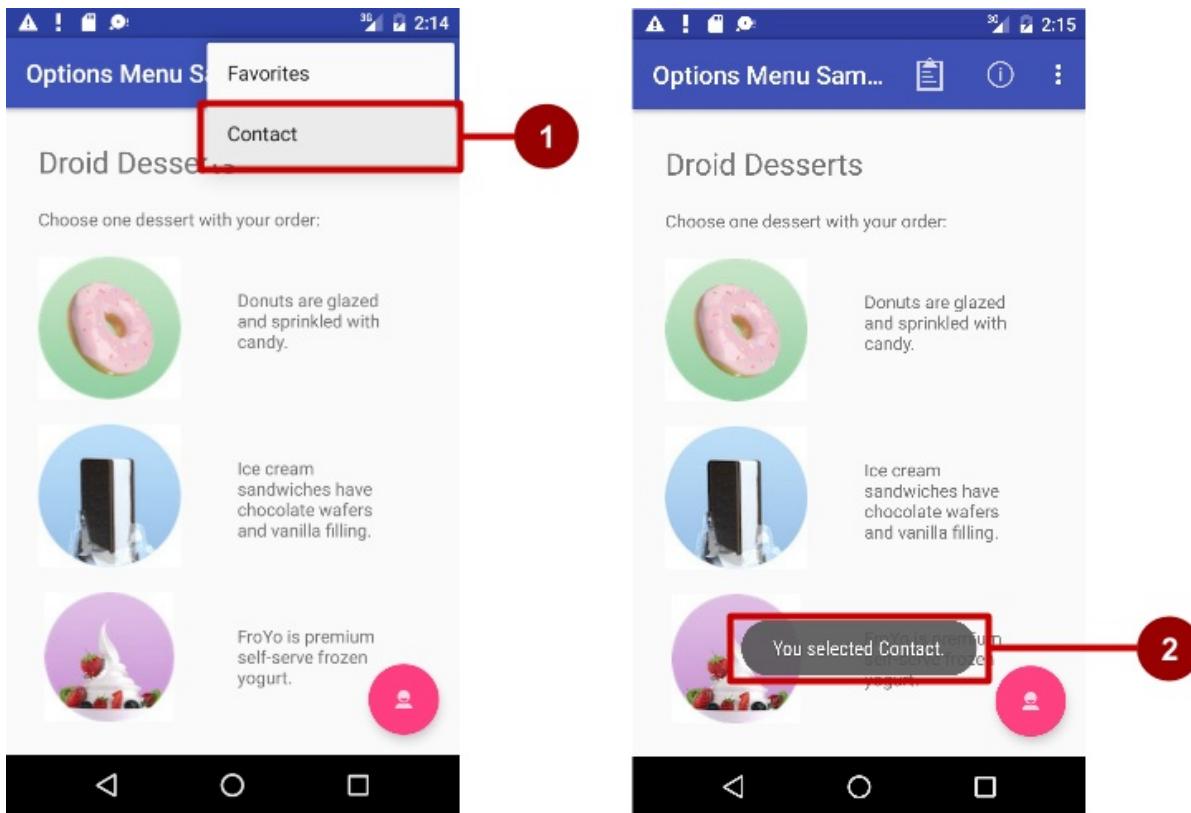
1. Find the `onOptionsItemSelected()` method. The `if` statement in the method, provided by the template, determines if a certain menu item was clicked, using the menu item's `id` (`action_order` in the below example):

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    int id = item.getItemId();  
    if (id == R.id.action_order) {  
        return true;  
    }  
    return super.onOptionsItemSelected(item);  
}
```

2. Replace the `if` statement with the following `switch case` block that sets the appropriate message based on the menu item's `id`:

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    switch (item.getItemId()) {  
        case R.id.action_order:  
            displayToast(getString(R.string.action_order_message));  
            return true;  
        case R.id.action_status:  
            displayToast(getString(R.string.action_status_message));  
            return true;  
        case R.id.action_favorites:  
            displayToast(getString(R.string.action_favorites_message));  
            return true;  
        case R.id.action_contact:  
            displayToast(getString(R.string.action_contact_message));  
            return true;  
    }  
    return super.onOptionsItemSelected(item);  
}
```

3. Run the app. You should now see a different toast message on the screen based on which menu item you choose.



In the above figure:

4. Selecting the Contact item in the options menu.
5. The toast message that appears.

## Solution code (includes code challenge)

`OptionsMenuSample` \*\*

## Task 4. Add radio buttons, images buttons, and up navigation

Radio buttons are useful for selecting only one option from a set of options. You should use radio buttons if you want the user to see all available options side-by-side. If it's not necessary to show all options side-by-side, you may want to use a spinner instead.

For an overview and sample code for radio buttons, see [Radio Buttons](#) in the User Interface section of the Android Developer Documentation.

### 4.1 Add another activity

1. Copy the **OptionsMenuSample** project folder, rename it to **OptionsMenuOrderActivity**, and refactor it. (See the [Appendix](#) for instructions on copying a project.)

2. After refactoring, change the `<string name="app_name">` value in the **strings.xml** file (within **app > res > values**) to **Options Menu Order Activity** (with spaces) as the app's name.
3. Right-click the folder for the app in the left column (`com.example.android.optionsmenuorderactivity`) and choose **New > Activity > Empty Activity**. Edit the Activity Name to be **OrderActivity**, and the Layout Name to be **activity\_order**. Leave the other options alone, and click **Finish**.

The `OrderActivity` class should now be listed under `MainActivity` in the **java** folder, and `activity_order.xml` should now be listed in the layout folder. The Empty Activity template added these files.

4. Open `MainActivity`, and change the `case R.id.action_order` statement from the previous section to the following in order to start `OrderActivity`:

```
case R.id.action_order:  
    Intent intent = new Intent(this, OrderActivity.class);  
    startActivity(intent);  
    return true;
```

## 4.2 Add the layout for radio buttons

To create each radio button option, you will create **RadioButton** elements in the `activity_order.xml` layout file, which is linked to `OrderActivity`.

Since radio buttons are mutually exclusive, you will group them together inside a **RadioGroup**. By grouping them together, the system ensures that only one radio button can be selected at a time. The order in which you list the `RadioButton` elements determines the order that they appear on the screen.

1. Copy a placeholder image into the project to show a prototype of the app's feature screen. The image "droid\_deserts\_order\_page.jpg" is provided with the app. To add the image to the project, close the project, copy the image file into the drawable folder (`project_name > app > src > main > res > drawable`), and reopen the project.
2. Open **activity\_order.xml**, and add an `ImageView` to the `RelativeLayout` to show the placeholder image, using the following attributes:

ImageView Attribute	Value
android:id	"@+id/order_layout_image"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_alignParentTop	"true"
android:contentDescription	"Mockup Image"
android:src	"@drawable/droid_deserts_order_page"

3. Add a `TextView` element under the `ImageView` element with the id `orderintrotext`:

TextView Attribute	Value
android:id	"@+id/orderintrotext"
android:layout_width	"match_parent"
android:layout_height	"match_parent"
android:layout_below	"@+id/order_layout_image"
android:layout_marginTop	"24dp"
android:layout_marginBottom	"6dp"
android:textSize	"18sp"
android:text	"Choose a delivery method:"

4. Extract the string resource for "choose a delivery method:" to be `choose_delivery_string`.
5. Extract the dimension resources for the margin values:
6. "24dp" to `text_margin_top`
7. "6dp" to `text_margin_bottom`
8. "18sp" to `intro_text_size`
9. Add a `RadioGroup` to the layout underneath the `TextView` you just added: ``xml

1. Add the following three `RadioButton` elements within the `RadioGroup`, using the following attributes. The `onRadioButtonClicked` entry for the `onClick` attribute will be highlighted until you add that method in the next task.

```
<table>
<tr>
<td><strong>RadioButton #1 Attribute</strong>
</td>
<td><strong>Value</strong>
</td>
</tr>
<tr>
<td>android:id
</td>
```

```
<td>"@+id/sameday"
</td>
</tr>
<tr>
<td>android:layout_width
</td>
<td>"wrap_content"
</td>
</tr>
<tr>
<td>android:layout_height
</td>
<td>"wrap_content"
</td>
</tr>
<tr>
<td>android:text
</td>
<td>"Same day messenger service"
</td>
</tr>
<tr>
<td>android:onClick
</td>
<td>"onRadioButtonClicked"
</td>
</tr>
</table>
<table>
<tr>
<td><strong>RadioButton #2 Attribute</strong>
</td>
<td><strong>Value</strong>
</td>
</tr>
<tr>
<td>android:id
</td>
<td>"@+id/nextday"
</td>
</tr>
<tr>
<td>android:layout_width
</td>
<td>"wrap_content"
</td>
</tr>
<tr>
<td>android:layout_height
</td>
<td>"wrap_content"
</td>
</tr>
```

```
<tr>
<td>android:text
</td>
<td>"Next day ground delivery"
</td>
</tr>
<tr>
<td>android:onClick
</td>
<td>"onRadioButtonClicked"
</td>
</tr>
</table>
<table>
<tr>
<td><strong>RadioButton #3 Attribute</strong>
</td>
<td><strong>Value</strong>
</td>
</tr>
<tr>
<td>android:id
</td>
<td>"@+id/pickup"
</td>
</tr>
<tr>
<td>android:layout_width
</td>
<td>"wrap_content"
</td>
</tr>
<tr>
<td>android:layout_height
</td>
<td>"wrap_content"
</td>
</tr>
<tr>
<td>android:text
</td>
<td>"Pick up"
</td>
</tr>
<tr>
<td>android:onClick
</td>
<td>"onRadioButtonClicked"
</td>
</tr>
</table>
```

1. Extract the three string resources for the `android:text` attributes to the following names, so that the strings can be easily translated:

```
*   `same_day_messenger_service`  
*   `next_day_ground_delivery`  
*   `pick_up`  
  
<a id="task4steps3" />  
  
### **4.3 Add the radio button handler**
```

The `android:onClick` attribute for each radio button element specifies the `onRadioButtonClicked()` method to handle the click event. Therefore, you need to add a new `onRadioButtonClicked()` method in the `OrderActivity` class.

You also need the same `displayToast()` method in this class that you created in the MainActivity class, in order to display the `message` about which item was tapped:

1. Open **strings.xml** and create a string resource named `chosen` for the string `"  
Chosen: "` (include the space after the colon).

1. Open **OrderActivity** and add the `displayToast` method:

```
```java  
public void displayToast(String message) {  
    Toast.makeText(getApplicationContext(), message,  
        Toast.LENGTH_SHORT).show();  
}
```

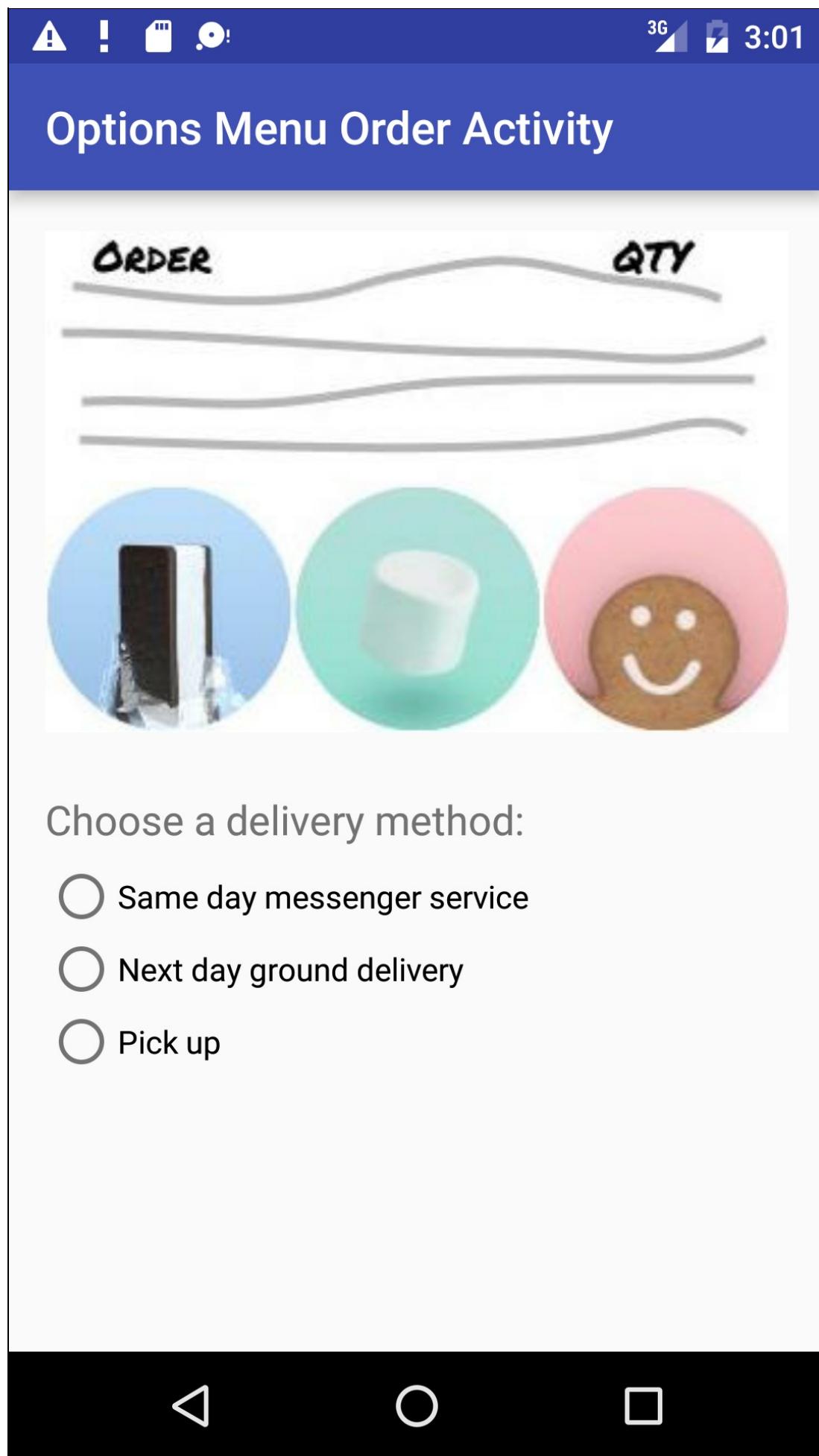
1. Add the following `onRadioButtonClicked()` method, which checks to see if a radio button has been checked, and uses a `switch case` block to determine which radio button item was selected, in order to set the appropriate `message` for that item to use with

```
displayToast() :
```

```
public void onRadioButtonClicked(View view) {  
    // Is the button now checked?  
    boolean checked = ((RadioButton) view).isChecked();  
    // Check which radio button was clicked  
    switch(view.getId()) {  
        case R.id.sameday:  
            if (checked)  
                // Same day service  
                displayToast(getString(R.string.chosen) +  
                            getString(R.string.same_day_messenger_service));  
            break;  
        case R.id.nextday:  
            if (checked)  
                // Next day delivery  
                displayToast(getString(R.string.chosen) +  
                            getString(R.string.next_day_ground_delivery));  
            break;  
        case R.id.pickup:  
            if (checked)  
                // Pick up  
                displayToast(getString(R.string.chosen) +  
                            getString(R.string.pick_up));  
            break;  
    }  
}
```

## 2. Run the app.

You should now be able to tap the **Order** button in the options menu to go to the Order Activity screen, which offers a choice of delivery methods for your order.



## 4.4 Add an Up navigation button to the app bar

Your app should make it easy for users to find their way back to the app's main screen, which is the parent activity. One simple way to do this is to provide an Up button on the app bar for all activities that are children of the parent activity. When the user touches the Up button, the app navigates to the parent activity.

**Tip:** The back button below the screen differs from the Up button. The back button provides navigation to whatever screen you viewed previously. If you have several children activities that the user can navigate through, the back button would send the user back to the previous child activity. Use an Up button if you want to provide one button to navigate from any child activity back to the parent activity. For more information about Up navigation, see [Providing Up Navigation](#).

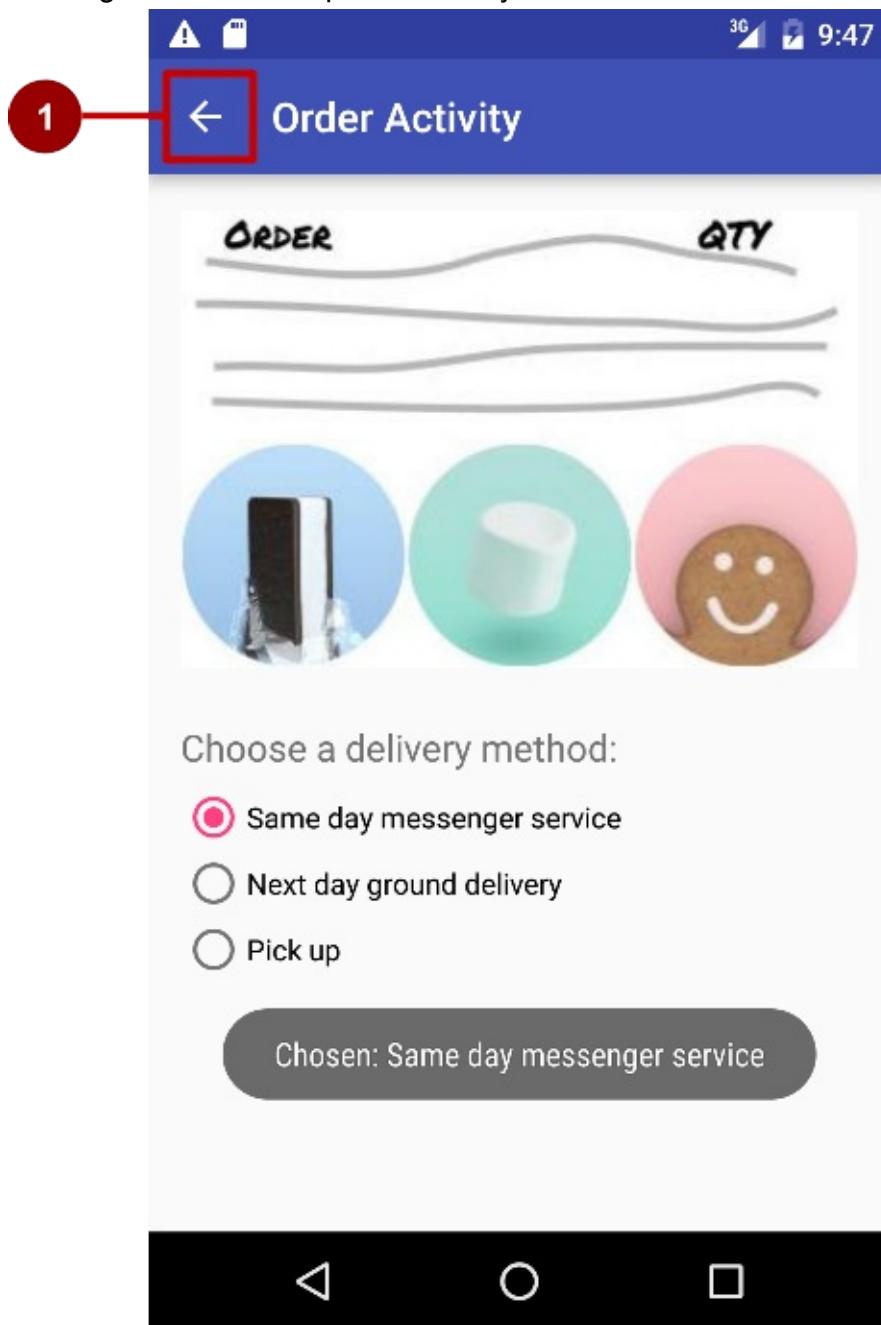
As you learned previously, when adding activities to an app, you can add Up-button navigation to a child activity such as `OrderActivity` by declaring the activity's parent to be `MainActivity` in the `AndroidManifest.xml` file. You can also set the `android:label` to a title for the activity screen, such as `"Order Activity"` (extracted into the string resource `title_activity_order` for the code below):

1. Open `AndroidManifest.xml`.
2. Change the activity element for `OrderActivity` to the following:

```
<activity android:name=".OrderActivity"
    android:label="@string/title_activity_order"
    android:parentActivityName="com.example.android.
                                optionsmenuorderactivity.MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

3. Run the app.

The Order Activity screen now includes the Up button (#1 in the figure below) in the app bar to navigate back to the parent activity.



## 4.5 Add onClick methods for image buttons

The images you added to the `content_main.xml` layout for `MainActivity` can be turned into image buttons by adding the `android:onClick` attribute to each one. You can easily add toast messages for these image buttons and use the `displayToast()` method you added previously to `MainActivity`.

1. Add the following string resources to the `strings.xml` file:

```
<string name="donut_order_message">You ordered a donut.</string>
<string name="ice_cream_order_message">You ordered an ice cream sandwich.</string>
<string name="froyo_order_message">You ordered a FroYo.</string>
```

2. Add the following methods to the **MainActivity.java** file:
- ```
```java /**
 * Shows a message that the donut was clicked. */
public void showDonutOrder(View view) { showToast(getString(R.string.donut_order_message)); }

/**
 * Shows a message that the ice cream sandwich was clicked. */
public void showIceCreamOrder(View view) {
    showToast(getString(R.string.ice_cream_order_message));
}

/**
 * Shows a message that the froyo was clicked. */
public void showFroyoOrder(View view) { showToast(getString(R.string.froyo_order_message)); }
```

```
1. Add the `android:onClick` attribute to the three ImageViews in content_main.xml:  
```xml  
    <ImageView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:padding="10dp"  
        android:id="@+id/donut"  
        android:layout_below="@+id/choose_dessert"  
        android:contentDescription="@string/donut"  
        android:src="@drawable/donut_circle"  
        android:onClick="showDonutOrder"/>  
    . . .  
    <ImageView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:padding="10dp"  
        android:id="@+id/ice_cream"  
        android:layout_below="@+id/donut"  
        android:contentDescription="@string/ice_cream_sandwich"  
        android:src="@drawable/icecream_circle"  
        android:onClick="showIceCreamOrder"/>  
    . . .  
    <ImageView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:padding="10dp"  
        android:id="@+id/froyo"  
        android:layout_below="@+id/ice_cream"  
        android:contentDescription="@string/froyo"  
        android:src="@drawable/froyo_circle"  
        android:onClick="showFroyoOrder"/>
```

---

### 1. Run the app.

Clicking the donut, ice cream sandwich, or froyo image displays a toast message about the order.

▲ ! ⌂ ⌚ 3G 3:31

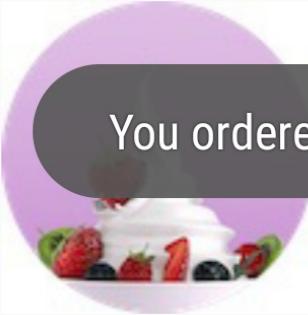
## Options Menu Ord...

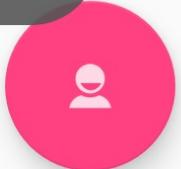
Droid Desserts

Choose one dessert with your order:

 Donuts are glazed and sprinkled with candy.

 Ice cream sandwiches have chocolate wafers and vanilla filling.

  
You ordered an ice cream sandwich.  
FroYo is premium self-serve frozen yogurt.



◀ ○ □

## Solution code

OptionsMenuOrderActivity project: \*\*

## Coding challenge #2 (optional)

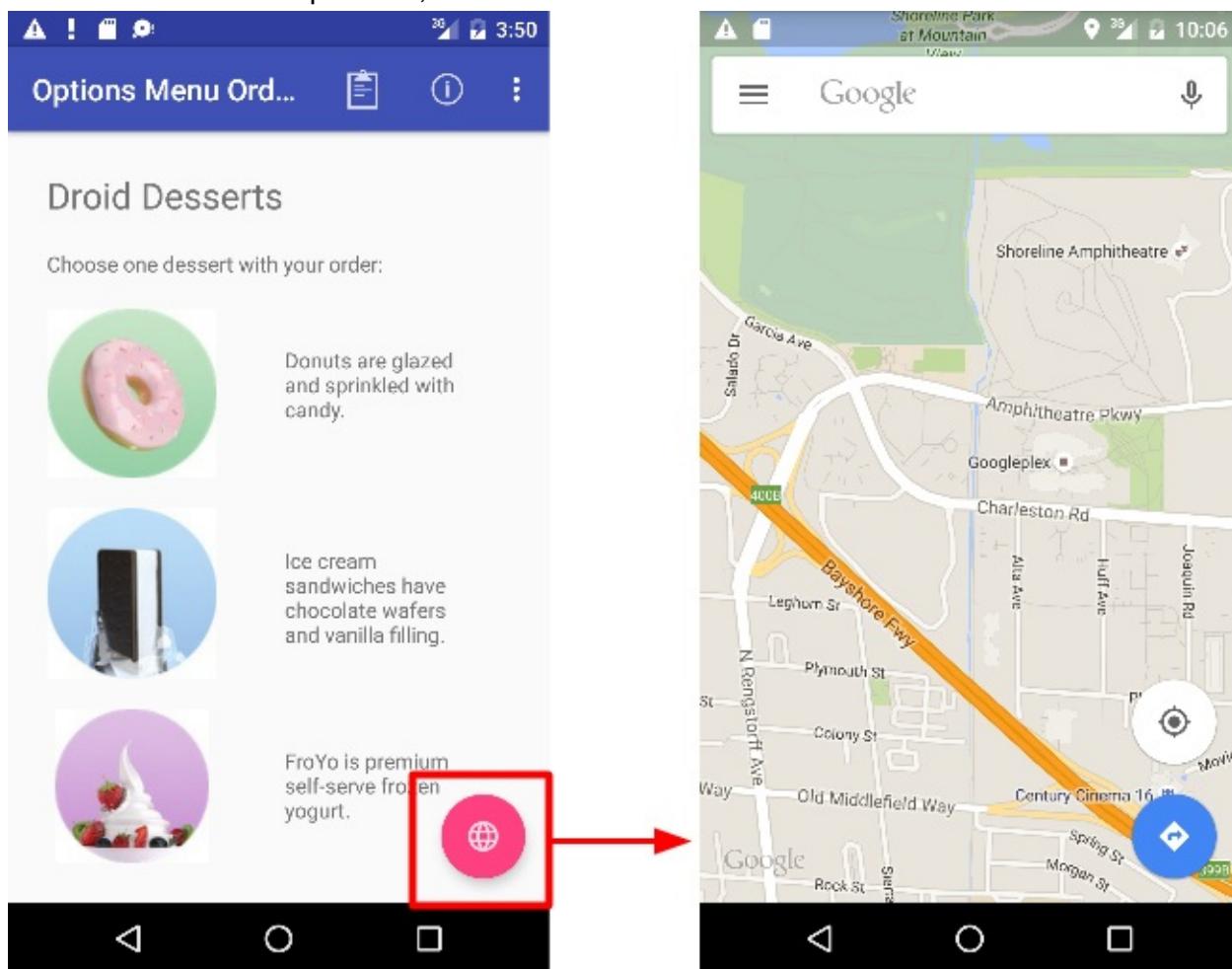
In the previous challenge, you changed the icon for the floating action button that appears at the bottom of the `MainActivity` screen, and you changed the text in the snackbar drawer that opens and closes.

For this challenge, change the icon for the floating action button to a map icon. In `MainActivity`, change the behavior of displaying a snackbar to making an implicit intent to launch the Maps app when the floating action button is tapped. You can use specific coordinates and the following method to start the Maps app:

```
public void displayMap() {
    Intent intent = new Intent();
    intent.setAction(Intent.ACTION_VIEW);
    // Using the coordinates for Google headquarters.
    String data = String.format("geo:%s,%s", 37.422114, -122.086744);
    String zoomLevel = "10";
    if (zoomLevel != null) {
        data = String.format("%s?z=%s", data, zoomLevel);
    }
    intent.setData(Uri.parse(data));
    if (intent.resolveActivity(getApplicationContext()) != null) {
        startActivity(intent);
    }
}
```

For examples of implicit intents including opening the Maps app, see [Common Implicit Intents](#) on github.

Be sure to also change the icon for the floating action button in `MainActivity` to something more suitable for a map button, such as the world icon.



## Solution Code

OptionsMenuOrderActivityMap project: \*\*

## Summary

In this practical, you learned the following:

- Setting up an options menu in the app bar:
  - Using the Basic Activity template to automatically set up the options menu and a floating action button.
  - Using `@string/appbar_scrolling_view_behavior` to provide the standard scrolling behavior of the app bar's options menu.
  - Using a `CoordinatorLayout` view group with the `AppBarLayout` class to create an options menu in the app bar.
  - Using an `include layout` statement in an XML layout file to include an entire layout defined in another XML file.

- Using the `NoActionBar` theme to prevent the app from using the native ActionBar class attributes for the app bar, in order to set the `windowActionBar` attribute to `false` (no window action bar), and the `windowNoTitle` attribute to `true` (no title).
  - Using an activity's `onCreate()` method to call the activity's `setSupportActionBar()` method to set the toolbar defined in the layout as the app bar for the activity.
  - Defining a menu and all its items in an XML menu resource, and then inflating the menu resource in an activity or fragment to load it as a Menu object.
  - Using the `android:orderInCategory` attribute to specify the order in which the menu items appear in the menu, with the lowest number appearing higher in the menu.
  - Using the `app:showAsAction` attribute to show menu items as icons in the app bar.
  - Adding event handlers for options menu items, and using the `onOptionsItemSelected()` method to retrieve the selection from the options menu.
- Using images and icons in a project:
    - Copying an image into the project, and defining an `ImageView` element to use it.
    - Adding icons to a project and using them to show menu items in the app bar.
    - Challenge: Changing the icon for a Snackbar, and changing the `Snackbar.make` code.
  - Using radio buttons:
    - Adding `RadioButton` elements within a `RadioGroup`.
    - Creating radio button handlers.
  - Adding Up-button navigation to a child activity by declaring the activity's parent in the `AndroidManifest.xml` file.
  - Challenge: Making an implicit intent to launch the Maps app with specific coordinates.

## Resources

- Android Developer Reference:
  - [AppBarLayout](#)
  - [Radio Buttons](#) (User Interface section)
  - [Toolbar](#)
  - [Menus](#)
  - [Providing Up Navigation](#)
- Android Developers Blog: [Android Design Support Library](#)
- Material Design Spec:
  - [App Bar](#)
  - [Scrolling Techniques](#)
- Best Practices for User Interface: [Adding the App Bar](#)
- Github: [Common Implicit Intents](#)
- Images and icons:

- [Image Asset Studio](#)
- [Compare Icons for Drawables](#)
- [Icons and other downloadable resources](#)

## 4.3 PC: Tab Navigation

### Contents:

- [App Overview](#)
- [Challenge](#)
- [Solution](#)
- [Resources](#)

Tabs appear across the top of a screen, providing navigation to other screens. Tab navigation is a very popular solution for lateral navigation from one child screen to another child screen that is a *sibling* — in the same position in the hierarchy and sharing the same parent screen. Tabs are most appropriate for small sets (four or fewer) of child screens.

The main class used for displaying tabs is [TabLayout](#). It provides a horizontal layout to display tabs. You can show the tabs below the app bar, and use the [PagerAdapter](#) class to populate pages inside of a [ViewPager](#), which is a layout manager that lets the user flip left and right through pages of data. You supply an implementation of a [PagerAdapter](#) to generate the pages that the view shows. ViewPager is most often used in conjunction with [Fragment](#), which is a convenient way to supply and manage the lifecycle of each page.

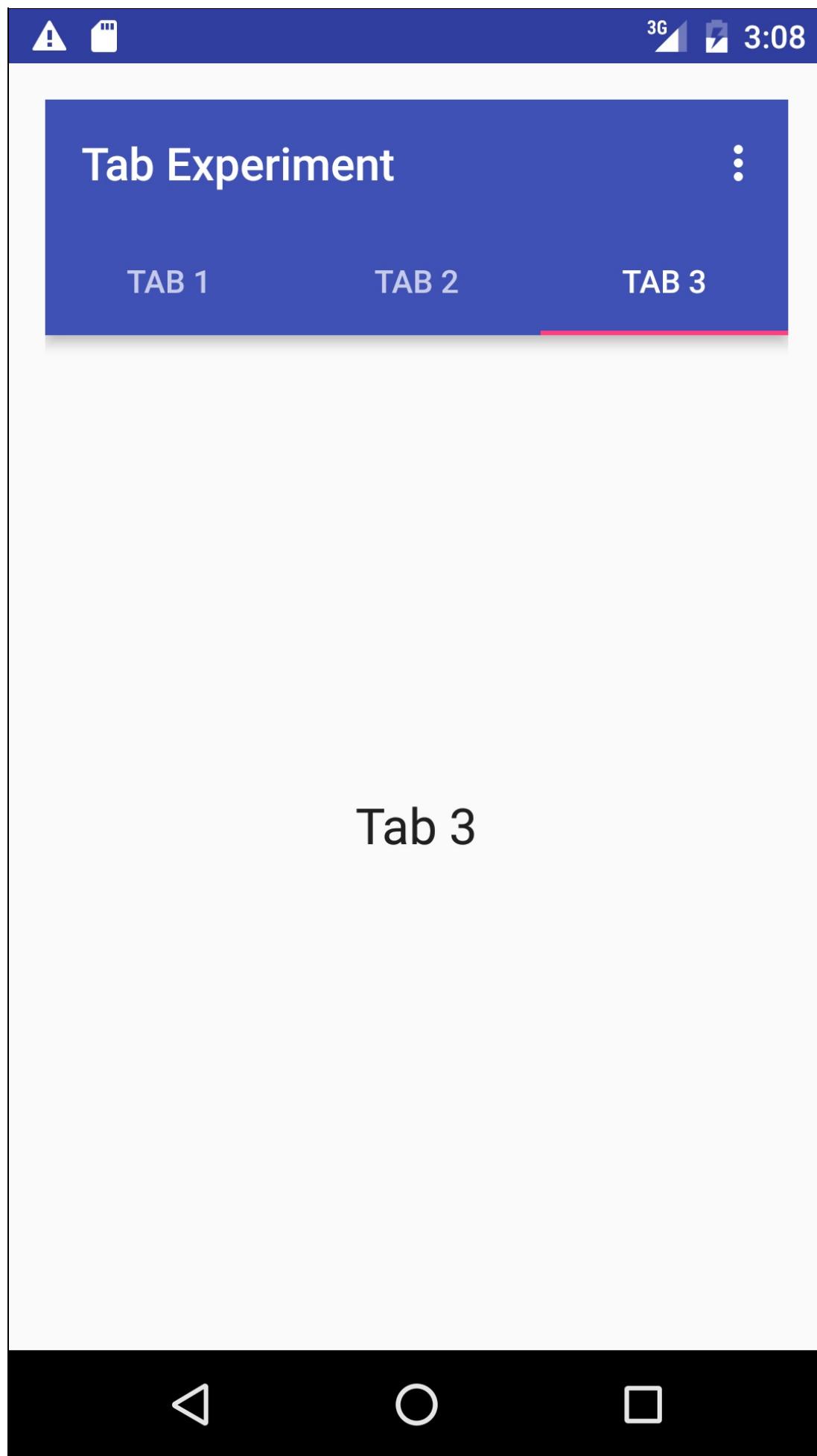
There are standard adapters for using fragments with the ViewPager:

- [FragmentPagerAdapter](#): Designed for navigating between sibling screens (pages) representing a fixed, small number of screens.
- [FragmentStatePagerAdapter](#): Designed for paging across a collection of screens (pages) for which the number of screens is undetermined. It destroys fragments as the user navigates to other screens, minimizing memory usage. The app for this practical challenge uses FragmentStatePagerAdapter.

## App Overview

The app for this practical challenge shows three tabs below the app bar to navigate to sibling screens. As the user taps a tab, the screen shows the “Tab 1”, “Tab 2”, or “Tab 3” screen depending on which tab is tapped. The user can also swipe left and right to visit the “Tab 1”, “Tab 2”, or “Tab 3” screens.

The app also includes the options menu with one menu item (Settings) that appears when the user clicks the overflow action icon in the app bar.



# Challenge

Follow these steps for this practical challenge:

1. Create a new project using the Empty Activity template. Name the app **Tab Experiment** or something similar.
2. In the **activity\_main.xml** layout, add a `Toolbar`, a `TabLayout`, and a `ViewPager` within the root layout. They should look like this:

```
<android.support.v7.widget.Toolbar  
    android:id="@+id/toolbar"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_alignParentTop="true"  
    android:background="?attr/colorPrimary"  
    android:minHeight="?attr/actionBarSize"  
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"  
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>  
  
<android.support.design.widget.TabLayout  
    android:id="@+id/tab_layout"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_below="@id/toolbar"  
    android:background="?attr/colorPrimary"  
    android:minHeight="?attr/actionBarSize"  
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"/>  
  
<android.support.v4.view.ViewPager  
    android:id="@+id/pager"  
    android:layout_width="match_parent"  
    android:layout_height="fill_parent"  
    android:layout_below="@id/tab_layout"/>
```

3. Create a layout for each of three sibling screens: **tab\_fragment1.xml**, **tab\_fragment2.xml**, and **tab\_fragment3.xml**. Each one should display a `TextView` with a string, such as “Tab 1”, “Tab 2”, or “Tab 3”.
4. Create a menu resource file (**menu\_main.xml**) for the options menu with one menu item: “Settings”. Include the `app:showAsAction="never"` attribute for the item so that it appears only when the user clicks the action overflow icon in the app bar.
5. Add a class for each fragment representing a screen the user can visit by clicking a tab: **TabFragment1.java**, **TabFragment2.java**, and **TabFragment3.java**. Each class extends `Fragment` and inflates the layout associated with the screen (`tab_fragment1`, `tab_fragment2`, and `tab_fragment3`). For example, **TabFragment1.java** looks like this:

```

public class TabFragment1 extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
                            ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.tab_fragment1, container, false);
    }
}

```

6. Add a `PagerAdapter` that extends `FragmentStatePagerAdapter` and:

- i. Defines the number of tabs.
- ii. Uses the `getItem()` method of the `Adapter` class to determine which tab is clicked.
- iii. Uses a switch case block to return the screen (page) to show based on which tab is clicked.

```

public class PagerAdapter extends FragmentStatePagerAdapter {
    int mNumOfTabs;

    public PagerAdapter(FragmentManager fm, int NumOfTabs) {
        super(fm);
        this.mNumOfTabs = NumOfTabs;
    }

    @Override
    public Fragment getItem(int position) {

        switch (position) {
            case 0:
                return new TabFragment1();
            case 1:
                return new TabFragment2();
            case 2:
                return new TabFragment3();
            default:
                return null;
        }
    }

    @Override
    public int getCount() {
        return mNumOfTabs;
    }
}

```

7. Inflate the `Toolbar` in the `onCreate()` method in **MainActivity.java**:

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

8. Create an instance of the tab layout from the `tab_layout` element in the layout, and set the text for each tab using `addTab()`:

```
// Create an instance of the tab layout from the view.
TabLayout tabLayout = (TabLayout) findViewById(R.id.tab_layout);
// Set the text for each tab.
assert tabLayout != null;
tabLayout.addTab(tabLayout.newTab().setText("Tab 1"));
tabLayout.addTab(tabLayout.newTab().setText("Tab 2"));
tabLayout.addTab(tabLayout.newTab().setText("Tab 3"));
// Set the tabs to fill the entire layout.
tabLayout.setTabGravity(TabLayout.GRAVITY_FILL);
```

9. Use `PagerAdapter` to manage screen (page) views in the fragments. Each screen is represented by its own fragment:

```
// Using PagerAdapter to manage page views in fragments --
// each page is represented by its own fragment.
// This is another example of the adapter pattern.
final ViewPager viewPager = (ViewPager) findViewById(R.id.pager);
final PagerAdapter adapter =
        (getSupportFragmentManager(), tabLayout.getTabCount());
assert viewPager != null;
viewPager.setAdapter(adapter);
// Setting a listener for clicks.
viewPager.addOnPageChangeListener(new
        TabLayout.TabLayoutOnPageChangeListener(tabLayout));
tabLayout.addOnTabSelectedListener(new TabLayout.OnTabSelectedListener() {
    @Override
    public void onTabSelected(TabLayout.Tab tab) {
        viewPager.setCurrentItem(tab.getPosition());
    }

    @Override
    public void onTabUnselected(TabLayout.Tab tab) {

    }

    @Override
    public void onTabReselected(TabLayout.Tab tab) {
    }
});
```

10. Inflate the options menu by overriding the `onCreateOptionsMenu()` method, and override the `onOptionsItemSelected()` method to handle the Settings option menu item.

# Solution

Android Project: Tab Navigation

## Resources

### Developer Documentation:

- [TabLayout](#)
- [Android Material Design working with Tabs](#)
- [Android Tabs Example – With Fragments and ViewPager](#)
- [Creating Swipe Views with Tabs](#)

## 4.4 P: Create a Recycler View

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Create and configure a WordList project](#)
- [Task 2: Create a dataset](#)
- [Task 3: Create a RecyclerView](#)
- [Task 4: Add onClick to list items](#)
- [Task 5: Add a FAB to insert items](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

Displaying and manipulating a scrollable list of similar data items, as you did in the scrolling view practical, is a common feature of apps. For example, contacts, playlists, photos, dictionaries, shopping lists, an index of documents, or a listing of saved games are all examples.

RecyclerView is a subclass of ViewGroup. It is a more resource-efficient way to display scrollable lists. You can define the appearance of each list item and update list items dynamically.

In this series of practicals you will use a RecyclerView to:

- Display a scrollable list of items.
- Add a click handler to each item.
- Add items to the list using a [floating action button \(FAB\)](#). In the screenshot below, it's the pink button. A floating action buttons can be used for common actions, a promoted action, that is, an action that you want the user to take.

## What you should already KNOW

For this practical you should be familiar with:

- Creating a Hello World app with Android Studio.
- Implementing different layouts for apps.

- Creating and using string resources.
- Adding an onClick handler to a view.

## What you will LEARN

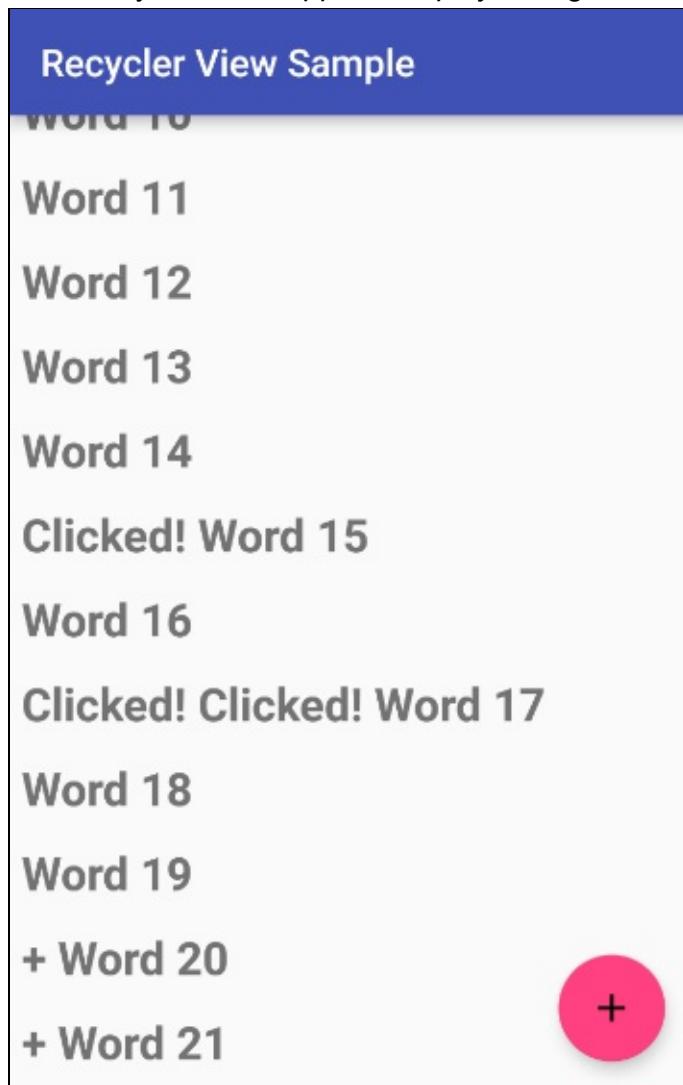
- How to use the RecyclerView class to display items in a scrollable list.
- How to dynamically add items to the RecyclerView.
- How to perform an action when the user taps a specific item.
- How to show a floating action button and perform an action when the user taps it.

## What you will DO

Create a new application that uses a [RecyclerView](#) to display a list of items as a scrollable list and associate click behavior to the list items. By using a floating action button, you will allow the user to add items to the RecyclerView.

## App Overview

The "RecyclerView" app will display a long list of words.



- Tapping an item marks it clicked.
- Tapping the floating action button adds an item.

## Task 1. Create and configure a new project

In this practical, you will create and configure a new project for the RecyclerView Sample app.

### 1.1. Create the project

1. Start Android Studio and create a new project with the following parameters

Attribute	Value
Application Name	Recyclerview
Company Name	com.example.android or your own domain
Phone and Tablet Minimum SDK	API15: Android 4.0.3 IceCreamSandwich
Template	Empty Activity
Generate Layout file box	Checked

- Build and run the app on an emulator or hardware device. You should see the "WordList" title and "Hello World" in a blank view.

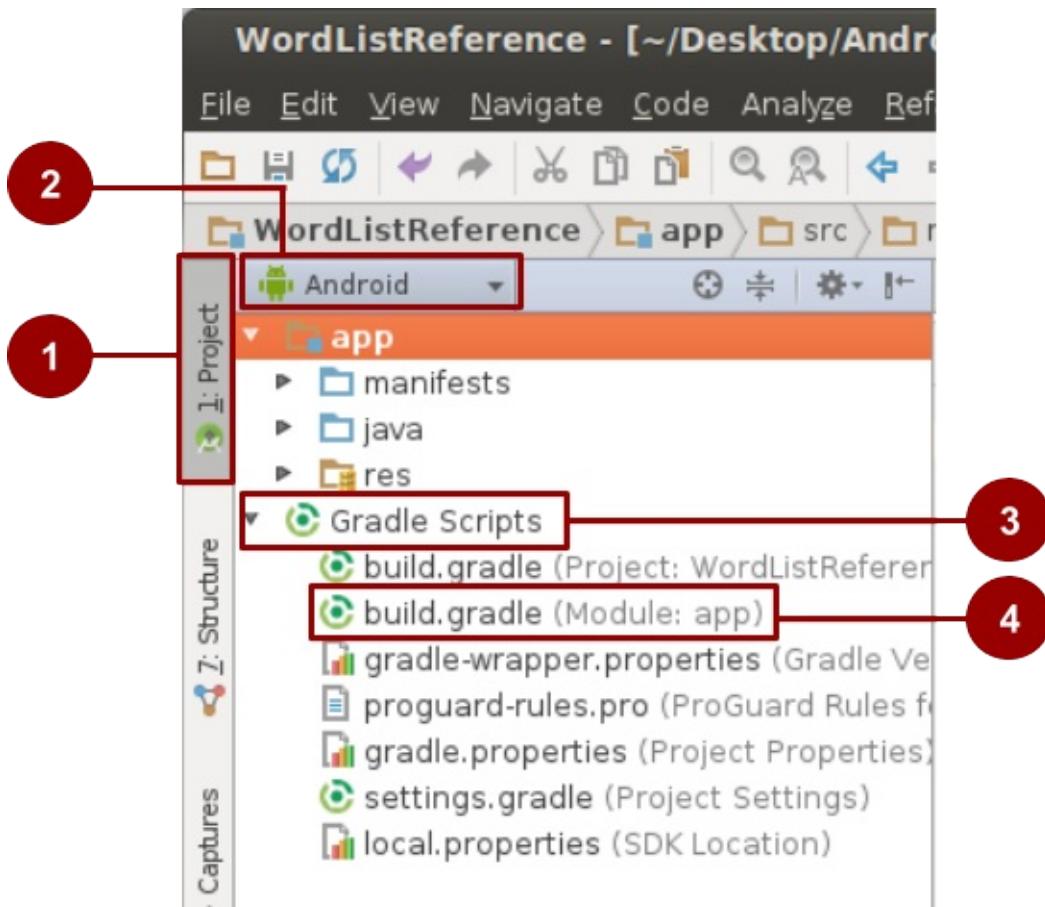
## 1.2. Add support libraries to the build file

In order to use the RecyclerView and the floating action button (FAB), you need to add the respective [Android Support Libraries](#) to your build.

**Why:** Android Support libraries provide backward-compatible versions of Android framework APIs, additional UI components and a set of useful utilities. The RecyclerView class is located in the Android Support package; two dependencies must be included in the Gradle build process to use it. You will learn more about support libraries in a later practical.

Follow these steps and refer to the screenshot:

- In Android Studio, in your new project, make sure you are in the **Project** pane (1) and in the **Android** view (2).
- In the hierarchy of files, find the **Gradle Scripts** folder (3).
- Expand **Gradle Scripts**, if necessary, and open the **build.gradle (Module: app)** file (4).  
(Note: build.gradle (Project: WordList) is NOT the right file to change.)



4. Towards the end of the **build.gradle (Module: app)** file, find the dependencies section.
5. Add these two dependencies as the last two lines (inside the curly braces):

```
compile 'com.android.support:recyclerview-v7:23.1.1'
compile 'com.android.support:design:23.1.1'
```

- There is probably an existing line like this one, but the number may be different:  
`compile 'com.android.support:appcompat-v7:23.1.1'`
- Add your lines below that line.
- **Match the version number of your lines to the version number of that existing line.**
- Make sure the version numbers of all the libraries are the same and match up with the `compileSdkVersion` at the top of the file. (If these don't match, you will get a build time error.)

6. If prompted, sync your app now.
7. Build and run your app. You should see the same WordList app displaying "Hello World". (If you get gradle errors, sync your project. You do not need to install additional plugins.)

### Solution:

This is an example of the dependencies section of the build.gradle file. Note that your file may be slightly different, e.g., the version number.

```
dependencies {  
    compile fileTree(dir: 'libs', include: ['*.jar'])  
    testCompile 'junit:junit:4.12'  
    compile 'com.android.support:appcompat-v7:23.1.1'  
    compile 'com.android.support:recyclerview-v7:23.1.1'  
    compile 'com.android.support:design:23.1.1'  
}
```

## Task 2. Create a dataset

Before you can display anything, you need data to display. In a more sophisticated app, your data could come from internal storage (a file, SQLite database, saved preferences), from another app (Contacts, Photos), or from the internet (cloud storage, Google Sheets, or any data source with an API). For this exercise, you will simulate data by creating it in the `MainActivity`'s `onCreate` method.

**Why:** Storing and retrieving data is a topic of its own covered in the data storage chapter. You will have an opportunity to extend `WordList` to use real data in that later lesson.

### 2.1. Add code to create data

In this practical you will dynamically create a linked list of twenty word strings that end in increasing numbers, such that `["Word 1", "Word 2", "Word 3", .... ]`.

Note: You must use a `LinkedList` for this practical. Refer to the solution code, if you need help.

1. Open the `MainActivity.java` file.
2. Add a class variable for the `mWordList` linked list.
3. Add an integer counter class variable to track the word's number.
4. Add code that populates `mWordList` with words. Concatenate the string "Word" with the value of `mCount`, then increase the count.
5. Since you cannot display the words yet for testing, add a log statement that verifies that words are correctly added to the list.
6. Build and run your app.

The app UI has not changed, but you should see a list of log messages in log cat, such as:  
`android.example.com.wordlist D/WordList: Word 1`

#### Solution:

Class variables:

```
private final LinkedList<String> mWordList = new LinkedList<>();
private int mCount = 0;
```

In the onCreate method of MainActivity:

```
for (int i = 0; i < 20; i++) {
    mWordList.addLast("Word " + mCount++);
    Log.d("WordList", mWordList.getLast());
}
```

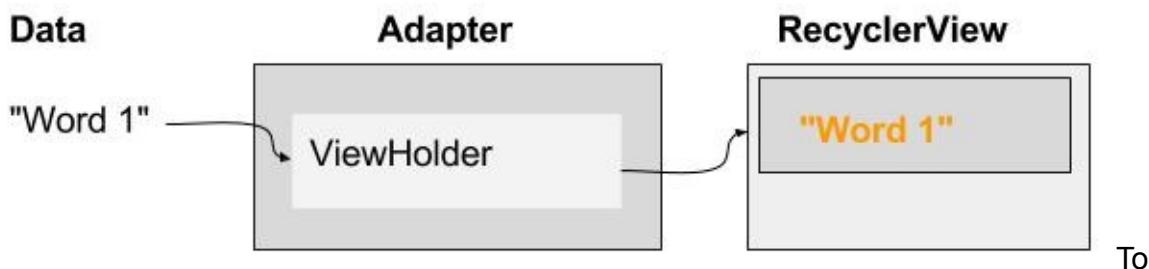
## Task 3. Create a RecyclerView

In this practical, you will display data in a RecyclerView. Since there are several parts to creating a working RecyclerView make sure you immediately fix any errors that you see in Android Studio.

To display your data in a RecyclerView, you need the following things:

- Data. You created that in the previous task.
- A layout manager. The layout manager manages the organization (layout) of user interface components in a view. You have already used the simple LinearLayout in a previous practical. RecyclerView requires a layout manager to manage the arrangement of list items contained within it. This layout could be vertical, horizontal, or as a grid. You will use a vertical linear layout manager provided by Android.
- An adapter. The adapter connects your data to the RecyclerView. Specifically you will create an adapter that matches your generated list data to your views. Inside the adapter, you will create a ViewHolder that contains the view information for displaying each item. A ViewHolder is an abstract class that implements the ViewHolder pattern which optimizes the performance of a list of displayed data. An Adapter is an abstract subclass of ViewHolder that implements the Adapter pattern to simplify the connection between the data in the ViewHolder and the layout manager.

The diagram below shows the relationship between the data, the adapter, the view holder, and the layout manager.



implement these pieces, you will need to:

1. Create the XML layout for the WordList app (`activity_main.xml`).
2. Create the XML layout for a single list item (`wordlist_item.xml`).
3. Create an adapter (`WordListAdapter`) with a view holder (`WordViewHolder`).
4. In the `onCreate` method of the `MainActivity`, create a `RecyclerView` and initialize it with the adapter and a standard layout manager.

Let's do these one at a time.

### 3.1. Create the main layout in `activity_main.xml`

In the previous apps, you used `LinearLayout` to arrange your views. In order to accommodate the `RecyclerView` and the floating action button that you will add later, you need to use a different view group called a coordinator layout. `CoordinatorLayout` is more flexible than `LinearLayout` when arranging views. For example, views like the floating action button can overlay other views.

In `main_activity.xml`, replace the code created by the Empty Activity with code for a `CoordinatorLayout`, and then add a `RecyclerView`:

1. Open `activity_main.xml`.
2. Select all the code in `activity_main.xml` and replace it with this code:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
</android.support.design.widget.CoordinatorLayout>
```

3. Inspect the code and note the following:
  - The properties specified for this view group are the same as for `LinearLayout`, because some basic properties, such as `layout_width` and `layout_height`, are required for all views and view groups.
  - Because `CoordinatorLayout` is in the support library, you have to specify the full path to the support library. You will have to do the same for the `RecyclerView`.
4. Add the `RecyclerView` code inside the `CoordinatorLayout`:
  - You need to specify the full path, because `RecyclerView` is part of the support library.

```
<android.support.v7.widget.RecyclerView>
</android.support.v7.widget.RecyclerView>
```

5. Give your `RecyclerView` the following properties:

Attribute	Value
android:id	"@+id/recyclerview"
android:layout_width	match_parent
android:layout_height	match_parent

6. Build and run your app and make sure there are no errors displayed in logcat. You will only see a blank screen, because you haven't put any items into the RecyclerView yet.

### Solution:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    </android.support.v7.widget.RecyclerView>

</android.support.design.widget.CoordinatorLayout>
```

### Resources:

- [CoordinatorLayout](#)
- [RecyclerView](#)
- Learn more about the [Android Support Library](#).

## 3.2. Create the layout for one list item

The adapter will need the layout for each individual item in the list. You need to specify that list item layout in a separate layout resource file, because it gets inflated by the adapter, separately from the RecyclerView.

Create a simple word item layout using a vertical LinearLayout with a TextView:

1. Right-click the app/res/layout folder and choose **New > Layout resource file**.
2. Name the file `wordlist_item.xml` and click **OK**.
3. In Text mode, change the LinearLayout that was created with the file to match with these attributes:

Attribute	Value
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:orientation	"vertical"
android:padding	"6dp"

4. Add a TextView for the word to the LinearLayout:

Attribute	Value
android:id	"@+id/word"
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:textSize	"24sp"
android:textStyle	"bold"

You can use styles to allow elements to share display attributes. An easy way to create a style is to extract the style of a UI element that you already created.

Extract the style information for the word text view:

5. Anywhere in the text view **Right-click > Refactor > Extract > Style**.
6. In the **Extract Android Style** dialog,
  - Name your style **word\_title**.
  - Leave all boxes checked.
  - Check the **Launch 'Use Style Where Possible'** box.
  - Click **OK**.
7. When prompted, apply the style to the **Whole Project**.
8. Find and examine the `word_title` style in `values/styles.xml`.
9. Go back to `wordlist_item.xml`. The text view now references the style instead of using individual styling properties.
10. Run your app. Since you have removed the default "Hello World" text view, you should see the "Word List" title and a blank view.

### Solution:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="6dp">

    <TextView
        android:id="@+id/word"
        style="@style/word_title" />

</LinearLayout>

```

### 3.3. Create an adapter with a view holder

Android uses adapters (from the [Adapter class](#)) to connect data with their views. There are many different kinds of adapters available. You can also write your own custom adapters. In this task you will create an adapter that associates your list of words with word list item views.

To connect data with views, the adapter needs to know about the views into which it will place the data. The adapter contains a view holder (from the [ViewHolder class](#)) that describes an item view and its place within the RecyclerView. The following diagram shows the relationship between data, adapter, view holder, and the RecyclerView.

In this task you will build an adapter with a view holder that bridges the gap between the data in your word list and the RecyclerView that displays it.

1. Right-click `java/com.android.example.wordlist` and select **New > Java Class**. (Not the (test) or (androidTest) directories.)
2. Name the class `WordListAdapter`.
3. Give `WordListAdapter` the following signature:

```

public class WordListAdapter extends RecyclerView.Adapter<WordListAdapter.WordView
Holder> {}

```

Note that `WordViewHolder` shows an error, because it has not been defined yet.

4. Click on the class declaration (i.e., “`WordListAdapter`”) and then click on the red light bulb on the left side of the pane. Choose **Implement methods**.

This brings up a dialog box that asks you to select the methods you want to implement. Select all three methods and click “OK”.

This creates empty placeholders for all the methods that you must implement. Note how onCreateViewHolder and onBindViewHolder both reference the view holder, which hasn't been implemented yet.

### 3.3.1 Create the view holder

1. Inside the WordListAdapter class, add a new WordViewHolder inner class with this signature:

```
class WordViewHolder extends RecyclerView.ViewHolder {}
```

2. You will see an error about a missing default constructor. You can see details about the errors by hovering your mouse cursor over the red-underlined source code or over any red horizontal line on the right margin of the open-files pane.
3. Add class variables to the WordViewHolder inner class for the text view and the adapter:

```
public final TextView wordItemView;  
final WordListAdapter mAdapter;
```

4. In the inner class WordViewHolder, add a constructor that initializes the view holder's text view from the XML resources and sets its adapter:

```
public WordViewHolder(View itemView, WordListAdapter adapter) {  
    super(itemView);  
    wordItemView = (TextView) itemView.findViewById(R.id.word);  
    this.mAdapter = adapter;  
}
```

5. Build and run your app. You should see the familiar blank view. Take note of the **E/RecyclerView: No adapter attached; skipping layout** warning in logcat.

### 3.3.2 Storing your data in the adapter

1. To hold your data in the adapter, create a private linked list of strings class variable in WordListAdapter and call it mWordList.

```
private final LinkedList<String> mWordList;
```

2. You can now fill in the getItemCount() method to return the size of `mWordList`.

```

@Override
public int getItemCount() {
    return mWordList.size();
}

```

Next, WordListAdapter needs a constructor that initializes the word list from that data.

To create a view for a list item, the WordListAdapter needs to inflate the XML for a list item. You use a [layout inflator](#) for that job. A LayoutInflator reads a layout XML description and converts it into the corresponding views.

3. Create a member variable for the inflater in WordListAdapter.

```
private LayoutInflater mInflater;
```

4. Implement the constructor for WordListAdapter. The constructor needs to have have a context parameter, and a linked list of words with the app's data. The method needs to instantiate a layout inflater for `mInflater` and set `mWordList`.

```

public WordListAdapter(Context context, LinkedList<String> wordList) {
    mInflater = LayoutInflater.from(context);
    this.mWordList = wordList;
}

```

5. Fill out the `onCreateViewHolder()` method with the code below.

The `onCreateViewHolder` method inflates the item layout, and returns a view holder with the layout and the adapter.

```

@Override
public WordViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View mItemView = mInflater.inflate(R.layout.wordlist_item, parent, false);
    return new WordViewHolder(mItemView, this);
}

```

6. Fill out the `onBindViewHolder` method with the code below.

The `onBindViewHolder` method connects your data to the view holder.

```

@Override
public void onBindViewHolder(WordViewHolder holder, int position) {
    String mCurrent = mWordList.get(position);
    holder.wordItemView.setText(mCurrent);
}

```

7. Build and run your app. You will still see the E/RecyclerView: No adapter attached warning. You will fix that in the next task.

## 3.4. Create the RecyclerView in the Main Activity

Now that you have an adapter with a view holder, you can finally create a RecyclerView and connect all the pieces to display your data.

1. Open MainActivity.java
2. Add member variables to MainActivity for the RecyclerView and the adapter.

```
private RecyclerView mRecyclerView;
private WordListAdapter mAdapter;
```

3. In the onCreate method of MainActivity, add the following code that creates the RecyclerView and connects it with an adapter and the data. Read the code comments!

Note you must insert this code after the mWordList initialization.

```
// Get a handle to the RecyclerView.
mRecyclerView = (RecyclerView) findViewById(R.id.recyclerview);
// Create an adapter and supply the data to be displayed.
mAdapter = new WordListAdapter(this, mWordList);
// Connect the adapter with the RecyclerView.
mRecyclerView.setAdapter(mAdapter);
// Give the RecyclerView a default layout manager.
mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
```

4. Build and run your code.

You should see your list of words displayed, and you can scroll the list.

## Task 4. Make the list interactive

Looking at lists of items is interesting, but it's a lot more fun and useful if your user can interact with them.

To see how the RecyclerView can respond to user input, you will programmatically attach a click handler to each item. When the item is tapped, the click handler is executed, and that item's text will change.

### 4.1. Make items respond to clicks

1. Open the WordListAdapter.java file.

2. Change the WordViewHolder class signature to implement View.OnClickListener.

```
class WordViewHolder extends RecyclerView.ViewHolder implements View.OnClickListener
```

3. Click on the class header and on the red light bulb to implement stubs for the required methods, which in this case is just the onClick() method.
4. Add the following code to the body of the onClick() method.

```
// Get the position of the item that was clicked.  
int mPosition = getLayoutPosition();  
// Use that to access the affected item in mWordList.  
String element = mWordList.get(mPosition);  
// Change the word in the mWordList.  
mWordList.set(mPosition, "Clicked! " + element);  
// Notify the adapter that the data has changed, so it gets the views updated.  
mAdapter.notifyDataSetChanged();
```

5. Connect the onClickListener with the view by adding this code to the WordViewHolder constructor (below the "this.mAdapter = adapter" line):

```
itemView.setOnClickListener(this);
```

6. Build and run your code. Click on items to see their text change.

### Solution code for WordListAdapter.java

### Solution code for MainActivity.java

## Task 5. Add a FAB to insert items

There are multiple ways in which you can add additional behaviors to the list and list items. One way is to use a floating action button (FAB). For example, in Gmail, the FAB is used to compose a new email. In this task you will implement a FAB to add an item to the word list.

**Why:** You have already seen that you can change the content of list items. The list of items that a RecyclerView displays can be modified dynamically-- it's not just a static list of items.

For this practical, you will generate a new word to insert into the list. For a more useful application, you would get data from your users. You will learn how to do this in a later lesson.

### 5.1. Add a Floating Action Button (FAB)

The FAB is a standard control from the Material Design specification and is part of the Android design support library. These UI controls have predefined properties. To create a FAB for your app, add the following code inside the coordinator layout of activity\_main.xml.

```
<android.support.design.widget.FloatingActionButton  
    android:id="@+id/fab"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="bottom|end"  
    android:layout_margin="16dp"  
    android:clickable="true"  
    android:src="@drawable/ic_add_24dp" />
```

Note the following:

- `@+id/fab`: It is customary to give the FAB the "fab" id.
- `android:layout_gravity="bottom|end"`: The FAB is commonly placed at the bottom and at the end of the reading/writing flow.
- `android:src="@drawable/ic_add_black_24dp"`: Is marked red by Android Studio because the resource is missing.

Android provides an icon library for standard Android icons. `ic_add_black_24dp` is one of the standard icons. You have to add it to your drawable resources to use it.

1. Right-click your drawable folder.
2. Select **New > Vector Asset**
3. Make sure Material Icon is selected.
4. Click the **Choose** button.
5. Scroll to find the + sign. The resource name is **ic\_add\_black\_24dp**.
6. Click **OK**.
7. Leave everything else unchecked and click **Next**.
8. Click **Finish**.
9. Build and run your app.

Note: Because this is a vector drawing, it is stored as an XML file. Vector drawings are automatically scaled, so you do not need to keep around a bitmap for each screen resolution. Learn more: [Android Vector Asset Studio](#).

## 5.2. Add behavior to the FAB

In this task you'll add an onClick listener to the FAB that does the following:

- Adds a word to the end of the list of words.
- Notifies the adapter that the data has changed.

- Scrolls to the inserted item.
- In MainActivity.java, at the end of the onCreate() method, add the following code:

```
// Add a floating action click handler for creating new entries.  
FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);  
fab.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View view) {  
        int wordListSize = mWordList.size();  
        // Add a new word to the wordList.  
        mWordList.addLast("Word " + wordListSize);  
        // Notify the adapter, that the data has changed.  
        mRecyclerView.getAdapter().notifyItemInserted(wordListSize);  
        // Scroll to the bottom.  
        mRecyclerView.smoothScrollToPosition(wordListSize);  
    }  
});
```

- Build and run your app. Test your app by doing the following:
  1. Scroll the list of words.
  2. Click on items.
  3. Add items by clicking on the FAB.
  4. What happens if you rotate the screen? You will learn in a later lesson how to preserve the state of an app when the screen is rotated.

### Solution code

## Coding challenge

Creating a click listener for each item in the list is easy, but can hurt the performance of your app if you have a lot of data. Research how you could implement this more efficiently. This is an advanced challenge. Start by thinking about it conceptually, and then search for an implementation example.

**Note:** All coding challenges are optional and not prerequisite for the material in the next chapter.

## Summary

- RecyclerView is a resource-efficient way to display a scrollable list of items.
- To use RecyclerView, you associate the data to the Adapter/ViewHolder that you create and to the layout manager of your choice.
- Click listeners can be created to detect mouse clicks in a RecyclerView.

- Android support libraries contain backward-compatible versions of the Android framework.
- Android support libraries contain a range of useful utilities for your apps.
- Build dependencies are added to the build.gradle (Module app) file.
- Layouts can be specified as a resource file.
- A LayoutInflater reads a layout resource file and creates the View objects from that file.
- A Floating Action Button (FAB) can dynamically modify the items in a RecyclerView.

## Resources

### Developer Documentation:

- [Creating Lists and Cards](#)
- [RecyclerView Animations and Behind the Scenes \(Android Dev Summit 2015\)](#)

# 5.1 P: Drawables, Styles, and Themes

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1: Create the Scorekeeper Project](#)
- [Task 2: Create a Drawable resource](#)
- [Task 3: Apply styles to your views](#)
- [Task 4: Update the theme from the menu bar](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

In this chapter you will learn how to reduce your code, increase the readability and ease of maintenance by applying common styles to your views, use drawable resources, and apply themes to your application.

## What you should already KNOW

From the previous chapters you should be familiar with:

- How to create and run apps in Android Studio.
- How to create and edit UI elements using the Layout Editor, XML, and code.
- How to add onClick functionality to a button.
- How to update views at runtime.
- How to add menu items with onClick functionality.
- How to pass data between activities using Intents.
- Basic concepts of the Activity lifecycle.

## What you will LEARN

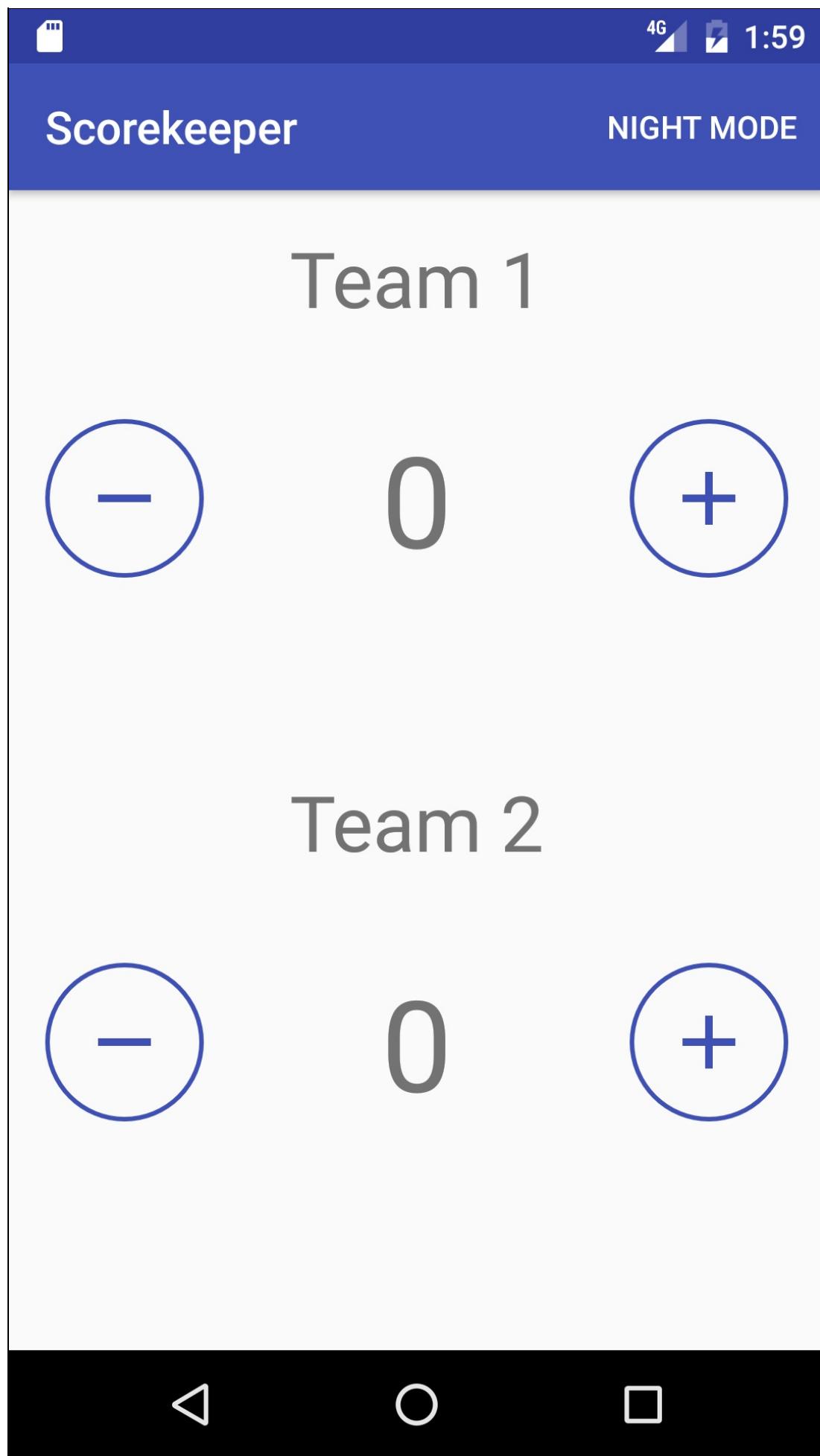
- How to define a style.
- How to apply a style to a view.
- How to apply a theme to an activity or application both in xml and programmatically
- How to use drawable resources.

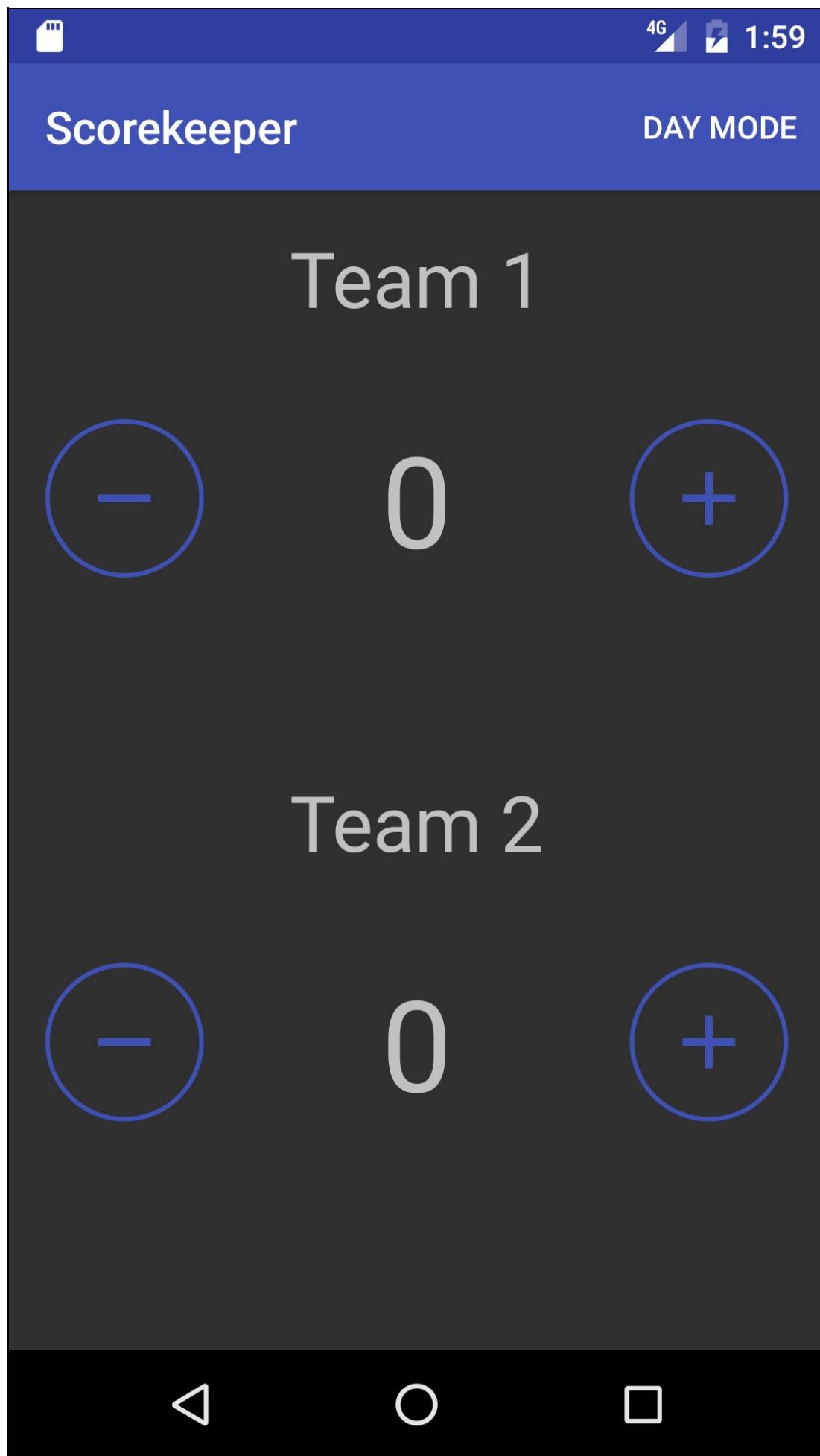
## What you will DO

- Create a new Android app and add buttons and TextViews to the Layout.
- Create drawable resources in XML and use them as backgrounds for your buttons.
- Apply styles to the UI elements of the application.
- Add a menu item that changes the theme of the application to a low contrast “night mode.”

## App Overview

The “Scorekeeper” application consists of two sets of buttons and two TextViews used to keep track of the score for any point-based game with two players.





# Task 1: Create The Scorekeeper App

In this section, you will create your Android Studio project, modify the layout, and add onClick functionality to its buttons.

## 1.1 Create the “Scorekeeper” Project

1. Start Android Studio and create a new Android Studio Project.
  - Name your project “Scorekeeper”.
  - Accept the defaults for the Company Domain and Project location.
2. Accept the default Minimum SDK.
3. Choose the **Empty Activity** template.
4. Accept the default name for the activity, make sure **Generate Layout File** is checked and click **Finish**.

## 1.2 Create the layout for the main activity

### Define the root view:

1. Open the layout file for the main activity.
2. Delete the TextView that says "Hello World."
3. Change the rootview to a LinearLayout and add the following attributes (without removing the existing attributes):

Attribute	Value
android:orientation	"vertical"
android:weightSum	"2"

### Define the score containers:

1. Inside the LinearLayout, add two RelativeLayout view groups (one to contain the score for each team) with the following attributes:

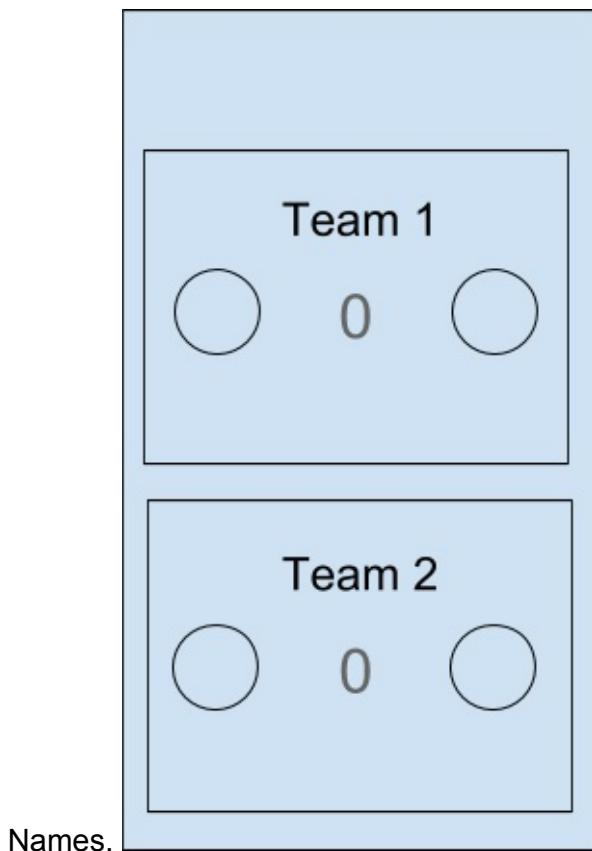
Attribute	Value
android:layout_width	"match_parent"
android:layout_height	"0dp"
android:layout_weight	"1"

You may be surprised to see that the layout\_height attribute is set to 0dp in these views. This is because we are using the “layout\_weight” and “weightSum” attributes to determine how much space these views take up in the parent layout. See the

[LinearLayout Documentation](#) for more information.

## Add views to your UI

1. Add two [ImageButton views](#) (one for increasing the score and one for decreasing the score) and a [TextView](#) for displaying the score in between the buttons to each [RelativeLayout](#).
2. Add `android:id` attributes to the score [TextViews](#) and all of the [ImageButtons](#).
3. Add one more [TextView](#) to each [RelativeLayout](#) above the score to represent the Team



## Add vector assets

1. Select **File > New > Vector Asset** to open the Vector Asset Studio.
2. Click on the icon to change it and select the **Content** category.
3. Choose the plus icon and click **OK**.
4. Rename the resource file “ic\_plus” and check the **Override** checkbox next to size options.
5. Change the size of the icon to 40dp x 40dp.
6. Click **Next** and then **Finish**.
7. Repeat this process to add a “minus” icon and name the file “ic\_minus”.

## Add attributes to your views

1. Change the score [TextViews](#) to read “0” and the team [Textviews](#) to read “Team 1” and “Team 2”.

2. Add the following attributes to your left ImageButtons:

```
    android:src="@drawable/ic_minus"  
    android:contentDescription="Minus Button"
```

3. Add the following attributes to your right ImageButtons:

```
    android:src="@drawable/ic_plus"  
    android:contentDescription="Plus Button"
```

4. Extract all of your string resources. This process removes all of your strings from the Java code and puts them in a single file: the string.xml file. This allows for your app to be easily localized into different languages. To learn how to accomplish this, see the [Appendix](#).



**Solution Code:**

**Note:** Your code may be a little different as there are multiple ways to achieve the same 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="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:weightSum="2"
    tools:context="com.example.android.scorekeeper.MainActivity">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentTop="true"
            android:layout_centerHorizontal="true"
            android:text="@string/team_1"/>

        <ImageButton
            android:id="@+id/decreaseTeam1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentLeft="true"
            android:layout_alignParentStart="true"
            android:layout_centerVertical="true"
            android:contentDescription="@string/minus_button"
            android:src="@drawable/ic_minus" />

        <TextView
            android:id="@+id/score_1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerHorizontal="true"
            android:layout_centerVertical="true"
            android:text="@string/initial_count"/>

        <ImageButton
            android:id="@+id/increaseTeam1"
            android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:layout_alignParentEnd="true"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:contentDescription="@string/plus_button"
        android:src="@drawable/ic_plus"/>
    </RelativeLayout>

<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:text="@string/team_2"/>

    <ImageButton
        android:id="@+id/decreaseTeam2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_centerVertical="true"
        android:contentDescription="@string/minus_button"
        android:src="@drawable/ic_minus"/>

    <TextView
        android:id="@+id/score_2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/initial_count"/>

    <ImageButton
        android:id="@+id/increaseTeam2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentEnd="true"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:contentDescription="@string/plus_button"
        android:src="@drawable/ic_plus"/>

```

## 1.3 Initialize your TextViews and score count variables

In order to keep track of the scores, you will need two things: integer variables so you can keep track of the scores, and a reference to your score TextViews in MainActivity so you can update the scores in them.

1. In the onCreate() method of MainActivity, find your score TextViews by id and assign them to member variables.
2. Create two integer member variables, representing the score of each team, and initialize them to 0.

## 1.4 Implement the onClick functionality for your buttons

1. In your MainActivity implement two onClick methods: increaseScore() and decreaseScore().  
**Note:** onClick methods all have the same signature - they return void and take a View as an argument.
2. The left buttons should decrement the score TextView, while the right ones should increment it.

### Solution Code:

**Note:** You must also add the android:onClick attribute to every button in the activity\_main.xml file. You can get which button was clicked by calling view.getId() in the onClick methods.

```
/**  
 * Method that handles the onClick of both the decrement buttons  
 * @param view The button view that was clicked  
 */  
public void decreaseScore(View view) {  
    //Get the ID of the button that was clicked  
    int viewID = view.getId();  
    switch (viewID){  
        //If it was on Team 1  
        case R.id.decreaseTeam1:  
            //Decrement the score and update the TextView  
            mScore1--;  
            mScoreText1.setText(String.valueOf(mScore1));  
            break;  
        //If it was Team 2  
        case R.id.decreaseTeam2:  
            //Decrement the score and update the TextView  
            mScore2--;  
            mScoreText2.setText(String.valueOf(mScore2));  
    }  
}  
  
/**  
 * Method that handles the onClick of both the increment buttons  
 * @param view The button view that was clicked  
 */  
public void increaseScore(View view) {  
    //Get the ID of the button that was clicked  
    int viewID = view.getId();  
    switch (viewID){  
        //If it was on Team 1  
        case R.id.increaseTeam1:  
            //Increment the score and update the TextView  
            mScore1++;  
            mScoreText1.setText(String.valueOf(mScore1));  
            break;  
        //If it was Team 2  
        case R.id.increaseTeam2:  
            //Increment the score and update the TextView  
            mScore2++;  
            mScoreText2.setText(String.valueOf(mScore2));  
    }  
}
```

## Task 2: Create a Drawable resource

You now have a functioning scorekeeper application! However, the layout is dull and does not communicate the function of the buttons. In order to make it more clear, the standard grey background of the buttons can be changed.

In Android, graphics are often handled by a resource called a **Drawable**. In the following exercise you will learn how to create a certain type of drawable called a **ShapeDrawable**, and apply it to your buttons as a background.

For more information on Drawables, see [Drawable Resource Documentation](#).

## 2.1 Create a Shape Drawable

A **ShapeDrawable** is a primitive geometric shape defined in an xml file by a number of attributes including color, shape, padding and more.

1. Right click on the **drawable** folder in your resources directory.
2. Choose **New > Drawable resource file**.
3. Name the file “button\_background” and click **OK**.
4. Remove all of the code except:

```
<?xml version="1.0" encoding="utf-8"?>
```

5. Add the following code which defines an oval shape with an outline:

```
<shape  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:shape="oval">  
    <stroke  
        android:width="2dp"  
        android:color="@color/colorPrimary"/>  
</shape>
```

## 2.2 Apply a drawable as a background

1. Open the layout file for your main activity.
2. For all of the buttons, add the drawable as the background:

```
    android:background="@drawable/button_background"
```

3. The size of the buttons needs to be such that it renders properly on all devices. Change the “layout\_height” and “layout\_width” attributes for each button to 70dp, which is a good size on most devices:

```
    android:layout_width="70dp"  
    android:layout_height="70dp"
```

4. Extract the dimension resource so you can access it one location. For information on how to do this, see the [Appendix](#).

5. Run your app.

## Task 3: Style your views

As you continue to add views and attributes to your layout, your code will start to become crowded and repetitive as you apply the same attributes to every element with a similar appearance. A style can specify common properties such as height, padding, font color, font size, background color, et al. Attributes that are layout-oriented such as height, width and relative location should remain in the layout resource file.

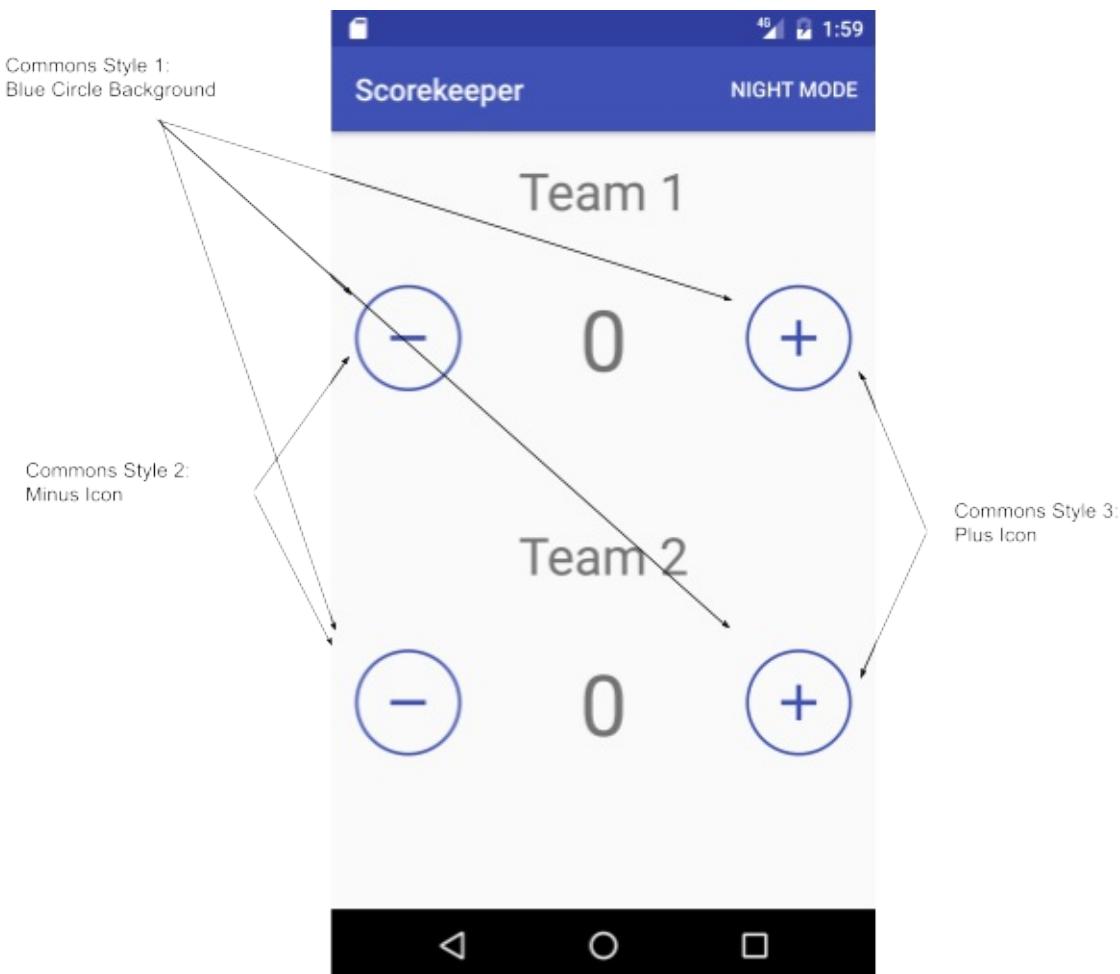
In the following exercise, you will learn how to create styles that can be applied to multiple views and layouts, allowing common attributes to be updated simultaneously from one location.

**Note:** styles are meant for attributes that modify the look of the view. Layout parameters such as height, weight and relative location should stay in the layout file.

### 3.1 Create button styles

In Android, styles follow the principles of inheritance. All attributes not explicitly defined in your style are inherited from the style set of the hierarchical parent, defined using an optional parameter. For example, all four buttons in this example share a common background drawable but with different icons for plus and minus. Furthermore, the two increment buttons share the same icon, as do the two decrement buttons. You can therefore create 3 distinct styles:

1. A score button style for all of the buttons, which includes the default properties of an `ImageButton` widget and also the drawable background.
2. A plus button style for the increment buttons, which inherits the attributes of the previous style and also includes the plus icon.
3. A minus button style for the decrement buttons, again inheriting from the score button style and also includes the minus icon.



Do the following:

1. In your resources directory, locate and open the “values/styles.xml” file.

This is where all of your style code will be located. The “AppTheme” style is always automatically added, and you can see that it extends from “Theme.AppCompat.Light.DarkActionBar”.

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
```

Note the “parent” attribute is how you specify your parent style using XML.

The name attribute, in this case "AppTheme" defines the name of the style which you will use to apply it to the application, an activity or even a single view. The parent attribute, in this case “Theme.AppCompat.Light.DarkActionBar”, defines the rest of the attributes which aren't explicitly defined in your style. In this case it is the default theme, with a light background and a dark action bar. A theme is a style that is applied to an entire activity or application, instead of a single view.

2. In between the `<resources>` tags, add another style with the following attributes to create a common style for all buttons:

- Set the parent style to “Widget.AppCompat.Button” to retain the default attributes of a button.
- Add an attribute that changes the background of the drawable to the one you created in the previous task.

```
<style name="ScoreButtons" parent="Widget.AppCompat.Button">
    <item name="android:background">@drawable/button_background</item>
</style>
```

3. Create the style for the plus buttons by extending the “ScoreButtons” style:

```
<style name="PlusButtons" parent="ScoreButtons">
    <item name="android:src">@drawable/ic_plus</item>
    <item name="android:contentDescription">@string/plus_button</item>
</style>
```

4. Create the style for the minus buttons:

```
<style name="MinusButtons" parent="ScoreButtons">
    <item name="android:src">@drawable/ic_minus</item>
    <item name="android:contentDescription">@string/minus_button</item>
</style>
```

5. In the layout file for the main activity. Remove all of the attributes that you defined in the styles for each button and add the appropriate style:

```
style="@style/MinusButtons"
style="@style/PlusButtons"
```

**Note:** the style attribute does not use the “android:” namespace.

## 3.2 Create TextView styles

The team name and score display TextViews can also be styled since they have common colors and fonts. Do the following:

1. Add the following attribute to all TextViews:

```
android:textAppearance="@style/TextAppearance.AppCompat.Headline"
```

2. Right-click anywhere in the first score TextView attributes and choose **Refactor > Extract > Style...**
3. Name the style “ScoreText” and check the textAppearance box (the attribute you just added) as well as the **Launch ‘Use Styles Where Possible’ refactoring after the style is extracted** (using the checkboxes). This will scan the layout file for views with

the same attributes and apply the style for you. Make sure that you do not extract the attributes that are related to the layout, which should remain in the layout file.

4. Choose **OK**.
5. Make sure the scope is set to the activity\_main.xml layout file and click **OK**.
6. A pane at the bottom of Android Studio will open up if the same style is found in other views, select **Do Refactor** to apply the new style to the views with the same attributes.
7. Run your app. There should be no change except that all of your styling code is now in your resources file and your layout file contains considerably less clutter.

## Solution Code:

### styles.xml

```
<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>

    <style name="ScoreButtons" parent="AppTheme">
        <item name="android:background">@drawable/button_background</item>
    </style>

    <style name="PlusButtons" parent="ScoreButtons">
        <item name="android:src">@drawable/ic_plus</item>
        <item name="android:contentDescription">@string/plus_button</item>
    </style>

    <style name="MinusButtons" parent="ScoreButtons">
        <item name="android:src">@drawable/ic_minus</item>
        <item name="android:contentDescription">@string/minus_button</item>
    </style>

    <style name="ScoreText">
        <item name="android:textAppearance">@style/TextAppearance.AppCompat.Headline</item>
    </style>
</resources>
```

### activity\_main.xml

```
<?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="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:weightSum="2"
    tools:context="com.example.android.scorekeeper.MainActivity">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1">

        <TextView
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:layout_alignParentTop="true"
            android:layout_centerHorizontal="true"
            android:text="@string/team_1"
            style="@style/ScoreText" />

        <ImageButton
            android:id="@+id/decreaseTeam1"
            android:layout_height="@dimen/button_size"
            android:layout_width="@dimen/button_size"
            android:layout_alignParentLeft="true"
            android:layout_alignParentStart="true"
            android:layout_centerVertical="true"
            android:onClick="decreaseScore"
            style="@style/MinusButtons" />

        <TextView
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:layout_centerVertical="true"
            android:layout_centerHorizontal="true"
            android:id="@+id/score_1"
            android:text="@string/initial_count"
            style="@style/ScoreText" />

        <ImageButton
            android:id="@+id/increaseTeam1"
            android:layout_height="@dimen/button_size"
            android:layout_width="@dimen/button_size"
            android:layout_alignParentRight="true"
            android:layout_alignParentEnd="true"
            android:layout_centerVertical="true"
            android:onClick="increaseScore"
            style="@style/PlusButtons" />

    </RelativeLayout>

    <RelativeLayout
```

```
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1">

    <TextView
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:text="@string/team_2"
        style="@style/ScoreText" />

    <ImageButton
        android:id="@+id/decreaseTeam2"
        android:layout_height="@dimen/button_size"
        android:layout_width="@dimen/button_size"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_centerVertical="true"
        android:onClick="decreaseScore"
        style="@style/MinusButtons"/>

    <TextView
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true"
        android:id="@+id/score_2"
        android:text="@string/initial_count"
        style="@style/ScoreText" />

    <ImageButton
        android:id="@+id/increaseTeam2"
        android:layout_height="@dimen/button_size"
        android:layout_width="@dimen/button_size"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:layout_centerVertical="true"
        android:onClick="increaseScore"
        style="@style/PlusButtons"/>
</RelativeLayout>
</LinearLayout>
```

### 3.3 Updating the styles

The power of using styles becomes apparent when you want to make changes to several elements of the same style. Make the text bigger, bolder and brighter, and change the icons to the color of the button backgrounds.

Make the following changes in your styles.xml file:

1. Add or modify each of the following attributes in the specified style block:

Attribute	Style Block
@color/colorPrimary	ScoreButtons
@style/TextAppearance.AppCompat.Display3	ScoreText

**Note:** The colorPrimary value is automatically generated by Android Studio when you create the project and can be found in the values/colors.xml file. The TextAppearance.AppCompat.Display3 attribute is a predefined text style supplied by android.

2. Create a new style called "TeamText" with the following attribute:

```
<item name="android:textAppearance">@style/TextAppearance.AppCompat.Display1</item>
```

3. In activity\_main.xml, change the style attribute of the team name TextViews to the newly created TeamText style.
4. Run your app. With just these adjustments to the style.xml file, all of the views updated to reflect the changes.

## Task 4: Themes and Final Touches

You've seen that views with similar characteristics can be styled together in the "styles.xml" file. But what if you want to define styles for an entire activity, or even application? It is possible to accomplish this by using "Themes". To set a theme for an Activity or set of Activities, you need to modify the AndroidManifest.xml file. In this task, you will add the "night mode" theme to your app which will allow them to use a low contrast version of your app that is easier on the eyes at night time, as well as a few polishing touches to the User Interface.

### 4.1 Explore themes

Do the following:

1. In the Android manifest file, In the `<application>` tag, change the `android:theme` attribute to:

```
    android:theme="@style/Theme.AppCompat.Light.NoActionBar"
```

This is predefined theme that removes the action bar from your activity.

2. Run your app. The toolbar disappears!
3. Change the theme of the application back to AppTheme, which is a child of the

Theme.AppCompat.Light.DarkActionBar theme as can be seen in styles.xml.

That's all there is to it. To apply a theme to an activity instead of the entire application, place the theme attribute in the activity tag instead of the application tag. For more information on Themes and Styles, see the [Style and Theme Guide](#).

## 4.2 Add theme button to the menu

One use for setting a theme for your application is to provide an alternate visual experience for browsing at night. In such conditions, it is often better to have a low contrast, dark layout. Fortunately, the Android framework provides a theme that is designed exactly for this: The DayNight theme.

This theme has several built in options that allow you to control the colors in your app programmatically: either setting to change automatically by time, or by user command. In this exercise you will add a menu button that will toggle the application between the regular theme and a “night-mode” theme.

1. Right click on the “res” directory and choose **New > Android resource file**.
2. Name the file “main\_menu”, change the **Resource Type** to **Menu**, and click **OK**.
3. Add a menu item with the following attributes:

```
<item  
    android:id="@+id/night_mode"  
    android:title="@string/night_mode"  
    app:showAsAction="withText|ifRoom"/>
```

**Note:** The “app:” namespace used above is probably not yet defined in your xml code and therefore it may appear red. To add the namespace definition, place your text cursor on the red word “app:” and press **Alt + Enter** (Option + Enter on a Mac) and choose **Create namespace declaration**.

4. Navigate to “strings.xml” and create two string resources:

```
<string name="night_mode">Night Mode</string>  
<string name="day_mode">Day Mode</string>
```

5. In your main activity Java file, press **Ctrl - O** to open the Override Method menu, select the **onCreateOptionsMenu** method located under the “android.app.Activity” category and click **OK**.
6. Inflate the menu you just created within the `onCreateOptionsMenu()` method:

```
getMenuInflater().inflate(R.menu.main_menu, menu);
```

## 4.3 Change the theme from the menu

The DayNight theme uses the AppCompatDelegate class to set the night mode options in your activity. To learn more about this theme, visit this [blog post](#).

1. In your styles.xml file, modify the parent of AppTheme to "Theme.AppCompat.DayNight.DarkActionBar".
2. Override the onOptionsItemSelected() method in MainActivity, and check which menu item was clicked:

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    //Check if the correct item was clicked  
    if(item.getItemId()==R.id.night_mode){}  
}
```

3. In response to a click on the menu button, check the current night mode setting by calling `AppCompatDelegate.getDefaultNightMode()` .
4. If the night mode is enabled, change it to the disabled state:

```
//Get the night mode state of the app  
int nightMode = AppCompatDelegate.getDefaultNightMode();  
//Set the theme mode for the restarted activity  
if(nightMode == AppCompatDelegate.MODE_NIGHT_YES) {  
    AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_NO);  
}
```

5. Otherwise, enable it:

```
else {  
    AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_YES);  
}
```

6. The theme can only change while the activity is being created, so call `recreate()` for the theme change to take effect.
7. Your onOptionsItemSelected() method should return true, since the item click was handled.
8. Run your app. The “Night Mode” menu item should now toggle the theme of your activity.

You may notice that the label for your menu item always reads “Night Mode”, which may be confusing to your user if the app is already in the dark theme.

9. Add the following code in the **onCreateOptionsMenu** method:

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    //Inflate the menu from XML
    getMenuInflater().inflate(R.menu.main_menu, menu);

    //Change the label of the menu based on the state of the app
    int nightMode = AppCompatDelegate.getDefaultNightMode();
    if(nightMode == AppCompatDelegate.MODE_NIGHT_YES){
        menu.findItem(R.id.night_mode).setTitle(R.string.day_mode);
    } else{
        menu.findItem(R.id.night_mode).setTitle(R.string.night_mode);
    }
    return true;
}

```

- Run your app. The menu button label now changes with the theme.

## 4.4 SaveInstanceState

You learned in previous lessons that you must be prepared for your activity to be destroyed and recreated at unexpected times, for example when your screen is rotated. In this application, the TextViews containing the scores are reset to the initial value of 0 when the device is rotated. To fix this, Do the following:

- Override the `onSaveInstanceState()` method in `MainActivity` to preserve the values of the two score TextViews.
- In the `onCreate()` method of `MainActivity.java`, check if there is a `savedInstanceState`. If there is, restore the scores to the TextViews.

That's it! Congratulations, you now have a styled Scorekeeper Application.

### Solution Code:

## Coding challenge

**Note:** All coding challenges are optional and not prerequisite for the material in the next chapter.

Right now, your buttons do not behave intuitively because they do not change their appearance when they are pressed. Android has another type of drawable called **StateListDrawable** which allows for a different graphic to be used depending on the state of the object.

For this challenge problem, create a drawable resource that changes the background of the button to the same color as the border when the state of the button is “pressed”. You should also set the color of the text inside the buttons to a selector that makes it white when the button is “pressed”.

## Summary

- Learned to use Drawables to enhance the look of an application
- A [ShapeDrawable](#) is a primitive geometric shape defined in an xml file by a number of attributes including color, shape, padding and more.
- A Style can specify common properties such as height, padding, font color, font size, background color, et al.
- Using styles can reduce the amount of common code for your UI components
- Style should not include layout-related information
- Styles can be applied to View, Activities or Applications
- Style applied to Activities or Applications must be defined in the Android Manifest XML file
- Styles can be inherited by identifying the parent style using XML
- When you apply a style to a collection of Views in an Activity or in your entire application, that is known as a Theme
- `Android:theme` is the attribute you need to set a style on a collection of Views in an Activity or Application
- The Android platform supplies a large collection of styles and themes

---

## Resources

### Developer Documentation:

- [LinearLayout Guide](#)
- [Drawable Resource Guide](#)
- [Styles and Themes Guide](#)
- [DayNight Theme Guide](#)

### Videos

- [Udacity - Themes and Styles](#)



## 5.2 P: Material Design: Cards and the FAB

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1: Download the starter code](#)
- [Task 2: Add a CardView and Images](#)
- [Task 3: RecyclerView Animation and Detail View](#)
- [Task 4: The FAB and Material Color Palette](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

This chapter introduces concepts from Google's [Material Design](#) guidelines, a series of best practices for creating visually appealing and intuitive applications. You will learn how to add and use the CardView and Floating Action Button Widgets, efficiently use images, as well as employ best design practices to make your user's experience delightful.

### What you should already KNOW

From the previous chapters you should be familiar with:

- Creating and run apps in Android Studio.
- Creating and editing UI elements using the Layout Editor, XML, and programmatically.
- Using a RecyclerView to display a list.

### What you will LEARN

- Recommended use of material widgets (Floating Action Button, CardView).
- How to efficiently use images in your app.
- Recommended best practices for designing intuitive layouts using bold colors.

### What you will DO

- Modify an application to follow Material Design guidelines.
- Add images and styling to a RecyclerView list.
- Implement a ItemTouchHelper to add Drag and Drop functionality to your app.

## App Overview

The Material Me! app is a mock sports news application with very poor design implementation. You will be fixing it up to meet the design guidelines and creating a delightful user experience!

The image shows a mobile application interface titled "Material Me!" with a green header bar. The screen displays three news cards, each with a large title, a preview text, and a corresponding image.

- Badminton News:** The card has a white background. The title "Badminton" is in bold black font. The preview text reads "Here is some Badminton news!". Below the text is a small image of a yellow shuttlecock in flight.
- Basketball News:** The card has a dark teal background. The title "Basketball" is in white font. The preview text reads "Here is some Basketball news!". Below the text is a large image of an orange basketball hitting a backboard.
- Bowling News:** The card has a purple background. The title "Bowling" is in white font. The preview text reads "Here is some Bowling news!". Below the text is a large image of a yellow bowling ball hitting pins.

At the bottom of the screen, there is a black navigation bar with three white icons: a triangle pointing left, a circle, and a square.

# Task 1: Download the starter code

The complete starter app project for this practical is available at [MaterialMe-start.zip](#). In this task you will load the project into Android Studio and explore some of the app's key features.

## 1.1 Open and Run the Material Me Project

1. Download [MaterialMe-start.zip](#) and unzip the file.
2. Open the app in Android Studio.
3. Build and run the app.

The current layout and style of the app makes it nearly unusable: there is no separation between each row of data and there is no imagery or color to engage the user.

## 1.2 Explore the App

Before making modifications to the app, explore its current structure. It contains the following elements:

### 1. **Sport.java**

This class represents the data model for each row of data in the RecyclerView. Right now it contains a field for the title of the sport and a field for some information about the sport.

### 2. **SportsAdapter.java**

This is the adapter for the RecyclerView. It uses an ArrayList of Sport objects as its data and populates each row with this data.

### 3. **MainActivity.java**

The MainActivity initializes the recyclerview and adapter, and creates the data from resource files.

### 4. **strings.xml**

This resource file contains all of the data for the application, including the titles and information for each sport.

### 5. **list\_item.xml**

This layout file defines each row of the RecyclerView. It consists of three textviews, one for each piece of data (the title and the info for each sport) and one used as a label.

## Task 2: Add a CardView and Images

One of the fundamental principles of Material Design is the use of bold imagery to enhance the user experience. Adding images to the RecyclerView list items would be a good start to creating a dynamic and captivating user experience.

When presenting information that has mixed media (like images and text), the guidelines recommend using a [CardView](#), which is a FrameLayout with some extra features (such as elevation and rounded corners) that give it a consistent look and feel across many different applications and platforms.

In this section, you will be moving each list item into a CardView and adding an Image to make the app comply with Material guidelines.

### 2.1 Add the CardView

CardView is not included in the default android SDK and must be added as a build.gradle dependency. Do the following:

1. In your app level build.gradle file, add the following line to the dependencies block:

```
compile 'com.android.support:cardview-v7:24.1.1'
```

**Note:** The version of the support library may have changed since the writing of this practical. Update it to its most current version and sync your gradle files.

2. In the list\_item.xml file, surround the root LinearLayout with a CardView with the following attributes:

Attribute	Value
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:layout_margin	"8dp"

3. Run the app. Now each row item is contained inside a CardView, which is elevated above the bottom layer and casts a shadow.

### 2.2 Download the images

The CardView is not intended to be used exclusively with plain text: it is best for displaying a mixture of content. This is a good opportunity to make this sports information app more exciting by adding banner images to every row!

Using images is resource intensive: the Android framework has to load the entire image into memory at full resolution, even if it is displayed in a small thumbnail in the application.

In this section you will learn how to use large images efficiently, without draining your resources or even crashing your app due to 'Out of Memory' exceptions.

Do the following:

1. Download the banner images [here](#).
2. Copy these files into the drawable directory of your app.

You will need an array with the path to each image so that you can include it in the Sports java object. Do the following:

3. In your string.xml file, define an array that contains all of the paths to the drawables as items. Be sure to that they are in the same order as the sports titles array:

```
<array name="sports_images">
    <item>@drawable/img_baseball</item>
    <item>@drawable/img_badminton</item>
    <item>@drawable/img_basketball</item>
    <item>@drawable/img_bowling</item>
    <item>@drawable/img_cycling</item>
    <item>@drawable/img_golf</item>
    <item>@drawable/img_running</item>
    <item>@drawable/img_soccer</item>
    <item>@drawable/img_swimming</item>
    <item>@drawable/img_tabletennis</item>
    <item>@drawable/img_tennis</item>
</array>
```

## 2.3 Modify the Sport object

The Sport.java object will need to include the drawable resource that corresponds to the sport. Do the following:

1. Add an integer member variable to the Sport object that will contain the drawable resource:

```
private final int mImageResource;
```

2. Modify the constructor so that it takes an integer as a parameter and assigns it to the member variable:

```
public Sport(String title, String info, int imageResource) {
    this.mTitle = title;
    this.mInfo = info;
    this.mImageResource = imageResource;
}
```

3. Create a getter for the resource integer:

```
public int getmImageResource() {
    return mImageResource;
}
```

## 2.4 Fix the initializeData() method

In MainActivity, the initializeData() method is now broken, because the constructor for the Sport object demands the image resource as the third parameter.

To obtain the image resources as an array that you can index in the same loop as the sports title and information, you will use the [TypedArray](#) class:

1. Get the TypedArray of resources id's by calling getResources().obtainTypedArray(), passing in the name of the array you defined in strings.xml:

```
TypedArray sportsImageResources = getResources().obtainTypedArray(R.array.sports_images);
```

2. Fix the code in the loop that creates the Sports objects, adding the appropriate drawable resource id as the third parameter by calling getResourceId() on the TypedArray:

```
for(int i=0;i<sportsList.length;i++){
    mSportsData.add(new Sport(sportsList[i],sportsInfo[i], sportsImageResources.getResourceId(i, 0)));
}
```

3. Recycle the typed array once you have created the Sport data:

```
sportsImageResources.recycle();
```

## 2.5 Add an ImageView to the list items

1. Change the LinearLayout inside the list\_item.xml file to a RelativeLayout, and delete the

- orientation attribute.
- Add an ImageView with the following attributes:

Attribute	Value
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:id	"@+id/sportsImage"
android:adjustViewBounds	"true"

The `adjustViewBounds` attribute makes the ImageView adjust its boundaries to preserve the aspect ratio of the image. Add the following attributes to the existing TextViews:

TextView id: title	
Attribute	Value
android:layout_alignBottom	"@+id/sportsImage"
android:textColor	"@android:color/white"
TextView id: newsTitle	
Attribute	Value
android:layout_below	"@+id/sportsImage"
android:textColor	"?android:textColorSecondary"
TextView id: subTitle	
android:layout_below	"@+id/newsTitle"

## 2.6 Load the images using Glide

After downloading the images and setting up the ImageView, the next logical step is to modify the SportsAdapter to load an image into the ImageView in `onBindViewHolder()`. If you take this approach, you will find that your app crashes due to "Out of Memory" errors. In fact, the Android framework has to load the image at full resolution into memory each time, no matter what the display size of the ImageView is. There are a number of ways to reduce the memory consumption when loading images, which you can learn about in [this guide](#), but one of the easiest approaches is to use an Image Loading Library like [Glide](#), which you will implement in this step. Do the following:

- In your app level build.gradle file, add the following dependency for Glide:

```
compile 'com.github.bumptech.glide:glide:3.5.2'
```

2. In the SportsAdapter class, add a variable in the ViewHolder class for the ImageView you created, and initialize it in the ViewHolder constructor:

```
mSportsImage = (ImageView) itemView.findViewById(R.id.sportsImage);
```

3. Add the following line of code to onBindViewHolder to get the image resource from the Sport object and load it into the ImageView using Glide:

```
Glide.with(mContext).load(currentSport.getImageResource()).into(holder.mSportsImage);
```

That's all takes to load an image with Glide. It also has several additional features that let you resize, transform and load images in a variety of ways. Head over to the [Glide github page](#) to learn more.

Run the app, your list items should now have bold graphics that make the user experience dynamic and exciting!

## 2.7 Landscape orientation and GridLayoutManager

After running the application turn the phone sideways and observe how the list items look in landscape orientation. The images now take up a majority of the card and the space is not used efficiently. You will solve this issue by changing the RecyclerView LayoutManager to be a GridLayoutManager when the phone is in the landscape orientation, moving the cards into two columns. Do the following:

1. In the onCreate() method of MainActivity, check the orientation of the device by calling getResources().getConfiguration().orientation.
2. If the phone is in the portrait configuration, use the LinearLayoutManager. Otherwise, use a GridLayoutManager with a column span of 2:

```
if(getResources().getConfiguration().orientation == Configuration.ORIENTATION_PORTRAIT) {  
    mRecyclerView.setLayoutManager(new LinearLayoutManager(this));  
} else {  
    m.RecyclerView.setLayoutManager(new GridLayoutManager(this, 2));  
}
```

3. Run the app and rotate the device. With this change the landscape orientation provides as much information as the portrait orientation.

# Task 3: RecyclerView Animation and Detail View

Using Cards creates user expectations: in the real world cards can be moved around and thrown out. In Material Design, a developer should seek to use these expectations to provide intuitive actions to the user. With a CardView, the real world analogy and existing Material apps set up the following expectations:

- A card can be dismissed, usually by swiping it away.
- A list of cards can be reordered by holding down and dragging the cards.
- Tapping on card will provide further details.

You will now implement these behaviours in your app.

## 3.1 Implement swipe to dismiss

The Android SDK includes a class called `ItemTouchHelper` that is used to define what happens to RecyclerView list items when the user performs various touch actions, such as swipe, or drag and drop. The level of control that you require depends on the use case, but some of the common use cases are already implemented in a set of callbacks called `ItemTouchHelper.SimpleCallback`, which lets you define the supported directions of swiping and moving list items, and implement the desired behavior. Do the following:

1. In the `onCreate()` method of `MainActivity`, create a new `ItemTouchHelper` object. For its argument, create a new instance of `ItemTouchHelper.SimpleCallback` and press **Enter** to make Android Studio fill in the required methods: `onMove()` and `onSwiped()`.
2. The `SimpleCallback` constructor will be underlined in red because you have not yet provided the required parameters: the direction that you plan to support for moving and swiping list items, respectively.

Because we are only implementing swipe to dismiss at the moment, you should pass in `0` for the supported move directions and `ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT` for the supported swipe directions:

```
ItemTouchHelper helper = new ItemTouchHelper(new ItemTouchHelper
    .SimpleCallback(0, ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT) {})
```

3. You must now implement the desired behavior in `onSwiped()`. In this case, swiping the card left or right should delete it from the list. Call `remove()` on the data set, passing in the appropriate index by getting the position from the `ViewHolder`:

```
mSportsData.remove(viewHolder.getAdapterPosition());
```

4. To allow the RecyclerView to animate the deletion properly, you must also call `notifyItemRemoved()`, again passing in the appropriate index by getting the position from the ViewHolder:

```
mAdapter.notifyItemRemoved(viewHolder.getAdapterPosition());
```

5. Call `attachToRecyclerView()` on the ItemTouchHelper instance to add it to your RecyclerView:

```
helper.attachToRecyclerView(mRecyclerView);
```

6. Run your app, you can now swipe list items left and right to delete them!

**Note:** When using the app in landscape mode, the left and right swipe to dismiss is not as intuitive. To improve the user experience, you would change the behavior to something that is more consistent in a grid as well as a list.

## 3.2 Implement drag and drop

You can also implement drag and drop functionality using the same SimpleCallback. Do the following:

1. Change the first argument of the SimpleCallback from 0 to include every direction, since we want to be able to drag and drop anywhere:

```
ItemTouchHelper helper = new ItemTouchHelper(new ItemTouchHelper.SimpleCallback(ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT | ItemTouchHelper.DOWN | ItemTouchHelper.UP, ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT) {})
```

2. In `onMove()`, get the original and target index from the 2nd and 3rd argument passed in (corresponding to the original and target viewholders):

```
int from = viewHolder.getAdapterPosition();
int to = target.getAdapterPosition();
```

3. Swap the items in the dataset by calling `Collections.swap()` and passing in the dataset, and the initial and final indexes:

```
Collections.swap(mSportsData, from, to);
```

- Notify the adapter that the item was moved, passing in the old and new indexes:

```
mAdapter.notifyItemMoved(from, to);
```

That's it! You can now delete your list items by swiping them left or right, or reorder them by a long press to activate Drag and Drop mode.

### 3.3 Implement the detail view

According to [Material Design guidelines](#), a card is used to provide an entry point to more detailed information. You may find yourself tapping on the cards to see more information about the sports, since this behavior is designed to be intuitive. In this section, you will add a detail activity that will be launched when any list item is pressed. For this practical, it will contain the name and image of the list item you clicked, but will contain generic placeholder detail text, so you don't have to create custom detail for each list item.

- Create a new activity by going to **File > New > Empty Activity**.
- Call it Detail Activity, accept all of the defaults.
- Navigate to the newly created layout file and remove the padding from the rootview RelativeLayout.
- Copy all of the views from the list\_item.xml file to the activity\_detail.xml file.
- Add the word "Detail" to every reference to an id, in order to differentiate it from list\_item ids. For example, change the ImageView id from sportsImage to sportsImageDetail, as well as any references to this id for relative placement such `layout_below`.
- For the subTitleDetail textView, remove the placeholder text string and paste a paragraph of generic text to substitute detail text (For example, a few paragraphs of [Lorem Ipsum](#)).
- Change the padding on the TextViews to 16dp.
- Wrap the entire activity\_detail.xml in a ScrollView and change the layout\_height attribute of the RelativeLayout to "wrap\_content".

**Note:** The attributes for the ScrollView might appear red at first. This is because you must first add an attribute that defines the android namespace. This is the attribute that shows up in all of your layout files by default:

`xmlns:android="http://schemas.android.com/apk/res/android"`. Simply move this declaration to the top level view and the red should go away.

- In the SportsAdapter class, make the ViewHolder inner class implement View.OnClickListener, and implement the required method (onClick()).

10. Set the OnClickListener to the itemview in the constructor:

```
itemView.setOnClickListener(this);
```

11. In the onClick() method, get the Sport object for the item that was clicked using getAdapterPosition().
12. Create an Intent that launches the Detail activity, and put the title and image resource as extras in the Intent:

```
Intent detailIntent = new Intent(mContext, DetailActivity.class);
detailIntent.putExtra("title", currentSport.getmTitle());
detailIntent.putExtra("image_resource", currentSport.getmImageResource());
```

13. Call startActivity() on the mContext variable, passing in the new Intent.
14. In DetailActivity.java, initialize the ImageView and title TextView:

```
TextView sportsTitle = (TextView) findViewById(R.id.titleDetail);
ImageView sportsImage = (ImageView) findViewById(R.id.sportsImageDetail);
```

15. Get the title from the incoming Intent and set it to the TextView:

```
sportsTitle.setText(getIntent().getStringExtra("title"));
```

16. Use Glide to load the image into the ImageView:

```
Glide.with(this).load(getIntent().getIntExtra("image_resource", 0)).into(sportsImage);
```

17. Run the app. Tapping on a list item now launches the detail activity.

## Task 4: The FAB and Material Color Palettes

One of the principles behind Material Design is using consistent elements across applications and platforms so that users recognize patterns and know how to use them. You have already used one such element: the [Floating Action Button](#). The FAB is a circular button that floats above the rest of the UI. It is used to promote a particular action to the user, one that he is very likely to use in a given activity. In this task, you will create a FAB that resets the dataset to its original state.

## 4.1 Add the FAB

The Floating Action Button is part of the [Design Support Library](#).

1. Add the following line of code to the app level build.gradle file to add the design support library dependency:

```
compile 'com.android.support:design:22.2.0'
```

2. Use the vector asset studio to download an icon to use in the FAB. The button will reset the contents of the RecyclerView so this icon should do: . Change the name to ic\_reset.
3. In activity\_main.xml, add a Floating Action Button view with the following parameters:

Attribute	Value
android:layout_height	"wrap_content"
android:layout_width	"wrap_content"
android:layout_alignParentBottom	"true"
android:layout_alignParentRight	"true"
android:layout_margin	"16dp"
android:src	"@drawable/ic_reset"
android:onClick	resetSports

4. Implement the resetSports() method in MainActivity, and have it simply call initializeData() to reset the data.
5. Run the app, you can now reset the data by pushing the FAB.

**Note:** Because the activity is destroyed and recreated when the configuration changes, rotating the device effectively resets the data in this implementation. In order to persist the changes (like reordering or removing data), you would have to implement onSaveInstanceState() or write the changes to a persistent source (like a database or SharedPreferences).

## 4.2 Material Palette

If you run your app you may notice that the FAB has a color that you didn't define anywhere. Furthermore, the App bar (the bar that contains the title of your app) also has a color that you did not explicitly set. Where are these colors defined?

Navigate to your styles.xml file (located in the values directory). Under the AppTheme style, there are three colors defined by default: colorPrimary, colorPrimaryDark and colorAccent. These styles are defined by values from the colors.xml file.

Material Design recommends picking at least these three colors for your app:

- A primary color. This one is automatically used to color your App bar.
- A primary dark color. A darker shade of the same color. This is used for the status bar above the app bar among other things.
- An accent color. A color that contrasts well with the primary color. This is used for various highlights, but also the default color of the FAB.

You can use the [Material Color Guide](#) to pick some colors to experiment with. Do the following:

1. Pick a color from the guide to use as your primary color. It should be within the 300-700 range so that you can still pick a proper accent and dark color.
2. Pick a darker shade of the same color to use as your primary dark color.
3. Pick an accent color for your FAB from the colors whose values start with an A. (like Orange A200).
4. In colors.xml, modify the hex values to represent the ones you just chose.
5. Add the android:tint attribute to the FAB and set it equal to white to change the icon color to white.
6. Run the app. The App Bar and FAB changed to reflect the new color palette!

**Note:** If you want to change the color of the FAB to something other than theme colors, use the app:backgroundTint attribute. Note that this uses the app: namespace and Android Studio will prompt you to add a statement to define the namespace.

## Coding challenge

**Note:** All coding challenges are optional and not prerequisite for the material in the next chapter.

This challenge consists of several small improvements to the application:

- Come up with a way to delete cards that feels more natural in the grid layout and implement it using the ItemTouchHelper.
- Add real details to the Sport object and pass them to the detail view.
- Implement a way to persist the state of the app across orientation changes.

## Conclusion

The Material Design guidelines are a set of guiding principles that aim to create consistent, intuitive and playful applications. These principles include:

- The use of bold imagery and colors to enhance user experience.

- Consistent elements across platforms (such as CardView and the FAB).
- Meaningful, intuitive motion like dismissable or rearrangeable cards.

## Resources

- [Material Design Guidelines](#)
- [Material Palette Generator](#)
- [Cards and Lists Guide](#)
- [Floating Action Button Reference](#)

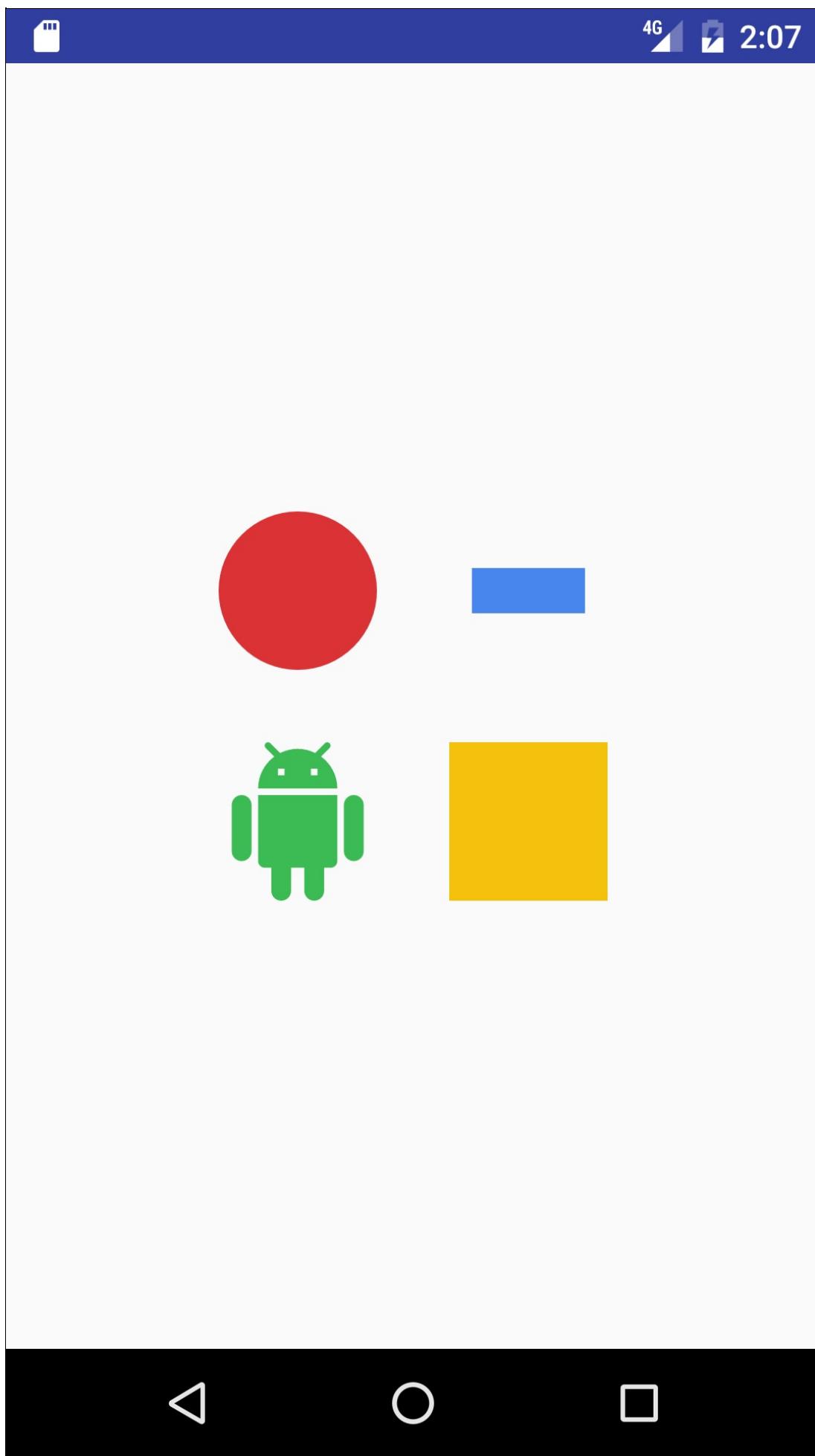
## 5.3 PC: Transitions and Animations

### Contents:

- [Challenge](#)
- [Solution](#)
- [Resources](#)

Create an application with 4 images arranged in a grid in the center of your layout. Make the first three solid colored backgrounds with different shapes (square, circle, line), and the fourth the [Android Material Design Icon](#). Each of these images should respond to clicks as follows:

1. One of the colored blocks relaunches the Activity using the [Explode](#) animation for both the enter and exit transitions.
2. Relaunch the Activity from another colored block, this time using the [Fade](#) transition.
3. Touching the third colored block starts an in place animation of the view (such as a rotation).
4. Finally, touching the android icon starts a secondary activity with a Shared Element Transition swapping the Android block with one of the other blocks.



**Note:** You must set your minimum SDK level to 21 or higher in order to implement shared element transitions.

## Solution

## Resources

### Developer Documentation:

- [Defining Custom Animations](#)
- [View Animation](#)

# 6.1 P: Use Espresso to test your UI

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App Overview
- Task 1: Set up Espresso in your project
- Task 2: Test for switching activities and entering text
- Task 3: Test the display of spinner selections
- Task 4: Record a test of a RecyclerView
- Coding challenge
- Summary
- Resources

A developer tests user interactions within an app helps to ensure that the app's users don't encounter unexpected results or have a poor experience when interacting with your app.

You can test a user interface for a complex app manually by running the app and trying the user interface. But you can't possibly cover all permutations of user interactions and all of the app's functionality. You would also have to repeat these manual tests on many different device configurations in an emulator, and on many different devices.

When you automate tests of UI interactions, you free yourself for other work. You can use suites of automated tests to perform all of the UI interactions automatically, which makes it easier to run tests for different device configurations. Get into the habit of creating user interface (UI) tests to verify that the UI of your app is functioning correctly.

Espresso is a testing framework for Android that makes it easy to write reliable user interface (UI) tests for an app. The framework, which is part of the Android Support Repository, provides APIs for writing UI tests to simulate user interactions within the app — everything from clicking buttons and navigating views to choosing menu selections and entering data.

## What you should already KNOW

You should be familiar with:

- Creating and running apps in Android Studio.

- Creating and editing UI elements using the Layout Editor, entering XML code directly, and accessing UI elements from your Java code.
- Adding onClick functionality to a button.
- Building the TwoActivities app from a previous lesson.
- Building the PhoneNumberSpinner app from a previous lesson.

## What you will LEARN

- How to set up Espresso in your app project.
- Writing an Espresso test that tests for user input and checks for the correct output.
- Writing an Espresso test to find a spinner, click one of its items, and check for the correct output.

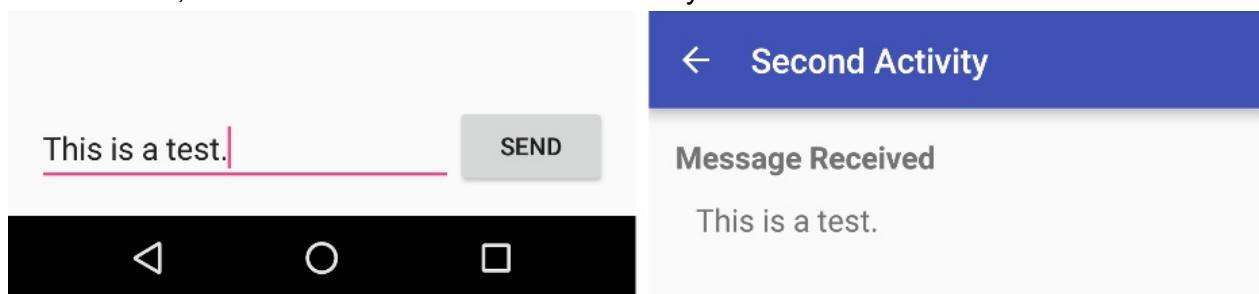
## What you will DO

In this practical application you will:

- Modify a project to create Espresso tests.
- Test the app's text input and output.
- Test clicking a spinner item and check its output.

## App Overview

You will modify the TwoActivities project to set up Espresso in the project for testing. You will then test the app's functionality, which enables a user to enter text into a text field, click the **Send** button, and view that text in a second activity.



**Tip:** For an introduction to testing Android apps, see [Test Your App](#).

## Task 1: Set up Espresso in your project

To use Espresso, you must already have the Android Support Repository installed with Android Studio. You must also configure Espresso in your project.

In this task you check to see if the repository is installed and if it's not, you install it. You then configure Espresso in the TwoActivities project you created previously.

## 1.1 Check for the Android Support Repository

1. Start Android Studio, and choose **Tools > Android > SDK Manager**.
2. Click the **SDK Tools** tab, and look for the Android Support Repository.
  - If “Installed” appears in the Status column, you’re all set. Click **Cancel**.
  - If “Not installed” appears, or an update is available:
    - i. Click the checkbox next to Android Support Repository. A download icon should appear next to the checkbox.
    - ii. Click one of the following:
      - **Apply** to start installing the repository and remain in SDK Manager to make other changes.
      - **OK** to install the repository and quit the SDK Manager.

## 1.2 Configure Espresso in your project

1. Open the TwoActivities project.
2. In Project view, under Gradle Scripts open the **build.gradle (Module: app)** file.  
**Note:** Do not make changes to the **build.gradle (Project: yourappname)** file.
3. Add the following instrumentation statement to the end of the `defaultConfig` section:

```
testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
```

**Instrumentation** is a set of control methods, or hooks, in the Android system. These hooks control an Android component independently of its normal lifecycle. They also control how Android loads apps. Using instrumentation makes it possible for tests to invoke methods in the app, and modify and examine fields in the app, independently of the app’s normal lifecycle.

4. Add the following code to the `dependencies` section:

```
dependencies {  
    ...  
    androidTestCompile 'com.android.support:support-annotations:24.1.1'  
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2'  
    androidTestCompile 'com.android.support.test:runner:0.5'  
    androidTestCompile 'com.android.support.test:rules:0.5'  
    androidTestCompile 'org.hamcrest:hamcrest-library:1.3'  
}
```

5. Update the version numbers, if necessary.

If the version numbers you specified are lower than the currently available library version numbers, Android Studio will warn you (such as, "a newer version of com.android.support:support-annotations is available"). Update the version numbers to the ones Android Studio tells you to use.

6. Click the **Sync Now** link in the notification about Gradle files in top right corner of the window.

## 1.3 Turn off animations on your test device

If you testing on a device rather than the emulator, turn off animations on your test device — leaving system animations turned on might cause unexpected results or may lead your test to fail. Turn off animations from Settings by opening **Developing Options** and turning the following options off:

- Window animation scale
- Transition animation scale
- Animator duration scale

## Task 2: Test for switching activities and entering text

You write Espresso tests based on what a user might do while interacting with your app. The key tasks to do in a test are *locating* a view in the UI, and then *interacting* with the view. You find a view you are interested in, and then check its state or interact with it. With Espresso, you create what is essentially a script of actions to take on views, and checks against expected results.

To create a test, you create a method within the test class that uses Hamcrest expressions. *Hamcrest* (an anagram of “matchers”) is a framework that assists writing software tests in Java. The framework lets you create custom assertion matchers, allowing match rules to be defined declaratively. With Espresso you use the following types of Hamcrest expressions to help find views and interact with them:

- *ViewMatchers*: A ViewMatcher expression lets you find a view in the current view hierarchy so that you can examine something or perform some action.
- *ViewActions*: A ViewAction expression lets you perform an action on the view already found by a ViewMatcher.
- *ViewAssertions*: A ViewAssertion expression lets you assert or checks the state of a view found by a ViewMatcher.

You would typically combine a ViewMatcher and a ViewAction in a single statement, followed by a ViewAssertion expression in a separate statement or included in the same statement.

You can see how all three expressions work in the following statement, which combines a ViewMatcher to find a view, a ViewAction to perform an action, and a ViewAssertion to check if the result of the action matches an assertion:

```
onView(withId(R.id.my_view))           // withId(R.id.my_view) is a ViewMatcher
    .perform(click())                 // click() is a ViewAction
    .check(matches(isDisplayed()));   // matches(isDisplayed()) is a ViewAssertion
```

You will use all three expressions in the test methods you create.

## 2.1 Define a class for a test and set up the activity

Android Studio creates a blank Espresso test class for you in the `src/androidTest/java/com.example.package` folder:

1. Expand `com.example.android.twoactivities (androidTest)`, and open `ApplicationTest`.
2. To make the test more understandable and describe what it does, rename the class from `ApplicationTest` to the following:

```
public class ActivityInputOutputTest
```

3. Add the following to the class definition:

```
@RunWith(AndroidJUnit4.class)
@LargeTest
public class ActivityInputOutputTest {
    @Rule
    public ActivityTestRule mActivityRule = new ActivityTestRule<>(
        MainActivity.class);
}
```

The class definition now includes several annotations:

4. `@RunWith` : To create an instrumented JUnit 4 test class, add the `@RunWith(AndroidJUnit4.class)` annotation at the beginning of your test class definition.
5. `@LargeTest` : The `@LargeTest` Android annotation tells you that the test may interact with multiple threads, the file system, or the network. In this case, the test is an integration test for two activities. For a description of the Android `@SmallTest`, `@MediumTest`, and `@LargeTest` annotations, see “[Test Sizes](#)” in the Google Testing Blog.

6. `@Rule` : The `@Rule` annotation lets you add or redefine the behavior of each test method in a reusable way, using one of the test rule classes that the Android Testing Support Library provides, such as `ActivityTestRule` or `ServiceTestRule`. The rule above uses an `ActivityTestRule` object, which provides functional testing of a single Activity — in this case, `MainActivity.class`. During the duration of the test you will be able to manipulate your Activity directly, using ViewMatchers, ViewActions, and ViewAssertions.

7. In the above statement, `ActivityTestRule` is in red. Click the red light bulb to import:

```
import android.support.test.rule.ActivityTestRule;
```

## 2.2 Test switching activities

The TwoActivities app has two activities:

- `Main` : Includes the `button_main` button for switching to the `Second` activity and the `text_header_reply` view that serves as a text heading for the `Main` activity.
- `Second` : Includes the `button_second` button for switching to the `Main` activity and the `text_header` view that serves as a text heading for the `Second` activity.

When you have an app that switches activities, you should test that capability. The Two Activities app provides a text entry field and a **Send** button (the `button_main` id). Clicking **Send** launches the `Second` activity with the entered text shown in the `text_header` view of the `Second` activity.

But what happens if no text is entered? Will the `Second` activity still appear? This test will show that the views appear regardless of whether text is entered.

1. Add the `activityLaunch()` method to test whether the views appear when clicking the buttons, and include the `@Test` notation on a line immediately above the method:

```
@Test  
public void activityLaunch() { ... }
```

The `@Test` annotation tells JUnit that the `public void` method to which it is attached can be run as a test case. A test method begins with the `@Test` annotation and contains the code to exercise and verify a single function in the component that you want to test.

2. Add a combined ViewMatcher and ViewAction expression to the `activityLaunch()` method to locate the view containing the `button_main` button, and include a ViewAction expression to perform a click:

```
onView(withId(R.id.button_main)).perform(click());
```

The `onView()` method lets you use ViewMatcher arguments to find views. It searches the view hierarchy to locate a corresponding View instance that meets some given criteria — in this case, the `button_main` view. The `.perform(click())` expression is a ViewAction expression that performs a click on the view.

3. In the above `onView` statement, `onView`, `withID`, and `click` are in red. For each one, click on the term, and then click the red lightbulb icon in the left margin. Choose **Static import method...** from the pop-up menu that appears. After doing this for each term, the following import statements are added:

```
import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.action.ViewActions.click;
import static android.support.test.espresso.matcher.ViewMatchers.withId;
```

4. Add another ViewMatcher expression to find the `text_header` view (which is in the `Second` activity), and a ViewAction expression to perform a check to see if the view is displayed:

```
onView(withId(R.id.text_header)).check(matches(isDisplayed()));
```

This statement uses the `onView()` method to locate the `text_header` view for the `Second` activity and check to see if it is displayed after clicking the `button_main` view.

5. In the above `onView` statement, other items may be in red. For each one, click on the term, and then click the red lightbulb icon in the left margin. Choose **Static import method...**, and the import statements are added.
6. Add similar statements to test whether clicking the `button_second` button in the `Second` activity switches to the `Main` activity:

```
onView(withId(R.id.button_second)).perform(click());
onView(withId(R.id.text_header_reply)).check(matches(isDisplayed()));
```

7. Review the method you just created. It should look like this:

```
@Test
public void activityLaunch() {
    onView(withId(R.id.button_main)).perform(click());
    onView(withId(R.id.text_header)).check(matches(isDisplayed()));
    onView(withId(R.id.button_second)).perform(click());
    onView(withId(R.id.text_header_reply)).check(matches(isDisplayed()));
}
```

8. To run the test, right-click (or Control-click) **ActivityInputOutputTest** and choose **Run ActivityInputOutputTest** from the pop-up menu. You can then choose to run the test on the emulator or on your device.

As the test runs, watch the test automatically start the app and click the button. The `Second` activity's view appears. The test then clicks the `Second` activity's button, and the `Main` activity view appears.

The Run window (the bottom pane of Android Studio) shows the progress of the test, and when finishes, it displays “Tests ran to completion.” In the left column Android Studio displays “All Tests Passed”.

## 2.3 Test text input and output

Write a test for text input and output. The TwoActivities app uses the `editText_main` view for input, the `button_main` button for sending the input to the `Second` activity, and the `Second` activity view that shows the output in the field with the id `text_message`.

1. Add another `@Test` annotation and a new `textInputOutput()` method to the `ApplicationTest` class to test text input and output:

```
@Test  
public void textInputOutput() {  
    onView(withId(R.id.editText_main)).perform(typeText("This is a test."));  
    onView(withId(R.id.button_main)).perform(click());  
}
```

The above method uses a `ViewMatcher` to locate the view containing the `editText_main` view, and a `ViewAction` to enter the text `"This is a test."`. It then uses another `ViewMatcher` to find the view with the `button_main` button, and another `ViewAction` to click the button.

2. If `typeText` is in red, click on the term, and then click the red lightbulb in the left margin. Choose **Static import method...**, and the following import statement is added:

```
import static android.support.test.espresso.action.ViewActions.typeText;
```

3. Add a `ViewMatcher` to locate the `Second` activity's `text_message` view, and a `ViewAssertion` to see if the output matches the input to test that the message was correctly sent:

```
onView(withId(R.id.text_message)).check(matches(withText("This is a test.")));
```

You may have to import the following:

```
import static android.support.test.espresso.matcher.ViewMatchers.withText;
```

4. Run the test.

As the test runs, the app starts and the text is automatically entered as input; the button is clicked, and the text appears on the second activity's screen.

The bottom pane of Android Studio shows the progress of the test, and when finished, it displays “Tests ran to completion.” In the left column Android Studio displays “All Tests Passed”. You have successfully tested the text input field, the Send button, and the text output field.

### Solution code:

```
package com.example.android.twoactivities;

import android.support.test.rule.ActivityTestRule;
import android.support.test.runner.AndroidJUnit4;
import android.test.suitebuilder.annotation.SmallTest;

import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;

import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.action.ViewActions.click;
import static android.support.test.espresso.action.ViewActions.typeText;
import static android.support.test.espresso.assertion.ViewAssertions.matches;
import static android.support.test.espresso.matcher.ViewMatchers.isDisplayed;
import static android.support.test.espresso.matcher.ViewMatchers.withId;
import static android.support.test.espresso.matcher.ViewMatchers.withText;

@RunWith(AndroidJUnit4.class)
@LargeTest
public class ActivityInputOutputTest {
    @Rule
    public ActivityTestRule mActivityRule = new ActivityTestRule<>(
        MainActivity.class);

    @Test
    public void activityLaunch() {
        onView(withId(R.id.button_main)).perform(click());
        onView(withId(R.id.text_header)).check(matches(isDisplayed()));
        onView(withId(R.id.button_second)).perform(click());
        onView(withId(R.id.text_header_reply)).check(matches(isDisplayed()));
    }

    @Test
    public void textInputOutput() {
        onView(withId(R.id.editText_main)).perform(typeText("This is a test."));
        onView(withId(R.id.button_main)).perform(click());
        onView(withId(R.id.text_message)).check(matches(withText("This is a test.")));
    }
}
```

## 2.4 Introduce an error to show a test failing

Introduce an error in the test to see what a failed test looks like.

1. Change the match check on the `text_message` view from "This is a test." to "This is a failing test." :

```
onView(withId(R.id.text_message)).check(matches(withText("This is a failing test.")))
```

2. Run the test again. This time you will see the message in red, "1 test failed", above the bottom pane, and a red exclamation point next to `textInputOutput` in the left column. Scroll the bottom pane to the message "Test running started" and see that all of the results after that point are in red. The very next statement after "Test running started" is:

```
android.support.test.espresso.base.DefaultFailureHandler$AssertionFailedWithCauseError: 'with text: is "This is a failing test."' doesn't match the selected view.  
Expected: with text: is "This is a failing test."
```

## Task 3: Test the display of spinner selections

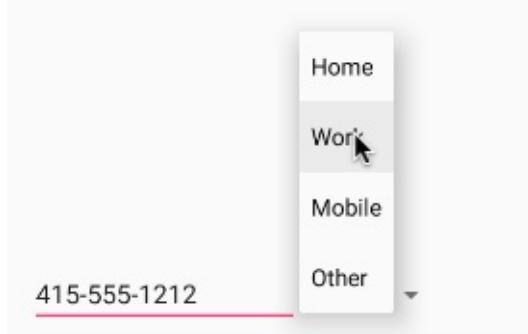
In an AdapterView such as a spinner, the view is dynamically populated with child views at runtime. If the target view you want to test is inside a spinner, the `onView()` method might not work because only a subset of the views may be loaded in the current view hierarchy.

The Espresso API handles this problem by providing a separate `onData()` entry point, which is able to first load the adapter item and bring it into focus prior to operating on it or any of its children.

PhoneNumberSpinner is an app from a previous lesson that shows a spinner, with the id `label_spinner`, for choosing the label of a phone number (**Home**, **Work**, **Mobile**, and **Other**). The app displays the choice in a text field, concatenated with the entered phone number.

The goal of this test is to open the spinner, click each item, and then verify that the TextView `text_phonelabel` contains the item. The test demonstrates that the code retrieving the spinner selection is working properly, and the code displaying the text of the spinner item is also working properly. You will write the test using string resources and iterate through the

spinner items so that the test works no matter how many items are in the spinner, or how those items are worded; for example, the words could be in a different language.



### 3.1 Create the test method

1. Set up and configure Espresso in your project as described previously.
2. Expand `com.example.android.phonenumberspinner (androidTest)`, and open `ApplicationTest`.
3. Rename `ApplicationTest` to `SpinnerSelectionTest` in the class definition, and add the following:

```
@RunWith(AndroidJUnit4.class)
@LargeTest
public class SpinnerSelectionTest {
    @Rule
    public ActivityTestRule<MainActivity> mActivityRule = new ActivityTestRule<>(
        MainActivity.class);
}
```

4. Create the `iterateSpinnerItems()` method as `public void`.

### 3.2 Access the array used for the spinner items

You want the test to click each item in the spinner based on the number of elements in the array. But how do you access the array?

1. Assign the array used for the spinner items to a new array to use within the `iterateSpinnerItems()` method:

```
public void iterateSpinnerItems() {
    String[] myArray =
        mActivityRule.getActivity().getResources()
            .getStringArray(R.array.labels_array);
}
```

In the statement above, the test accesses the application's array (with the id `labels_array`) by establishing the context with the `getActivity()` method of the

- [ActivityTestRule](#) class, and getting a resources instance in the application's package using `getResources()`.
- Get the size (length) of the array, and construct a for loop using the size as the maximum number for a counter.

```
int size = myArray.length;
for (int i=0; i<size; i++) {
```

### 3.3 Locate spinner items and click on them

- Add an `onView()` statement within the for loop to find the spinner and click on it:

```
// Find the spinner and click on it.
onView(withId(R.id.label_spinner)).perform(click());
```

Your test must click the spinner itself in order click any item in the spinner, so it must continually click the spinner first before clicking the item.

If any of the terms such as `onView`, `withId`, or `click` appear in red, click on the term, and then click the red lightbulb icon in the left margin. Choose **Static import method...** from the pop-up menu that appears.

- Write an `onData()` statement to find and click a spinner item:

```
// Find the spinner item and click on it.
onData(is(myArray[i])).perform(click());
```

As you enter statement, some portions of it may appear in red; click on each red term, click the red light bulb in the left margin, and choose **Static import method...** from the pop-up menu that appears. If choices are provided, choose the Hamcrest Matchers framework versions or Android Support Library versions.

The above statement matches if the object is a specific item in the spinner, as specified by the `myArray[i]` array element.

- Add two more `onView()` statements to the for loop:

```
// Find the button and click on it.
onView(withId(R.id.button_main)).perform(click());
// Find the text view and check that the spinner item is part of the string.
onView(withId(R.id.text_phonelabel))
    .check(matches(withText(containsString(myArray[i]))));
```

The first statement locates the `button_main` and clicks it. The second statement checks to see if the resulting `text_phonelabel` matches the spinner item specified by `myArray[i]`.

4. To run the test, right-click (or Control-click) **SpinnerSelectionTest** and choose **Run **SpinnerSelectionTest**** from the pop-up menu. You can then choose to run the test on the emulator or on your device.

The test runs the app, clicks the spinner, and “exercises” the spinner — it clicks each spinner item from top to bottom, checking to see if the item appears in the text field. It doesn’t matter how many spinner items are defined in the array, or what language is used for the spinner’s items — the test performs all of them and checks their output against the array.

The bottom pane of Android Studio shows the progress of the test, and when finished, it displays “Tests ran to completion.” In the left column Android Studio displays “All Tests Passed”.

## Solution code

```
package com.example.android.phonenumberspinner;

import android.support.test.rule.ActivityTestRule;
import android.support.test.runner.AndroidJUnit4;
import android.test.suitebuilder.annotation.SmallTest;

import org.hamcrest.Matchers;
import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;

import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.action.ViewActions.click;
import static android.support.test.espresso.assertion.ViewAssertions.matches;
import static android.support.test.espresso.matcher.ViewMatchers.withId;
import static android.support.test.espresso.matcher.ViewMatchers.withText;
import static org.hamcrest.Matchers.containsString;
import static org.hamcrest.Matchers.instanceOf;
import static org.hamcrest.core.Is.is;

@RunWith(AndroidJUnit4.class)
@LargeTest
public class SpinnerSelectionTest {

    @Rule
    public ActivityTestRule mActivityRule =
            new ActivityTestRule<>(MainActivity.class);

    @Test
    public void iterateSpinnerItems() {
        // Get the string array of spinner elements.
        String[] myArray =
                mActivityRule.getActivity().getResources()
                        .getStringArray(R.array.labels_array);

        // Iterate through the spinner array of items.
        int size = myArray.length;
        for (int i=0; i<size; i++) {
            // Find the spinner and click on it.
            onView(withId(R.id.label_spinner)).perform(click());
            // Find the spinner item and click on it.
            onData(is(myArray[i])).perform(click());
            // Find the button and click on it.
            onView(withId(R.id.button_main)).perform(click());
            // Find the text view and check that the spinner item is part of the string.
            onView(withId(R.id.text_phonelabel))
                    .check(matches(withText(containsString(myArray[i]))));
        }
    }
}
```

# Task 4: Record a test of a RecyclerView

You learned how to create a [RecyclerView](#) in a previous chapter with the Recycler View app. Like an AdapterView (such as a spinner), a RecyclerView dynamically populates child views at runtime. But a RecyclerView is not an AdapterView, so you can't use `onData()` to interact with list items as you did in the previous task with a spinner. What makes a RecyclerView complicated from the point of view of Espresso is that `onView()` can't find the child view if it is off the screen.

Fortunately, you have two handy tools to circumvent these complications:

- A class called [RecyclerViewActions](#) that exposes a small API to operate on a RecyclerView.
- An Android Studio feature (in version 2.2 and newer) that lets you *record* an Espresso test. Use your app as a normal user — as you click through the app UI, editable test code is generated for you. You can also add assertions to check if a view holds a certain value.

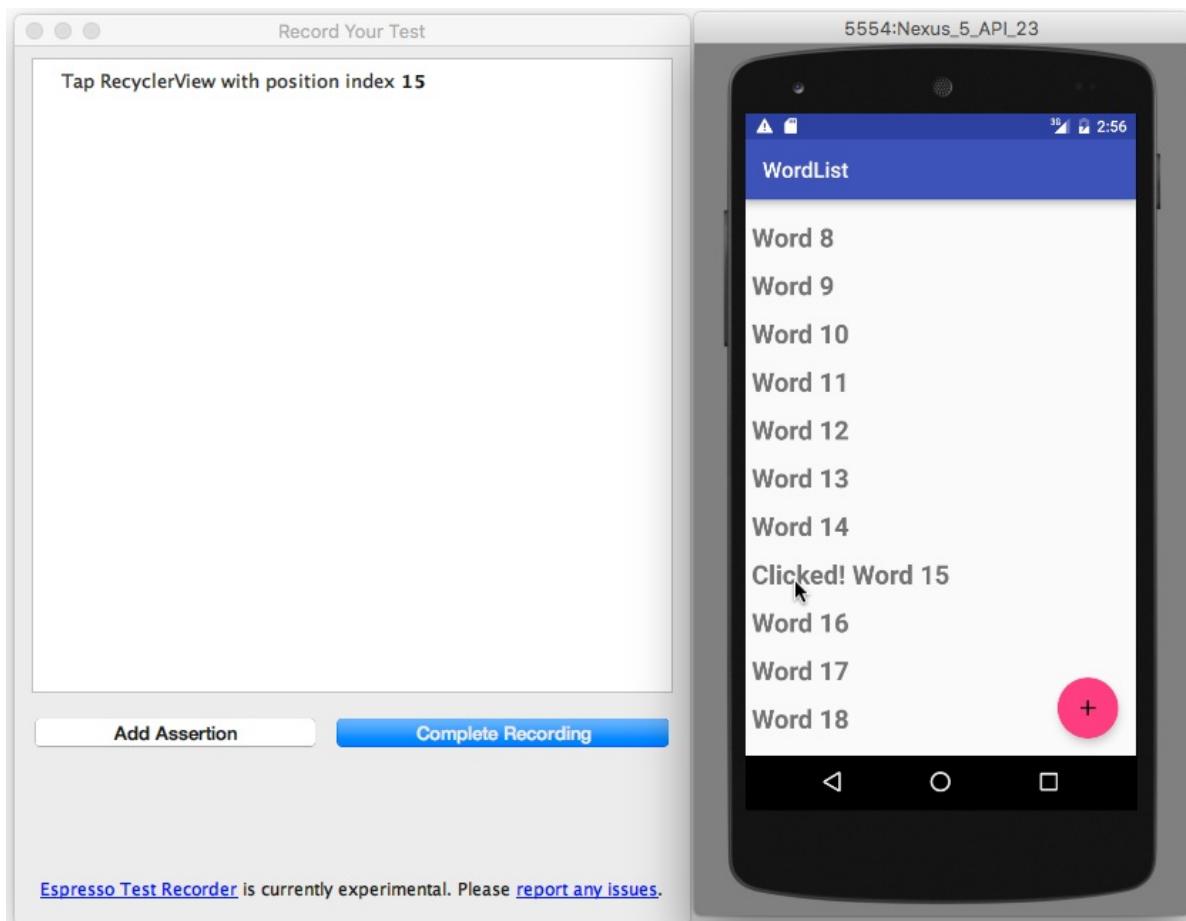
## 4.1 Open and run the app

1. Open the Recycler View project you created in a previous chapter.
2. Run the app to ensure that it runs properly. You can use the emulator or an Android device.

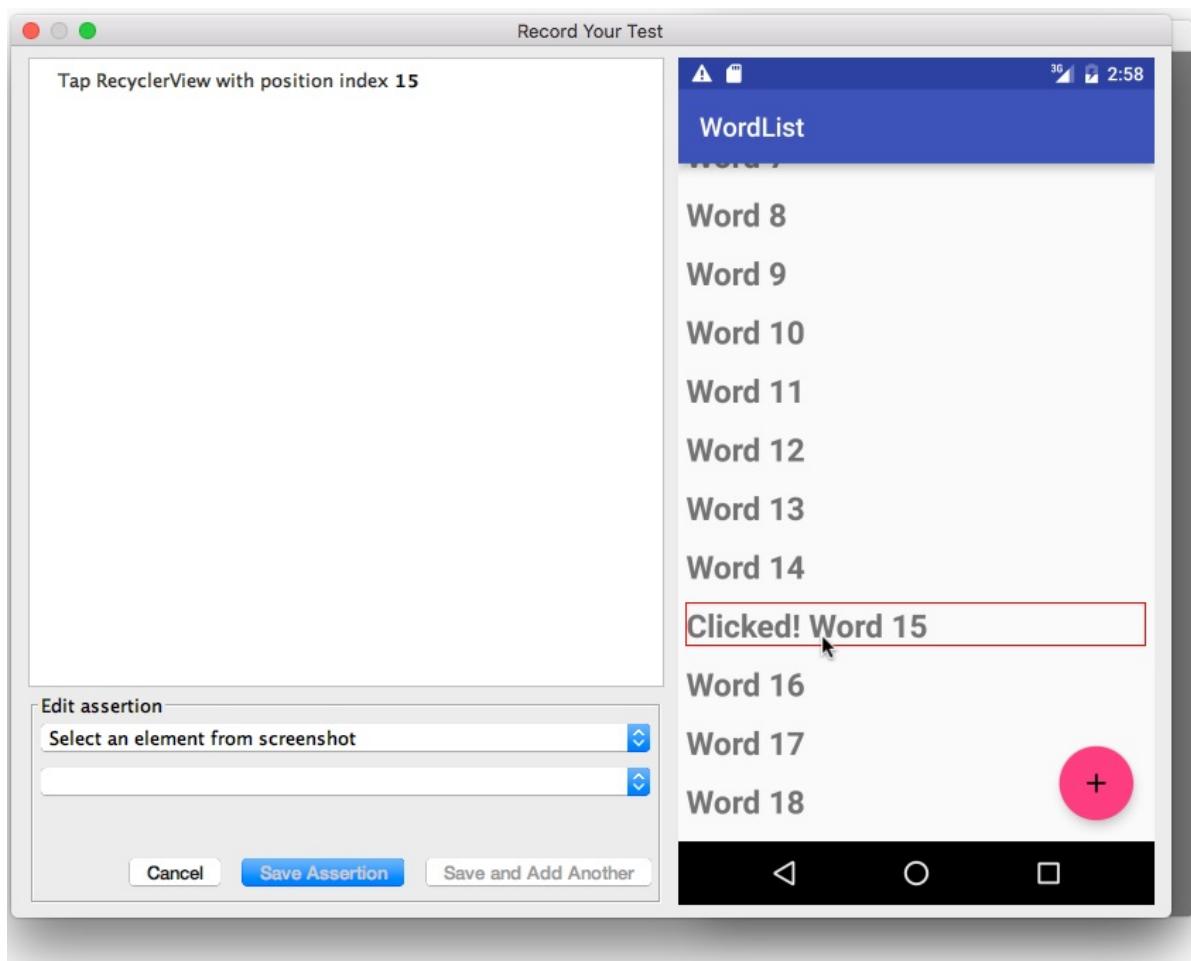
The app lets you scroll a list of words. When you click on a word such as “Word 15” the word in the list changes to “Clicked! Word 15”.

## 4.2 Record the test

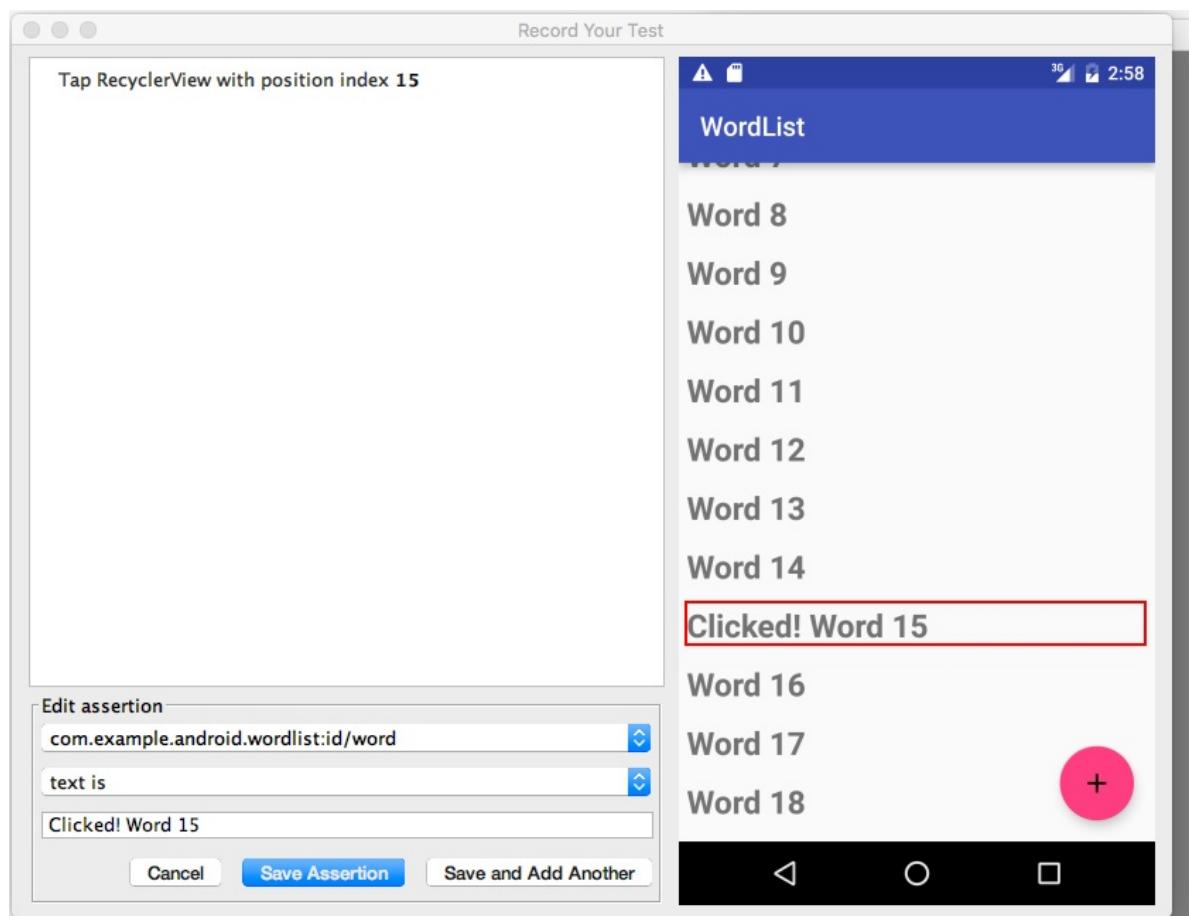
1. Choose **Run > Record Espresso Test**, and click **Restart app**. Select your deployment target (an emulator or a device), and click **OK**.
2. Scroll the word list in the app, and click on **Word 15**. The Record Your Test window shows the action that was recorded (“Tap RecyclerView with position index 15”).



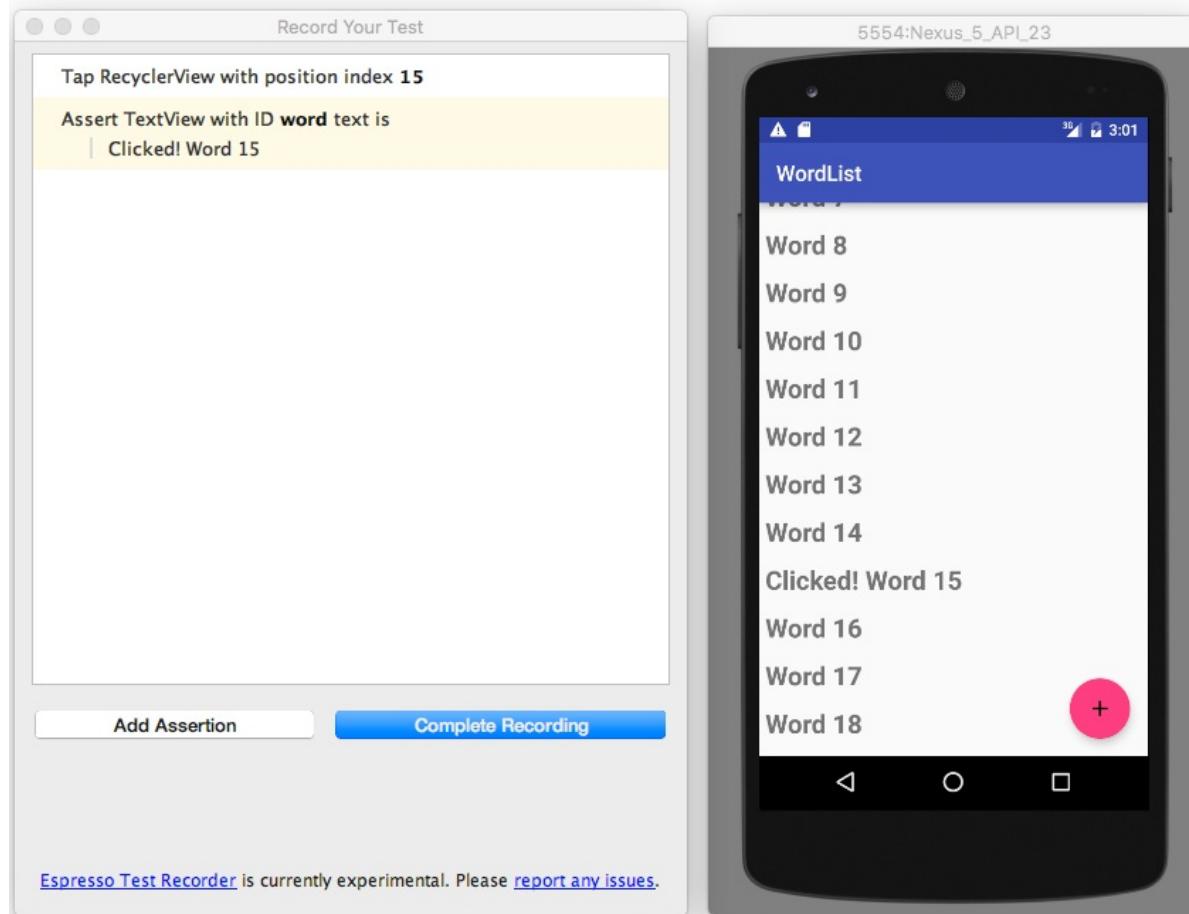
3. Click **Add Assertion** in the Record Your Test window, and select **Clicked! Word 15** in the screenshot on the right side as the UI element you want to check.



4. Choose **text is** from the second drop-down menu. The text you expect to see is already entered in the field below the drop-down menu.



5. Click **Save Assertion**, and then click **Complete Recording**.



6. In the dialog that appears, edit the name of the test to **RecyclerViewTest** so that it is easy to understand the test's purpose.
7. If you have not configured your project for Espresso, a warning appears with a request to add the appropriate dependencies to your Gradle Build file. Click **Yes** to add the dependencies.
8. Expand **com.example.android.recyclerview (androidTest)** to see the test, and run the test. It should pass.

The following is the test as recorded:

```
package com.example.android.recyclerview;

import android.support.test.espresso.ViewInteraction;
import android.support.test.rule.ActivityTestRule;
import android.support.test.runner.AndroidJUnit4;
import android.test.suitebuilder.annotation.LargeTest;

import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;

import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.action.ViewActions.click;
import static android.support.test.espresso.assertion.ViewAssertions.matches;
import static android.support.test.espresso.contrib.RecyclerViewActions.actionOnItemAtPosition;
import static android.support.test.espresso.matcher.ViewMatchers.isDisplayed;
import static android.support.test.espresso.matcher.ViewMatchers.withId;
import static android.support.test.espresso.matcher.ViewMatchers.withText;
import static org.hamcrest.Matchers.allOf;

@LargeTest
@RunWith(AndroidJUnit4.class)
public class RecyclerViewTest {

    @Rule
    public ActivityTestRule<MainActivity> mActivityTestRule =
            new ActivityTestRule<>(MainActivity.class);

    @Test
    public void recyclerViewTest() {
        ViewInteraction recyclerView = onView(
                allOf(withId(R.id.recyclerview), isDisplayed()));
        recyclerView.perform(actionOnItemAtPosition(15, click()));

        ViewInteraction textView = onView(
                allOf(withId(R.id.word), withText("Clicked! Word 15"), isDisplayed()))
        ;
        textView.check(matches(withText("Clicked! Word 15")));

    }
}
```

The test uses a `recyclerView` object of the `ViewInteraction` class, which is the primary interface for performing actions or asserts on views, providing both `check()` and `perform()` methods. Each interaction is associated with a view identified by a view matcher:

- In the first statement below, `recyclerView` is defined to be the `RecyclerView`. The second statement uses `.perform` with the `actionOnItemAtPosition()` method of the `RecyclerViewActions` class to scroll to the position (15) and click the item:

```
ViewInteraction recyclerView = onView(
    allOf(withId(R.id.recyclerview), isDisplayed()));
recyclerView.perform(actionOnItemAtPosition(15, click()));
```

- In the first statement below, the view matcher is the `word` view with the text “Clicked! Word 15”. The second statement checks to see if it matches the assertion that it should be “Clicked! Word 15”:

```
ViewInteraction textView = onView(
    allOf(withId(R.id.word), withText("Clicked! Word 15"), isDisplayed()));
textView.check(matches(withText("Clicked! Word 15")));
```

You can record multiple interactions with the UI in one recording session. You can also record multiple tests, and edit the tests to perform more actions, using the recorded code as a snippet to copy, paste, and edit.

## Coding challenge

Write an Espresso text for the Scorekeeper app from a previous lesson that tests whether the Day Mode button appears after clicking **Night Mode**, and whether the Night Mode button appears after clicking **Day Mode**.

Project: ScorekeeperEspressoTest \*\*

## Summary

In this practical, you learned the following:

- Setting up Espresso to test an Android Studio project:
  - Checking for and Installing the Android Support Repository.
  - Adding instrumentation and dependencies to the **build.gradle (Module: app)** file.
  - Turning off animations in your test device.
  - Defining the test class.
- Testing to see whether an activity is launched:
  - Using the `onView()` method with ViewMatcher arguments to find views.
  - Using a ViewAction expression to perform a click.
  - Using a ViewAssertion expression to check if the view is displayed.
  - Using a ViewAssertion expression to see if the output matches the input.
- Testing a spinner:
  - Using the `onData()` method with a view that is dynamically populated by an

- adapter at runtime.
- Getting items from an app's array by establishing the context with `getActivity()` and getting a resources instance using `getResources()`.
- Using an `onData()` statement to find and click each spinner item.
- Using the `onView()` method with a ViewAction and ViewAssertion to check if the output matches the selected spinner item.
- Recording a test of a RecyclerView:
  - Using the [RecyclerViewActions](#) class that exposes methods to operate on a RecyclerView.
  - Recording an Espresso test to automatically generate the test code.

# Resources

## Android Studio Documentation:

- [Test Your App](#)
- [Espresso basics](#)
- [Espresso cheat sheet](#)

## Android Developer Documentation:

- [Best Practices for Testing](#)
- [Getting Started with Testing](#)
- [Testing UI for a Single App](#)
- [Building Instrumented Unit Tests](#)
- [Espresso Advanced Samples](#)
- [The Hamcrest Tutorial](#)
- [Hamcrest API and Utility Classes](#)
- [Test Support APIs](#)

## Android Testing Support Library:

- [Espresso documentation](#)
- [Espresso Samples](#)

## Videos

- [Android Testing Support - Android Testing Patterns #1](#) (introduction)
- [Android Testing Support - Android Testing Patterns #2](#) (onView view matching)
- [Android Testing Support - Android Testing Patterns #3](#) (onData and adapter views)

## Other:

- Google Testing Blog: [Android UI Automated Testing](#)

- Atomic Object: “[Espresso – Testing RecyclerViews at Specific Positions](#)”
- Stack Overflow: “[How to assert inside a RecyclerView in Espresso?](#)”
- GitHub: [Android Testing Samples](#)
- Google Codelabs: [Android Testing Codelab](#)

# 7.1 P: Create an AsyncTask

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [Task 1: Set up the SimpleAsyncTask project](#)
- [Task 2: Create the AsyncTask subclass](#)
- [Task 3: Final Steps](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

A thread is an independent path of execution in a running program. When an Android program is launched, the Android runtime system creates a thread called the “Main” thread. As your program runs, each line is executed in a serial fashion, line by line. This main thread is how your application interacts with components from the Android UI toolkit, and it’s why the main thread is sometimes called the “UI thread”. However, sometimes an application has a need to perform some resource-intensive work, such as downloading files, database queries, playing media or computing complex analytics. This type of intensive work can block the UI thread and yield poor performance of the user experience. Users may get frustrated and uninstall your Android app.

To keep the user experience (UX) running smoothly and responding quickly to user gestures,

the Android framework provides a helper class called `AsyncTask` which processes work off of the UI thread. An `AsyncTask` is an abstract Java class that provides one way to move this intensive processing onto a separate thread, allowing the UI thread to remain very responsive. Since the separate thread does not operate in synchronization with the calling thread, it is called an asynchronous thread. An `AsyncTask` also contains a callback that allows you to display the results back in the UI thread.

In this practical, you will learn how to add a background task to your Android app using an [AsyncTask](#).

## What you should already KNOW

- How to create an Activity.

- How to add a TextView to the layout for the activity.
- How to programmatically get the id for the TextView and set its content.
- How to use Button views and their onClick functionality.

## What you will LEARN

- How to add an AsyncTask to your app to run a task in your background.
- The drawbacks of using AsyncTask for background tasks.

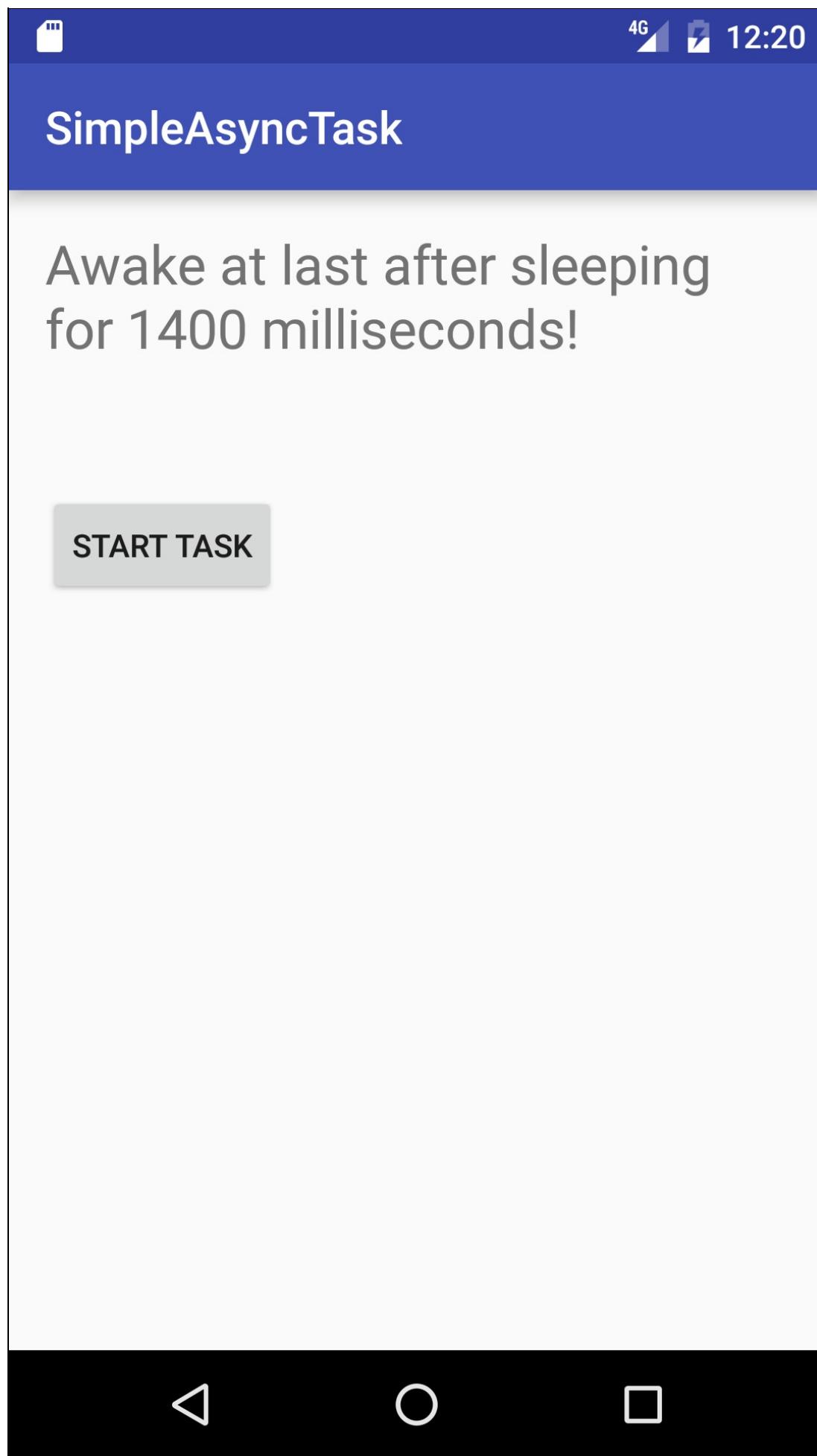
## What you will DO

- Create a simple application that executes a background task using an AsyncTask.
- Run the app and see what happens when you rotate the screen.

## App Overview

You will build an app that has one TextView and one button. When the user clicks the button, the app sleeps for a random amount of time, and then displays a message in the TextView when it wakes up.

Here's what the finished app will look like:



# Task 1. Set up the SimpleAsyncTask Project

The SimpleAsyncTask UI is straightforward. It contains a button that launches the AsyncTask as well as a TextView that will display the status of the application.

## 1.1 Create the layout

1. Create a new project called SimpleAsyncTask, that uses the **Empty Activity** template (use the defaults for the other options).
2. Change the rootview RelativeLayout to a LinearLayout.
3. Add the following essential UI elements in the layout for the main activity:

View	Attributes	Values
LinearLayout	android:orientation	vertical
TextView	android:text android:id	I am ready to start work! @+id/textView1
Button	android:text android:onClick	Start Task startTask

**Note:** You can set the layout height and width of each view to whatever you want, as long the views remain on the screen independent of the screen size (wrap\_content is probably the best bet).

4. The onClick attribute for the button will be highlighted in yellow, since the startTask() method is not yet implemented in the MainActivity. Place your cursor in the highlighted text, press **Alt + Enter (Option + Enter on a Mac)** and choose **Create 'startTask(View)' in 'MainActivity'** to create the method stub in MainActivity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/ready_to_start"
        android:id = "@+id/textView1"
        android:textSize="24sp"/>

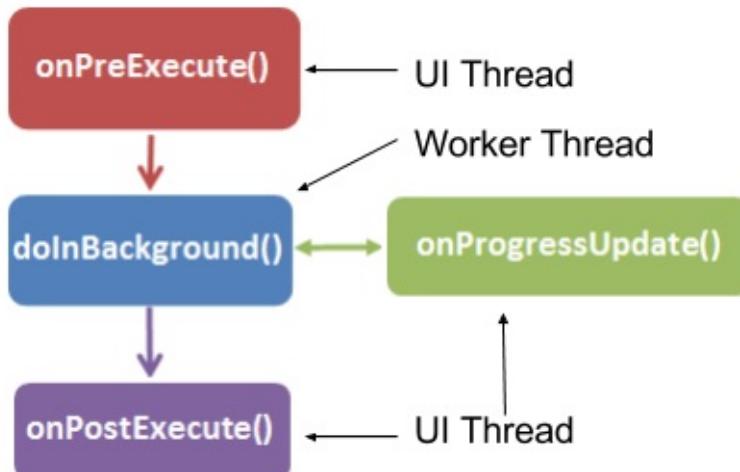
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/start_task"
        android:id="@+id/button"
        android:layout_marginTop="56dp"
        android:onClick="startTask" />
</LinearLayout>
```

## Task 2. Create the AsyncTask subclass

Since [AsyncTask](#) is an abstract class, you need to subclass it in order to use it. In this example the AsyncTask will execute a very simple background task: it just sleeps for a random amount of time. In a real app, the background task could perform all sorts of work, from querying a database, to connecting to the Internet, to calculating the next Go move to beat the current Go champion.

An AsyncTask has the following useful methods for performing work off of the main thread:

- `onPreExecute()` : This method runs on the UI thread, and used for setting up your task (like showing a progress bar).
- `doInBackground()` : This where your task is accomplished on a separate thread.
- `onProgressUpdate()` : Also invoked on the UI thread and used for updating progress in the UI (such as filling up a progress bar)
- `onPostExecute()` : Again on the UI thread, used for updating the results to the UI once the AsyncTask has finished loading.



When you create an

AsyncTask, you may need to give it information about the work to perform, whether and how to report its progress, and in what form the return the result.

In this exercise you will use an AsyncTask helper class to define work that will run in a different thread than the UI thread avoiding performance issues.

When you create an AsyncTask, you can configure it using these parameters:

- Params -- the data type of the parameters sent to the task upon executing the `doInBackground()` override method.
- Progress -- the data type of the progress units published using the `onProgressUpdated()` override method.
- Result -- the data type of the result delivered by the `onPostExecute()` override method.

For example, an AsyncTask with the following class declaration would take a String as a parameter in `doInBackground()` (to use in a query, for example), an Integer for `onProgressUpdate()` (percentage of job complete), and a Bitmap for the the result in `onPostExecute()` (the query result):

```
public class MyAsyncTask extends AsyncTask <String, Integer, Bitmap>{}
```

## 2.1 Subclass the AsyncTask

In your first AsyncTask implementation, the AsyncTask subclass will be very simple. It does not require a query parameter or publish its progress. You will only be using the `doInBackground()` and `onPostExecute()` methods.

1. Create a new Java class called SimpleAsyncTask that extends AsyncTask and takes three generic type parameters:
  - Void for the params, since this AsyncTask does not require any inputs.
  - Void for the progress type, since the progress is not published.

- A String as the result type, since you will update the TextView with a string when the AsyncTask has completed execution.

```
public class SimpleAsyncTask extends AsyncTask <Void, Void, String>{}
```

**Note:** The class declaration will be underlined in red, since the required method doInBackground() has not yet been implemented

The AsyncTask will need to update the TextView once it has completed sleeping. The constructor will therefore need to include the TextView, so that it can be updated in onPostExecute().

2. Define a member variable mTextView.
3. Implement a constructor for AsyncTask that takes a TextView and sets mTextView to the passed in TextView:

```
public SimpleAsyncTask(Textview tv) {
    mTextView = tv;
}
```

## 2.2 Implement doInBackground()

1. Add the required doInBackground() method. Place your cursor on the highlighted class declaration, press **Alt + Enter (Option + Enter on a Mac)** and select Implement methods. Choose doInBackground and click **OK**:

```
@Override
protected String doInBackground(Void... voids) {
    return null;
}
```

2. Implement doInBackground() to:

- generate a random integer between 0 and 10
- multiply that number by 200
- in a try/catch block, put the current thread to sleep. (Use Thread.sleep())
- return the String "Awake at last after xx milliseconds" (where xx is the number of milliseconds the app slept)

```

@Override
protected String doInBackground(Void... voids) {

    // Generate a random number between 0 and 10
    Random r = new Random();
    int n = r.nextInt(11);

    // Make the task take long enough that we have
    // time to rotate the phone while it is running
    int s = n * 200;

    // Sleep for the random amount of time
    try {
        Thread.sleep(s);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // Return a String result
    return "Awake at last after sleeping for " + s + " milliseconds!";
}

```

## 2.3 Implement onPostExecute()

When the doInBackground() method completes, the return value is automatically passed to the onPostExecute() callback.

1. Implement onPostExecute to take a String argument (this is what you defined in the third parameter of AsyncTask and what your doInBackground() method returned) and display that string in the TextView:

```

protected void onPostExecute(String result) {
    mTextView.setText(result);
}

```

**Note:** You can update the UI in onPostExecute() because it is run on the main thread. You cannot call mTextView.setText() in doInBackground() because that method is executed on a separate thread.

## Task 3. Final Steps

### 3.1 Implement the method that starts the AsyncTask

Your app now has an AsyncTask that performs work in the background (or it would if it didn't call sleep() as our simulated work). You can now implement the method that gets called when the Start Task button is clicked, to trigger the background task.

1. In the MainActivity.java file, add a member variable to store the TextView.

```
private TextView mTextView;
```

2. In the onCreate() method, initialize mTextView to refer to the TextView in our UI.
3. Add code to the startTask() method to create an instance of SimpleAsyncTask, passing the TextView to the constructor.
4. Call execute() on that SimpleAsyncTask instance.  
**Note:** the execute() method us where you would usually pass in the parameters (separated by commas) that would then be passed into doInBackground() by the system. Since this asynctask has no parameters, you will leave it blank.
5. Update the TextView to show the text "Napping..."

```
public void startTask (View view) {  
    // Put a message in the text view  
    mTextView.setText("Napping... ");  
  
    // Start the AsyncTask.  
    // The AsyncTask has a callback that will update the text view.  
    new SimpleAsyncTask(mTextView).execute();  
}
```

### Solution Code for MainActivity:

```

package android.example.com.simpleasynctask;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    // The TextView where we will show results
    TextView mTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Initialize mTextView
        mTextView = (TextView) findViewById(R.id.textView1);

    }

    public void startTask (View view) {
        // Put a message in the text view
        mTextView.setText("Napping... ");

        // Start the AsyncTask.
        // The AsyncTask has a callback that will update the text view.
        new SimpleAsyncTask(mTextView).execute();
    }
}

```

## 3.2 AsyncTask and rotations

1. Run the app and click the **Start Task** button. How long does the app nap?
  2. Click the Start Task button again, and while the app is napping, rotate the device.
- Note:** If the background task completes too quickly before you can rotate the phone while it is napping, try again. Alternatively, update the code and make it sleep for longer.

You'll notice that when the device is rotated, the TextView resets to its initial content and the AsyncTask doesn't seem able to update the TextView. There are two things going on here:

- Rotating the device restarts the app, calling `onDestroy()` and then `onCreate()`, restarting the activity lifecycle. AsyncTasks are not connected to the lifecycle of your app and will continue running if the activity is destroyed, without the ability to reconnect to the activity. The task will continue running in the background, but will not be able to update the TextView that was passed to it, because that particular TextView has been destroyed. The task will continue running to completion in the background, consuming

system resources, but never showing the results in the UI, which gets reset in `onCreate()`.

- Even without the `AsyncTask`, the rotation resets all of the UI elements to their default state, which for the `TextView` implies a particular string that you set in the `activity_main.xml` file.

For these reasons, `AsyncTasks` are not well suited to tasks which may be interrupted by the destruction of the Activity. In use cases where this is critical you can use a different type of class called a Loader which you will implement in a later practical. You've already learned how to maintain the state of the views in activity in the [Activity Lifecycle & State](#) practical, using the `SavedInstanceState` class. Revisit that lesson if this section proves challenging.

### 3.3 Implement `onSavedInstanceState()`

You will now implement `onSavedInstanceState()` to preserve the content of your `TextView` when the activity is spontaneously destroyed.

1. Override the `onSavedInstanceState()` method in `MainActivity` to preserve the text inside the `TextView` when the activity is destroyed:

```
outState.putString(TEXT_STATE, mTextView.getText().toString());
```

2. Retrieve the value of the `TextView` when the activity is restored in the `onCreate()` method.

```
// Restore TextView if there is a savedInstanceState
if(savedInstanceState!=null){
    mTextView.setText(savedInstanceState.getString(TEXT_STATE));
}
```

#### Solution Code for `MainActivity`:

```
package android.example.com.simpleasynctask;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.TextView;

/**
 * The SimpleAsyncTask app contains a button that launches an AsyncTask
 * which sleeps the asynchronous thread for a random amount of time.
 */
public class MainActivity extends AppCompatActivity {
```

```

//Key for saving the state of the TextView
private static final String TEXT_STATE = "currentText";

// The TextView where we will show results
private TextView mTextView = null;

/**
 * Initializes the activity.
 * @param savedInstanceState The current state data
 */
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // Initialize mTextView
    mTextView = (TextView) findViewById(R.id.textView1);

    // Restore TextView if there is a savedInstanceState
    if(savedInstanceState!=null){
        mTextView.setText(savedInstanceState.getString(TEXT_STATE));
    }
}

/**
 * Handles the onClick for the "Start Task" button. Launches the AsyncTask
 * which performs work off of the UI thread.
 *
 * @param view The view (Button) that was clicked.
 */
public void startTask (View view) {
    // Put a message in the text view
    mTextView.setText(R.string.napping);

    // Start the AsyncTask.
    // The AsyncTask has a callback that will update the textView.
    new SimpleAsyncTask(mTextView).execute();
}

/**
 * Saves the contents of the TextView to restore on configuration change.
 * @param outState The bundle in which the state of the activity is saved
 * when it is spontaneously destroyed.
 */
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // Save the state of the TextView
    outState.putString(TEXT_STATE, mTextView.getText().toString());
}
}

```

# Coding challenge

Note: All coding challenges are optional.

AsyncTask provides another very useful override method: `onProgressUpdate()`, which allows you to update the UI while the AsyncTask is running. Use this method to update the UI with the current sleep time. Look to the [AsyncTask documentation](#) to see how `onProgressUpdate()` is properly implemented. Remember that in the class definition of your AsyncTask, you will need to specify the data type to be used in the `onProgressUpdate()` method.

## Summary

- An AsyncTask is an abstract Java class that moves intensive processing onto a separate thread
- AsyncTask must be subclassed to be used
- Avoid resource-intensive work in the UI thread which may make your UI sluggish or erratic.
- Any code that does not involve drawing the UI or responding to the user input should be moved from the UI thread to another, separate thread
- AsyncTask has 4 useful methods: `onPreExecute()`, `doInBackground()`, `onPostExecute()` and `onProgressUpdate()`
- `doInBackground()` is the only method that is run on a separate worker thread. You should not call UI methods in this method.
- The other methods of AsyncTask run in the UI thread and allow calling methods of UI components
- Rotating an Android device destroys and recreates an Activity. This can disassociate the UI from the background thread, which will continue to run.
- You can save the state of the UI components during a device rotation using  
`onSaveInstanceState ()`

In this exercise, you used an AsyncTask to accomplish some work off of the UI thread, in order to preserve a fluid user experience. However there is one significant drawback to AsyncTask. We've learned an Activity, i.e., a screenful of GUI components, can be destroyed and recreated when a device is rotated. If the worker thread in an AsyncTask finishes and attempts to update the UI by using references to objects that were destroyed, no updates to the UI will occur. Needless to say this is not a good user experience. If the task must display results in the UI even when the device's configuration changes, for example, the device is rotated, you should use an AsyncTaskLoader instead of an AsyncTask. You will learn about AsyncTaskLoader in the next lesson.

# Resources

## Android Developer Documentation

### Guides

- [Processes and threads](#)
- [Processing Bitmaps off the UI thread using AsyncTask](#)

### Reference

- [AsyncTask](#)

## Other Web resources

- <https://realm.io/news/android-threading-background-tasks/>

## Videos

- [Threading Performance 101](#) by Performance Guru Colt McAnlis. Learn more about the main thread and why it's bad to run long-running tasks on the main thread.
- [Good AsyncTask Hunting](#) by Colt McAnlis. Learn more about AsyncTasks.

# 7.2 P: Broadcast Receivers

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [Task 1. Setup the PowerReceiver Project](#)
- [Task 2. Send and Receive a Custom Broadcast](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

There are certain events that can happen in the Android system which might affect the functionality of applications installed on the device. For example, if the system has finished booting, you might like to your weather app to update its information. The Android framework handles this by sending out system broadcasts containing Intents that are meant to be received using [BroadcastReceivers](#). Additionally, you can create Intents with custom actions and broadcast them yourself from your application using the `sendBroadcast()` method, which will be received by all application with a BroadcastReceiver registered for that action. To learn more about broadcast Intents and Broadcast receivers, visit the [Intent documentation](#).

In this practical you'll create an app that responds to a change in the charging state of your device, as well as sends and receives a custom Broadcast Intent.

## What you should already KNOW

- How to identify key parts of the `AndroidManifest.xml` file.
- How to create Implicit Intents.

## What you will LEARN

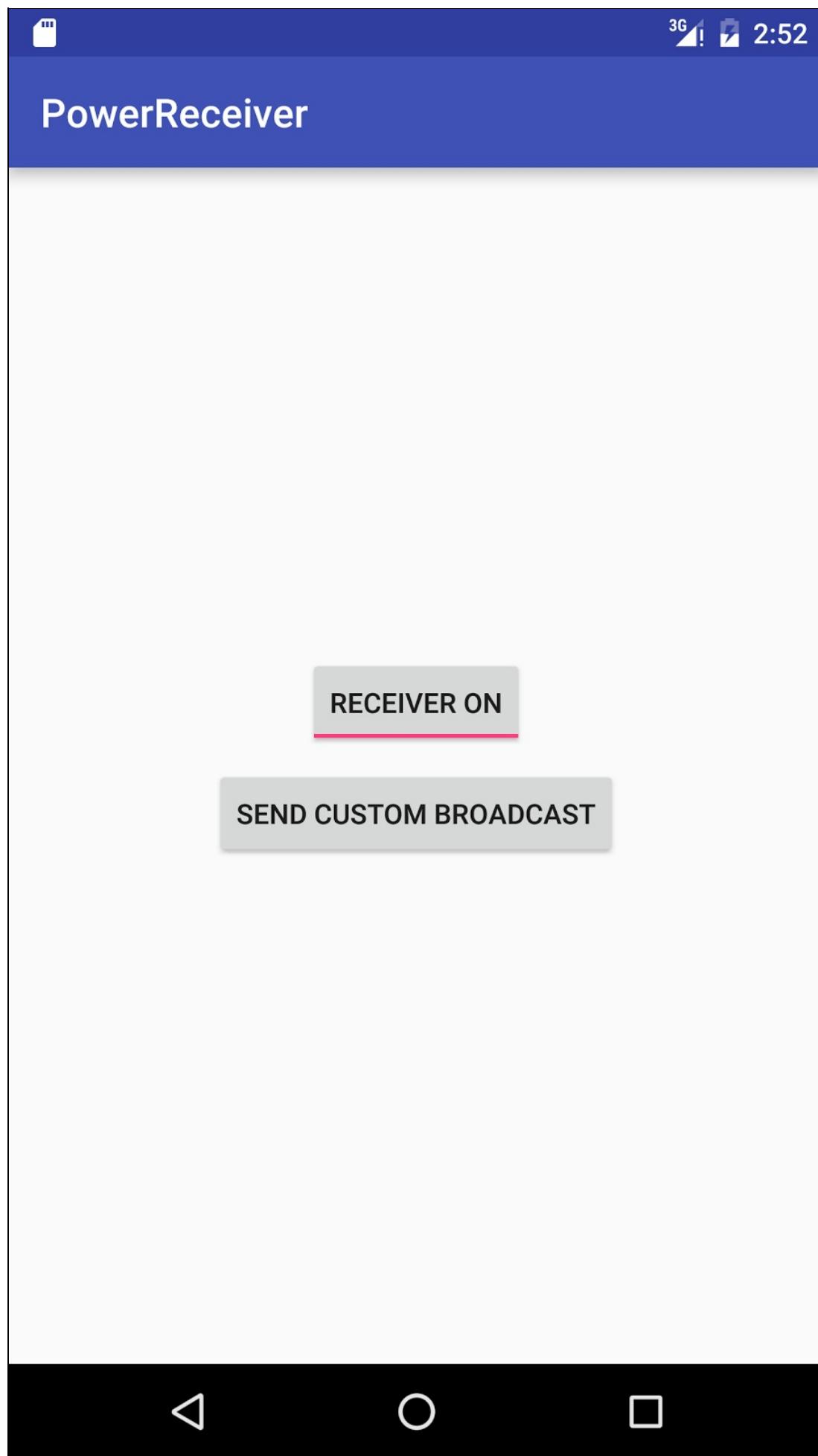
- How to subclass and implement a BroadcastReceiver.
- How to register for system Broadcast intents.
- How to create and send custom Broadcast intents.

## What you will DO

- Subclass a BroadcastReceiver to show a Toast when a broadcast is received.
- Register your receiver to listen to system broadcasts.
- Send and receive a custom broadcast intent.

## App Overview

The PowerReceiver application will register a BroadcastReceiver that displays a Toast when the device is connected or disconnected from power. It will also send and receive a custom Broadcast Intent to display a different Toast message.



# Task 1. Setup the PowerReceiver Project

## 1.1 Create the Project

To begin, create a new project called "PowerReceiver", accept the default options and use the empty template . Then do the following:

1. Navigate to **File > New > Other > Broadcast Receiver**.
2. Name the class "Custom Receiver" and make sure "Exported" and "Enabled" are checked.  
**Note:** the "Exported" feature allows your application to respond to outside broadcasts, while "Enabled" allows it to be instantiated by the system.
3. Navigate to your manifest file. Note that Android studio automatically generated a `<receiver>` tag with your chosen options as attributes. BroadcastReceivers can also be registered programmatically, but it is easiest to define them statically in the manifest.

## 1.2 Register your Receiver for system broadcasts

In order to receive any broadcasts, you must first determine which broadcast intents you are interested in. In the [Intent documentation](#), under "Standard Broadcast Actions", you can find some of the common broadcast intents sent by the system. In this app, you will be interested in two particular broadcasts: `ACTION_POWER_CONNECTED` and `ACTION_POWER_DISCONNECTED`. BroadcastReceivers register for broadcast the same way you registered your activities for implicit Intents: you use an intent filter.

In your manifest file, add the following code between the `<receiver>` tags in order to register your Receiver for the system Intents:

```
<intent-filter>
    <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
    <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
</intent-filter>
```

## 1.3 Implement onReceive() in your BroadcastReceiver

Once the BroadcastReceiver intercepts a broadcast that it is registered for, the Intent is delivered to the receiver's `onReceive()` method, along with the context in which the receiver is running.

1. Navigate to your CustomReceiver file, and delete the default implementation inside the

- onReceive() method.
2. Obtain the action from the intent and store it in a String variable called "intentAction":

```
@Override
public void onReceive(Context context, Intent intent) {
    String intentAction = intent.getAction();
}
```

3. Create a switch statement on the intentAction string, so that you can display a different Toast message for each specific action your receiver is registered for:

```
switch (intentAction){
    case Intent.ACTION_POWER_CONNECTED:
        break;
    case Intent.ACTION_POWER_DISCONNECTED:
        break;
}
```

4. Initialize a String variable called "toastMessage" before the switch statement, and make its value `null` so that it can be set depending on the broadcast action you receive.
5. Assign toastMessage to "Power connected!" if the action is `ACTION_POWER_CONNECTED`, and "Power disconnected!" if it is `ACTION_POWER_DISCONNECTED`. Extract your string resources.
6. Display a Toast for a short duration after the switch statement:

```
Toast.makeText(context, toastMessage, Toast.LENGTH_SHORT).show();
```

7. Run your app. After it is installed, unplug your device. It may take a moment the first time, but sure enough, a Toast is displayed each time you plug in, or unplug your device.

**Note:** If you are using an emulator, you can toggle the power connection state by selecting the ellipses icon for the menu, choose **Battery** on the left bar, and toggle using the **Charger connection** setting.

## 1.4 Add a receiver toggle to your MainActivity

Broadcast Receivers are always active, and therefore your app does not even need to be running for the onReceive method to be called. Go ahead, try it out: close your app, and plug or unplug your device; The Toast message is still displayed! This means there is a lot of responsibility on you, the developer, to not overwhelm your user with notifications or unwanted functionality every time a broadcast occurs. In this example, having a Toast pop

up every time the power state is changed is clearly an annoyance to the user. You will therefore add a toggle switch that will control the state of the BroadcastReceiver, and keep track of its state in SharedPreferences. Do the following:

1. Navigate to your activity\_main.xml file, remove the "Hello World!" TextView and change the root view to a vertical LinearLayout.
2. Add a ToggleButton view with the following attributes:

Attribute	Value
android:id	@+id/receiverToggle
android:layout_width	wrap_content
android:layout_height	wrap_content
android:textOn	"Receiver On"
android:textOff	"Receiver Off"

3. Set your LinearLayout gravity attribute to "center" and extract your String resources.
4. In MainActivity, create a member variable to keep track of the state of the Receiver:

```
private boolean mReceiverState;
```

5. Find and set the receiver toggle button by id in onCreate():

```
ToggleButton toggle = (ToggleButton) findViewById(R.id.receiverToggle);
```

6. Get a reference to the SharedPreferences file, and initialize the mReceiverState variable, with a default state of false (so that the receiver is disabled by default):

```
final SharedPreferences sharedpreferences = getPreferences(MODE_PRIVATE);
mReceiverState = sharedpreferences.getBoolean("receiveState", true);
```

7. Set the checked state of the toggle button based on the mReceiverState variable.

```
toggle.setChecked(mReceiverState);
```

The state of the BroadcastReceiver can be toggled by using the [PackageManager](#) class. To learn more about managing your BroadcastReceivers, visit [this guide](#). Do the following:

8. Get a reference to your receiver component by creating a ComponentName variable:

```
final ComponentName receiver = new ComponentName(this,CustomReceiver.class);
```

9. Create a variable for your PackageManager and initialize it using getPackageManager():

```
final PackageManager packageManager = getPackageManager();
```

10. Create an onCheckChangedListener for your toggle button, and set the mReceiverState variable appropriately:

```
toggle.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {  
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {  
        if (isChecked) {  
            mReceiverState = true;  
  
        } else {  
            mReceiverState = false;  
        }  
    }  
});
```

11. Create a receiverStateFlag variable that takes the value of one of two integer flags for the setComponentEnabledSetting() method of the package manager, depending on the value of mReceiverState:

```
int receiverStateFlag = (mReceiverState ?  
    PackageManager.COMPONENT_ENABLED_STATE_ENABLED :  
    PackageManager.COMPONENT_ENABLED_STATE_DISABLED);
```

12. At the end of the onCheckChanged() method, save the mReceiverState variable to Shared Preferences and update the BroadcastReceiver state using the PackageManager class:

```
sharedPreferences.edit().putBoolean("receiverState", mReceiverState).apply();  
packageManager.setComponentEnabledSetting(receiver, receiverStateFlag, PackageManager.DONT_KILL_APP);
```

**Note:** The SharedPreferences, PackageManager, and ComponentNames must be declared final since they are accessed from within the onCheckChanged() inner method.

## Task 2. Send and Receive a Custom Broadcast

Besides responding to system broadcasts, your application can also send and receive custom Broadcast Intents. In this task, you will add a button to your activity that sends a custom Broadcast Intent, which will be registered by your receiver and displayed in a Toast.

### 2.1 Define your custom Broadcast Action string

In order to register your receiver to respond to your custom Broadcast, you must have a unique string identify the action the Broadcast is representing. It is best practice to begin the action strings with your package name, so that it is assuredly unique.

Create a constant String variable in both your `MainActivity` and your `CustomReceiver` class to be used as the Broadcast Intent Action:

```
private static final String ACTION_CUSTOM_BROADCAST = "com.example.android.powerreceiver.ACTION_CUSTOM_BROADCAST";
```

### 2.2 Add a "Send Custom Broadcast" Button

1. In your `activity_main.xml` file, add a Button view with the following attributes:

Attribute	Value
<code>android:id</code>	"@+id/sendBroadcast"
<code>android:layout_width</code>	<code>wrap_content</code>
<code>android:layout_height</code>	<code>wrap_content</code>
<code>android:text</code>	"Send Custom Broadcast"
<code>android:layout_margin</code>	"8dp"
<code>android:onClick</code>	"sendBroadcast"

2. Extract your string resources.
3. Click in the yellow highlighted `onClick` method name. Press **Alt + Enter** and choose "**Create 'sendBroadcast(View)' in 'Main Activity'**".

### 2.3 Implement sendBroadcast()

Because this broadcast is meant to be used solely by your application, it is far more secure and efficient to use the `LocalBroadcastManager` class to manage your broadcasts. Do the following:

1. In the `sendBroadcast()` method in `MainActivity`, create a new Intent, with your custom

- action string as the argument.
2. Call LocalBroadcastManager.getInstance(this).sendBroadcast(customBroadcastIntent) to send the broadcast using the LocalBroadcastManager class.

## 2.4 Register your Custom Broadcast

For system broadcasts, you registered your receiver in the AndroidManifest.xml file. It is also possible (in fact, for broadcasts sent using LocalBroadcastManager, it is required) to register your receiver for specific actions programmatically. If you do, you must remember to also unregister the receiver when it is no longer needed. In this application, our receiver will only need to respond to the custom broadcast when it is running, so we can therefore register the receiver with the action in onCreate() and unregister it in onDestroy().

1. Create a member variable in MainActivity for your Receiver:

```
private CustomReceiver mReceiver;
```

2. Get an instance of LocalBroadcastManager and register your receiver with the custom intent action:

```
LocalBroadcastManager.getInstance(this)  
    .registerReceiver(mReceiver, new IntentFilter(ACTION_CUSTOM_BROADCAST));
```

3. Override the onDestroy() method and unregister your receiver from the LocalBroadcastManager:

```
@Override  
protected void onDestroy() {  
    LocalBroadcastManager.getInstance(this).unregisterReceiver(mReceiver);  
    super.onDestroy();  
}
```

## 2.5 Respond to the Custom Broadcast

In your CustomReceiver class, add a case statement for the custom Intent Action, modifying the toast message to "Custom Broadcast Received", and extract your resources:

```
case ACTION_CUSTOM_BROADCAST:  
    toastMessage = context.getString(R.string.custom_broadcast_toast);  
    break;
```

**Note:** Broadcast Receivers that are registered programmatically are not affected by the

enabling or disabling done by the PackageManager class, which is meant for components listed in the Android Manifest file. Enabling or disabling such receivers is done by registering or unregistering them, respectively. In this case, turning off the "Receiver Enabled" toggle will stop the power connected or disconnected Toasts, but not the Custom Broadcast Intent Toast for this reason.

That's it! Your app now delivers custom Broadcast intents and is able to receive both system and custom Broadcasts.

## Coding challenge

A common pattern for broadcast receivers is starting some update or action once the device has booted. Implement a broadcast receiver that will show a Toast message half an hour after the device has booted.

Note: All coding challenges are optional.

## Conclusion

Broadcast Receivers are one of the fundamental components of an android application, with the ability to receive Intents broadcasted by both the system and applications. In order to use a Broadcast Receiver, you must:

- Subclass the BroadcastReceiver class and implement onReceive to process the incoming Intent.
- Register your receiver either in the manifest or programmatically.
- Use LocalBroadcastManager for Broadcasts that are private to your application.
- Avoid putting any long-running tasks in the onReceive() method, instead offload the work to a Service.

## Resources

### Android Developer Documentation

#### Guides

- [Intents and Intent Filters](#)
- [Manipulating Broadcast Receivers On Demand](#)

#### Reference

- BroadcastReceiver

## 7.3 P: Notifications

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [Task 1. Setup the Notify Me! Project](#)
- [Task 2. Update and Cancel your Notification](#)
- [Task 3. Notification Actions](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

Until now, you have been focusing on UI elements that are visible only when your app is running. The only exception to this is the BroadcastReceiver you implemented that showed a Toast when the device was connected or disconnected from power. There are many times when you may want to let your user know of some information even when your application is not running; for example, you might let them know that new content is available, or update them on their favorite team score. Instead of using simple Toast messages, the Android framework provides a convenient mechanism by which to notify your user regarding your application when it is not in the foreground: a [Notification](#).

In this practical you'll create an app that triggers a notification when a button is pressed, and contains the ability to update and cancel your notification.

### What you should already KNOW

- How to implement the `onClick()` method for buttons.
- How to create Implicit Intents.
- How to send Custom Broadcast Intents.
- How to use Broadcast Receivers.

### What you will LEARN

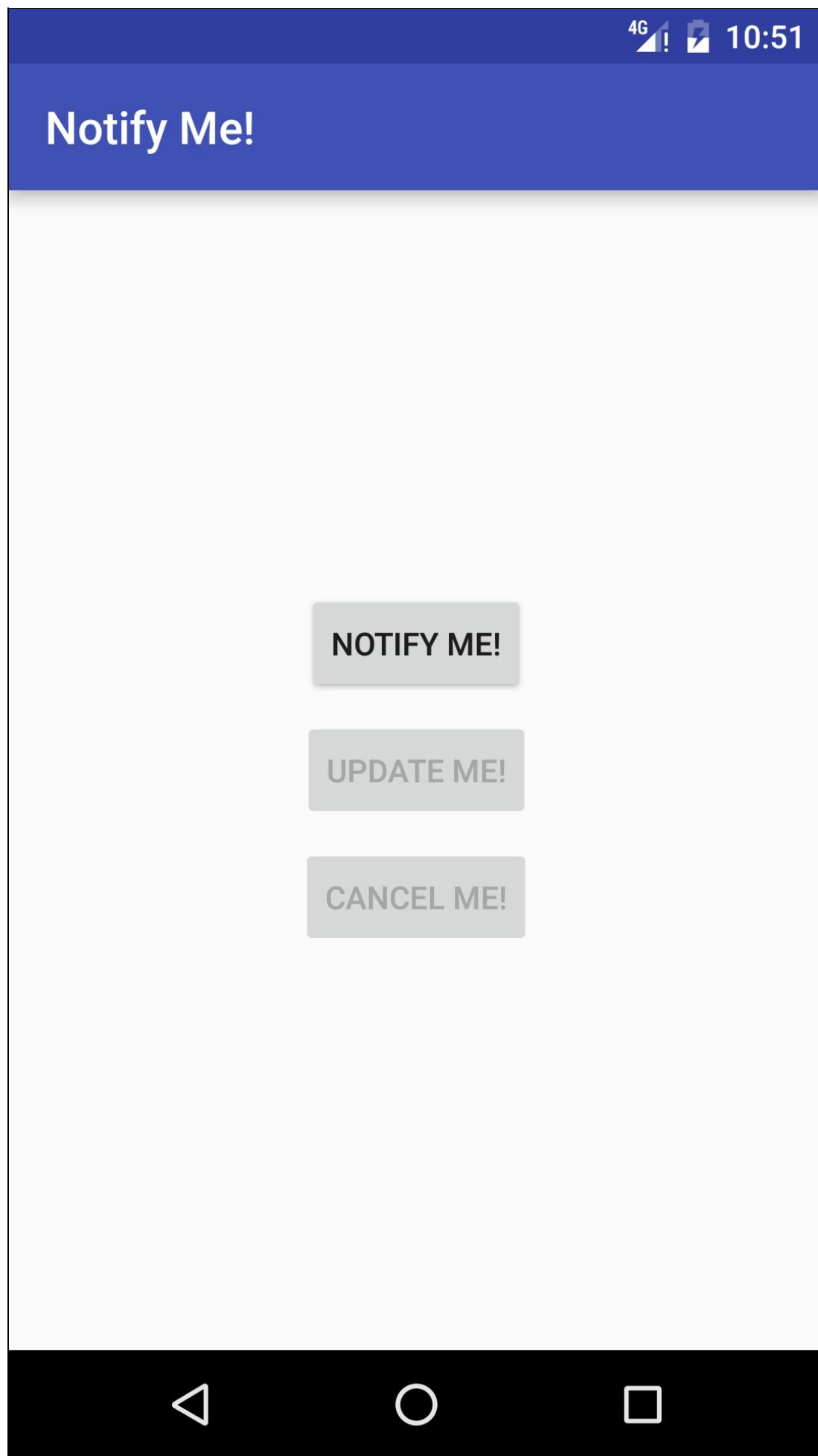
- How to create a Notification using the Notification Builder.
- How to use Pending Intents to respond to Notification actions.
- How to update or cancel existing Notifications.

## What you will DO

- Send a notification when a button is pushed.
- Update the notification both from a button, and an action located in the notification.
- Launch an implicit intent to a web page from the notification.

## App Overview

Notify Me! is an application that can trigger, update and cancel a notification. It also experiments with notification styles, actions and priorities.



# Task 1. Setup the Notify Me! Project

## 1.1 Create the Project

To begin, create a new project called "Notify Me!", accept the default options and use the empty template . Then do the following:

1. In your activity\_main.xml file, change the rootview element to a vertical LinearLayout with it's gravity attribute set to "center".
2. Add a button with the following attributes:

Attribute	Value
android:id	"@+id/notify"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:text	"Notify Me!"
android:layout_margin	"4dp"
android:onClick	"sendNotification"

3. Press **Alt + Enter** on the onClick method value and implement the method in MainActivity.

## 1.2 Build your first Notification

Notifications are created using the [NotificationCompat.Builder](#) class, which allows you to set the content and behavior of the Notification. A notification must contain the following elements:

- A title, set by `setContentTitle()`.
- Detail text, set by `setContentText()`.
- An icon, set by `setSmallIcon()`.

It is then deployed by the [NotificationManager](#), using a notification ID (used to update or cancel you notification in the future) and the notification object you create (calling `build()` on the [NotificationCompat.Builder](#) class).

Do the following:

1. Go to **File > New > Vector Asset**.
2. Click on **Choose** to select a material icon that you will use as the icon for your

notification. In this example, you can use the Android icon.

3. Rename the resource ic\_android and click OK.
4. Open up the new drawable resource and change the color to white by changing the fillColor attribute to "#FFFFFF".
5. In your MainActivity class, create two member variables to store the Notification Builder and Manager:

```
private NotificationCompat.Builder mNotifyBuilder;  
private NotificationManager mNotifyManager;
```

6. Create a constant variable for the Notification id. Since there will be only one active notification at a time, we can use the same id for all notifications:

```
private static final int NOTIFICATION_ID = 0;
```

7. Instantiate the NotificationManager in onCreate using getSystemService():

```
mNotifyManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
```

8. Instantiate the Notification Builder in the sendNotification() method:

```
mNotifyBuilder = new NotificationCompat.Builder(this)
```

9. Set the Notification Title to "You've been notified!".

10. Set the Notification Text to "This is your notification text."

11. Set the Notification icon to the android icon you added.

```
mNotifyBuilder = new NotificationCompat.Builder(this)  
.setContentTitle("You've been notified!")  
.setContentText("This is your notification text.")  
.setSmallIcon(R.drawable.ic_android);
```

12. Call notify() on the NotificationManager at the end of the sendNotification() method, passing in the notification id and the notification:

```
mNotifyManager.notify(NOTIFICATION_ID, mNotifyBuilder.build());
```

13. Run your app. The "Notify me!" button now issues a notification, but it is missing some essential features: there is no notification sound or vibration, clicking on the notification does not do anything. Let's add some functionality to the notification.

## 1.3 Add a Content Intent

In order to improve your notification, you will add a few more features available through the `NotificationCompat.Builder` class:

- A content intent, which is launched when the notification is tapped, set by `setContentIntent()`.
- Priority, which determines how the system displays the notification with respect to other notifications, set by `setPriority()`.
- The default options, such as sounds, vibration and LED lights (if available), set by `setDefaults()`.

Intents for notifications are very similar to the Intents you've been using throughout this course: they can be explicit intents to launch an activity, implicit intents to perform an action, or broadcast intents to notify the system of something. The major difference is that they must be wrapped in a [PendingIntent](#), which allows the notification to perform the action even if your application is not running. In effect, it authorizes the notification to send the intent on the application's behalf.

A `PendingIntent` is created by calling one of the following methods, depending on the type of intent it is meant to contain:

- `getActivity()` if the intent has a specific target (it can be both explicit or implicit).
- `getBroadcast()` if it contains a broadcast intent.
- `getService()` if the intent is meant to start a Service.

All of these methods require the following arguments:

1. The context from which the `PendingIntent` is set up.
2. A `requestCode` integer, that allows you to obtain the `PendingIntent` later on (for cancelling or updating)
3. The Intent it is meant to take over.
4. A flag that determines the behaviour of the `PendingIntent` if another one is issued.

For this example, the content intent of the notification (that is, the one that is launched when the notification is pressed) will simply launch the `MainActivity` of the application.

5. Create an explicit Intent to launch the `MainActivity` class:

```
Intent notificationIntent = new Intent(this, MainActivity.class);
```

6. Get a `PendingIntent` using `getActivity()`, passing in the notification id constant for the `requestCode` and using the `FLAG_UPDATE_CURRENT` flag:

```
PendingIntent notificationPendingIntent = PendingIntent
    .getActivity(this, NOTIFICATION_ID, notificationIntent, PendingIntent.
FLAG_UPDATE_CURRENT);
```

7. Add the PendingIntent to the Notification using setContentIntent():

```
.setContentIntent(notificationPendingIntent)
```

## 1.4 Add priority and defaults to your Notification

When your user clicks the "Notify Me!" button, the notification is issued but the only thing that the user sees is the icon in the notification bar. In order to catch the user's attention, the notification defaults and priority must be properly set.

Priority is in an integer value from PRIORITY\_MIN (-2) to PRIORITY\_MAX (2) that represents how important your notification is to the user. Notifications with a higher priority will be sorted above lower priority ones in the notification drawer. Furthermore, HIGH or MAX priority notifications will be delivered as "Heads - Up" Notifications, which drop down on top of the user's active screen.

Add the following line to the Notification Builder to set the priority of the notification to HIGH:

```
.setPriority(NotificationCompat.PRIORITY_HIGH)
```

The defaults option in the Builder is used to set the sounds, vibration, and LED color pattern for your notification (if the user's device has a LED indicator). In this example, we will use the default options by adding the following line to your Builder:

```
.setDefaults(NotificationCompat.DEFAULT_ALL)
```

## Task 2. Update and Cancel your Notification

After issuing a notification, it is useful to be able to update or cancel the notification if the information changes or becomes no longer relevant. Android also allows you to add actions to your notifications that can send other Intents than the one you used as your "Content Intent". In this task, you will learn how to update and cancel your notification, as well as include notification actions.

### 2.1 Add update and cancel buttons

When adding update and cancel functionality, it is important not to confuse the user and disable any functionality that doesn't make sense in a given context. For example, the update and cancel notification buttons should not be enabled if the notification is has not been delivered. Do the following:

1. In your layout file, create two copies of the "Notify Me!" button.
2. Change the text attribute in the copies to "Update Me!" and "Cancel Me!".
3. Change the id's to "update" and "cancel", respectively.
4. Change the onClick attributes to updateNotification, and cancelNotification.
5. Create method stubs for both of these methods.
6. Add a member variable for each of the three buttons and initialize them in onCreate().

It is time to set up the logic for enabling and disabling the various buttons depending on the state of notification. When the app is first run, the "Notify Me!" button should be the only one enabled as there is no notification yet to update or cancel. After a notification is sent, the cancel and update buttons should be enabled, and the notification button should disabled since it has already been delivered. After the notification is updated, the update and notify buttons should be disabled, leaving only the cancel button enabled. Finally, if the notification is cancelled, the buttons should return to the initial condition with the notify button being the only one enabled. Here is the enabled state toggle code for each method:

#### **onCreate():**

```
mNotifyButton.setEnabled(true);  
mUpdateButton.setEnabled(false);  
mCancelButton.setEnabled(false);
```

#### **sendNotification():**

```
mNotifyButton.setEnabled(false);  
mUpdateButton.setEnabled(true);  
mCancelButton.setEnabled(true);
```

#### **updateNotification():**

```
mNotifyButton.setEnabled(false);  
mUpdateButton.setEnabled(false);  
mCancelButton.setEnabled(true);
```

#### **cancelNotification():**

```
mNotifyButton.setEnabled(true);  
mUpdateButton.setEnabled(false);  
mCancelButton.setEnabled(false);
```

## 2.2 Implement the cancel and update notification methods

Cancelling a notification is very straightforward: you simply call `cancel()` on the `NotificationManager`, passing in the notification id:

```
mNotifyManager.cancel(NOTIFICATION_ID);
```

Updating a notification is a more involved topic. As your application accumulates information for the user, an amateur developer may continue to issue new notifications, crowding the notification tray. It is far better practice to update the existing notification, and to be as compact as possible.

Android notifications come with alternative styles that can help condense information or represent it differently. For example, the Gmail app uses "InboxStyle" notifications if there is more than a single unread message, condensing the information into a single notification.

In this example, you will update your notification to use the "BigPictureStyle" notification, which allows you to include an image in your notification. Do the following:

1. Find an image you want to put in your notification. If you can't think of any, visit [Androidify](#) and make yourself an avatar. Then download the gif and convert it to a .png file.
2. Put this image in the res/drawables folder.
3. In your `updateNotification()` method, convert your `Drawable` into a `Bitmap`:

```
Bitmap androidImage = BitmapFactory.decodeResource(getResources(), R.drawable.masco  
t_1);
```

4. Change the style of your notification using the same Builder as before, setting the image and the "Big Content Title":

```
mNotifyBuilder  
.setStyle(new NotificationCompat.BigPictureStyle()  
.bigPicture(androidImage)  
.setBigContentTitle("Notification Updated!"));
```

5. Change the priority to the default, so that you don't get another heads up notification when it is updated (heads up notifications can only be shown in the default style).

```
.setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

6. Call `notify()` on the Notification Manager, passing in the same notification id as before.

```
mNotifyManager.notify(NOTIFICATION_ID, mNotifyBuilder.build());
```

7. Run your app. After clicking update, check the notification again. It now has the image and an updated title! You can shrink back to the regular notification style by pinching on the extended one.

## Task 3. Notification Actions

Sometimes, a notification requires immediate action: Snoozing an alarm, replying to a text message, etc. You could make the actions available in your app, and the user could tap your notification to arrive at the proper activity, but this takes them out of what they were doing and requires the system to load the activity and perhaps functionality that the user does not need. The notification framework lets you embed actions directly in the notification, allowing the user to act on the notification without opening your application.

The components needed for an action are:

- An icon, to be placed in the notification.
- A label string, placed next to the icon.
- A PendingIntent, to be sent when the action is clicked.

For this example, you will add two actions to your notification: a "Learn More" action with an implicit intent, and a "Update" action with a broadcast intent.

### 3.1 Implement the "Learn More" action

As a first example of notification actions, you will implement one that launches an implicit intent to open a website. Do the following:

1. Create a member String variable that contains the URL to the [Notification design guide](#).
2. In the `sendNotification()` method, create an implicit Intent that opens the saved URL.
3. Create a PendingIntent from the implicit intent, using the flag `FLAG_ONE_SHOT` so that the PendingIntent cannot be reused:

```
Intent learnMoreIntent = new Intent(Intent.ACTION_VIEW, Uri.parse(NOTIFICATION_GUIDE_URL));
PendingIntent learnMorePendingIntent = PendingIntent.getActivity(this, NOTIFICATION_ID, learnMoreIntent, PendingIntent.FLAG_ONE_SHOT);
```



4. Download this vector icon using the Vector Asset Studio, and call it ic\_more:
  5. Change the color of the icon to white.
  6. Add the following line of code to your builder to add the action:
- ```
.addAction(R.drawable.ic_learn_more, "Learn More", learnMorePendingIntent);
```
7. Run your app. Your notification will now have a clickable icon that takes you to the web!

## 3.2 Notifications and Broadcast Receivers

You've seen that a notification action uses a PendingIntent to respond to user interaction. In the last step, you added an action that uses an implicit intent turned into a PendingIntent using the `getActivity()` method. You can also deliver a broadcast intent by calling `getBroadcast()` on the PendingIntent. Broadcast Intents are very useful in notifications, since a broadcast receiver can register the intent and respond accordingly, without launching a specific activity.

You will now implement a broadcast receiver that will run the `updateNotification()` method when the "Update" action in the notification is pressed. This is a common pattern: adding functionality to a notification that already exists in the app, so that the user does not need to launch the app to perform the action. Do the following:

1. Subclass a BroadcastReceiver as an inner class in `MainActivity` and override the `onReceive()` method. Don't forget to include an empty constructor:

```
public class NotificationReceiver extends BroadcastReceiver {

    public NotificationReceiver() {
    }

    @Override
    public void onReceive(Context context, Intent intent) {
    }
}
```

2. In the onReceive() method, call updateNotification(), passing in null as the argument (since the View parameter is not used).
3. Create a constant member variable in MainActivity to represent the update notification action for your broadcast intent. Make sure it begins with your package name to insure its uniqueness:

```
private static final String ACTION_UPDATE_NOTIFICATION = "com.ngamolsky.android.notifyme.ACTION_UPDATE_NOTIFICATION";
```

4. Create a member variable for you receiver and initialize it using the empty constructor in onCreate().
5. In the onCreate() method, register your broadcast receiver:

```
registerReceiver(mReceiver, new IntentFilter(ACTION_UPDATE_NOTIFICATION));
```

6. Override the onDestroy() method and unregister your receiver:

```
@Override  
protected void onDestroy() {  
    unregisterReceiver(mReceiver);  
    super.onDestroy();  
}
```

**Note:** In this example you are registering your broadcast receiver programmatically because your receiver is defined as a non-static, inner class. Receivers defined this way cannot be registered in the Manifest since they have the possibility of changing.

Although at first it may seem that the broadcast sent by the notification only concern your app, and therefore should be delivered with a LocalBroadcastManager, the use of PendingIntents delegates the responsibility of delivering the notification to the Android framework, and therefore LocalBroadcastManager can not be used. </div>

1. Create a broadcast Intent in the sendNotification() method using the custom update action.
2. Get a PendingIntent using getBroadcast():

```
Intent updateIntent = new Intent(ACTION_UPDATE_NOTIFICATION);  
PendingIntent updatePendingIntent = PendingIntent.getBroadcast(this, NOTIFICATION_ID, updateIntent, PendingIntent.FLAG_ONE_SHOT);
```

3. Download this vector icon using the Vector Asset Studio, call it ic\_update, and make it

white.



4. Add the action to the builder in the sendNotification() method, giving it the title "Update":

```
.addAction(R.drawable.ic_update, "Update", updatePendingIntent)
```

5. It does not make sense to include the update action in the already updated notification. Modify the actions in the updateNotification() method to only show the "Learn More" action:

```
mNotifyBuilder.addAction("Learn More", R.drawable.ic_update, PendingIntent.FLAG_UPDATE_CURRENT);
```

6. Run your app. You can now update your notification without opening the app!

## Coding challenge

Note: All coding challenges are optional.

Enabling and Disabling the various buttons provides an intuitive user experience, without the possibility of trying to update a notification before one exists. However, there is one use case in which the state of your buttons does not match the state of the notification: if a user dismisses the notification, by swiping it away or clearing the whole notification drawer. In this case, your app has no way of knowing that the notification was cancelled, and that the button state must be changed.

Create another broadcast intent that will let the application know that the user has dismissed the notification, and toggle the button states accordingly.

Hint: Check out the [NotificationCompat.Builder](#) class for a method that delivers an Intent when the notification has been dismissed by the user.

## Conclusion

Notifications provide an interface between your app and the user, even when the app is not running.

The major components of a notification are:

Required:

- A small icon using `setSmallIcon()`.
- A title using `setContentTitle()`.
- Some detail text using `setContentText()`.

Optional:

- A content intent, launched when the notification is tapped
- Actions containing `PendingIntents`
- Expanded Styles
- Priority
- Defaults
- And many more.. Check out the [NotificationCompat.Builder](#) documentation for all notification options.

## Resources

### Android Developer Documentation

#### Guides

- [Notifications](#)
- [Notification Design Guide](#)

#### Reference

- [NotificationCompat.Builder](#)
- [NotificationCompat.Style](#)

# 8.1 P: Alarm Manager

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [Task 1. Setup the Parking Alarm Project](#)
- [Task 2. Set up your notifications](#)
- [Task 3. Set up your alarms](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

Most of the time, your app only needs to respond to user interaction: pushing a button, tapping a notification, etc.. You've also learned how to respond to system events using BroadcastReceivers. But what if your app needs to respond at a specific time, as with a calendar notification? Enter [AlarmManager](#): a class that allows you to launch a PendingIntent (like the ones you used with notifications) at a specific time, with the possibility of repeating the task at a given interval.

Waking the device up at a specific time has its drawbacks: the framework can not compensate by waiting for the appropriate network connection or battery status and therefore can be resource intensive. It is **not** recommended to use AlarmManager unless the specific timing of your process is important.

That being said, AlarmManager includes some methods that help alleviate the inefficiency of specific timing: it is able to batch tasks together when the exact timing is not important. It is crucial for the developer to use the least precise timing possible in order to make the AlarmManager the most efficient it can be.

For most tasks, such as updating the weather information or news stories, it can weight until conditions are more favorable, such as being connected to wifi and charging. For these cases, the [JobScheduler](#) should be used, which you will learn about in the following lesson.

In this practical, you will create a parking meter timer that will use an AlarmManager to schedule a notification when your timer is about to run out.

## What you should already KNOW

- How to implement the onClick() method for buttons.
- How to send custom broadcast Intents.
- How to use broadcast receivers.
- How to implement Spinners.
- How to send notifications.

## What you will LEARN

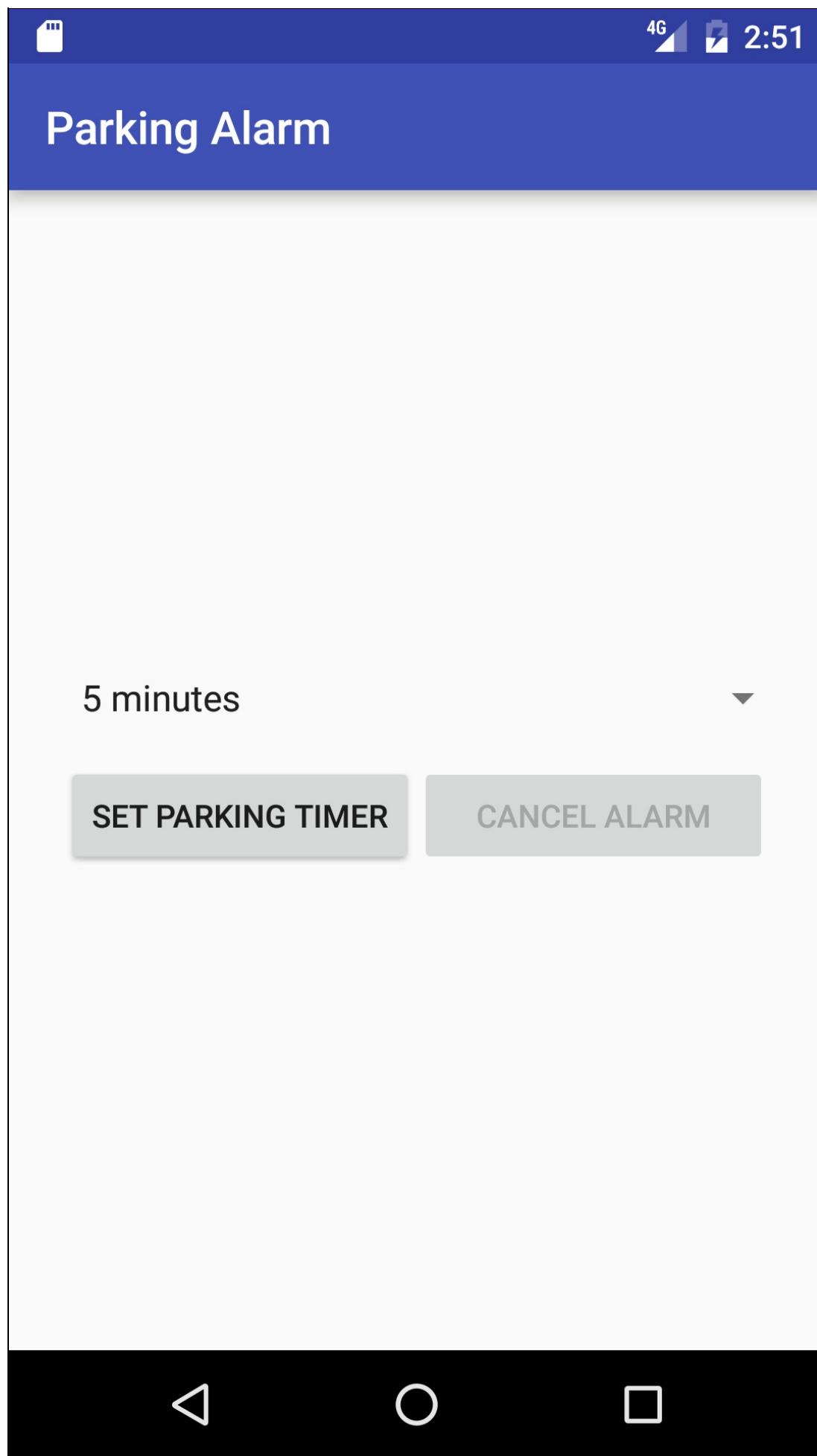
- How to schedule single alarms with AlarmManager.
- How to schedule repeating alarms with AlarmManager.
- How to cancel a repeating alarm.

## What you will DO

- Set an inexact repeating alarm to keep track of a parking meter timer using a spinner.
- Set a single alarm to trigger a notification when the parking meter is expired.
- Add a cancel button to both the notification and the activity to cancel the alarm.

## App Overview

Parking Alarm is an application in which the user can set a timer for the common parking time limits. The app will then issue an ongoing, low priority notification that keeps track of the remaining time and is updated periodically. Once the timer is about to run out, the app will issue a high priority notification to let the user know.



# Task 1. Setup the Parking Alarm Project

## 1.1 Create the Project

To begin, create a new project called "Parking Alarm", accept the default options and use the empty template . Then do the following:

1. In your activity\_main.xml file, change the rootview element to a vertical LinearLayout with it's gravity attribute set to "center".
2. Add a Spinner with the following attributes:

| Attribute             | Value                  |
|-----------------------|------------------------|
| android:id            | "@+id/durationSpinner" |
| android:layout_width  | "match_parent"         |
| android:layout_height | "wrap_content"         |
| android:layout_margin | "8dp"                  |

3. Create another LinearLayout beneath the Spinner view:

| Attribute             | Value          |
|-----------------------|----------------|
| android:orientation   | "horizontal"   |
| android:layout_width  | "match_parent" |
| android:layout_height | "wrap_content" |
| android:layout_margin | "8dp"          |

4. Create two buttons with the following attributes:

| Attribute             | Value          |
|-----------------------|----------------|
| android:layout_width  | "0dp"          |
| android:layout_weight | "1"            |
| android:layout_height | "wrap_content" |

5. Set the text of the first button to "Set Parking Timer", the onClick attribute to "setAlarm", and the id to "setAlarmButton".
6. Set the text of the second button to "Cancel Alarm", the onClick to "cancelAlarm", and the id to "cancelAlarmButton".
7. Implement both of the methods in MainActivity.

## 1.2 Set up the Duration Spinner

For this example, the user will only be able to select one of the preset parking time limits. You will define these preset durations as a string array resource in your string.xml file.

1. In your string.xml file, define a string array resource with the following string items:
  - i. 5 minutes
  - ii. 15 minutes
  - iii. 30 minutes
  - iv. 1 hour
  - v. 2 hours
  - vi. 4 hours
2. In MainActivity, initialize the Spinner view in onCreate().
3. Create an ArrayAdapter of parameterized type CharSequence, using the createFromResource() method, passing in the application context, the string array resource and the simple\_spinner\_item layout:

```
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this, R.array.duration_array, android.R.layout.simple_spinner_item);
```

4. Set the dropdown view to the default dropdown line items by calling setDropDownViewResource() and passing in the simple\_spinner\_dropdown\_item:

```
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
```

5. Set the adapter to the spinner:

```
durationSpinner.setAdapter(adapter);
```

6. Set an onItemSelectedListener on the adapter, creating a new listener and implementing the required methods:

```
durationSpinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
    @Override
    public void onItemSelected(AdapterView<?> adapterView, View view, int i, long l) {

    }

    @Override
    public void onNothingSelected(AdapterView<?> adapterView) {
    }
});
```

## 1.3 Get the selected duration

In order to set the timer, you must first get the selected duration from the user, in milliseconds, and then enable the "Set Parking Timer" button.

1. Create a member variable to store the selected duration in milliseconds.
2. In the `onItemSelected()` callback, get the selected item from the AdapterView:
3. Create a switch statement on the selected string, with a case for each item in the duration array.
4. In each case, set the duration variable to the selected time value, converted to milliseconds using the `TimeUnit` utility class:

```
switch (selection){  
    case "5 minutes":  
        mSelectedDuration = TimeUnit.MILLISECONDS.toMillis(5);  
        break;  
    case "15 minutes":  
        mSelectedDuration = TimeUnit.MILLISECONDS.toMillis(15);  
        break;  
    case "30 minutes":  
        mSelectedDuration = TimeUnit.MILLISECONDS.toMillis(30);  
        break;  
    case "1 hour":  
        mSelectedDuration = TimeUnit.HOURS.toMillis(1);  
        break;  
    case "2 hours":  
        mSelectedDuration = TimeUnit.HOURS.toMillis(2);  
        break;  
    case "4 hours":  
        mSelectedDuration = TimeUnit.HOURS.toMillis(4);  
        break;  
}
```

## Task 2. Set up your notifications

It is now time to think through the logic of the application more thoroughly: the user sets a timer, a low priority notification appears containing the remaining time on the parking meter and an action to cancel the alarm. Just before the timer expires, your app will issue another, high priority notification to let the user know they are out of time.

AlarmManager, like notifications, uses a `PendingIntent` that it delivers with the specified options. From the logic of the application, you can determine that you will need the following Intents:

- A content intent for the notifications, leading to the `MainActivity` when the notification is clicked.
- A cancel broadcast intent, sent when the cancel action in the notification is clicked.
- An alarm pending broadcast intent that updates the alarm pending notification with the

remaining time periodically.

- A final alarm broadcast intent that cancels the alarm pending broadcast and issues the final notification sent right before the parking timer expires.

The broadcast intents will be received in a broadcast receiver which will take the appropriate action (sending notifications or canceling the alarm) based on the Intent it receives.

## 2.1 Create the broadcast receiver

Your broadcast receiver will be responsible for responding to all of the broadcast intents sent by the AlarmManager and notifications. The three actions this represents are:

- Notify the user that an alarm has been set.
- Notify the user than an alarm is about to expire.
- Cancel the alarm and notification.

Do the following:

1. In your MainActivity, create an inner class with the following declaration:

```
public class AlarmReceiver extends BroadcastReceiver {
```

2. Implement the required onReceive() method.

3. Create string constants for each of the actions:

```
private static final String ACTION_NOTIFY_PENDING = "com.example.android.parkingalarm.ACTION_NOTIFY_PENDING";
private static final String ACTION_NOTIFY_FINAL = "com.example.android.parkingalarm.ACTION_NOTIFY_FINAL";
private static final String ACTION_CANCEL = "com.example.android.parkingalarm.ACTION_CANCEL";
```

4. In the onReceive() method, get the action from the incoming intent and create a switch statement on the action.

5. Create a case for each of the three possible actions.

6. Create the following methods in MainActivity that will issue your notifications:

```
private void notifyPending() {}
private void notifyFinal() {}
```

## 2.2 Create your notifications

You will now create the notifications that will be triggered by the Alarm Manager.

1. Create member variables for a NotificationManager and NotificationCompat.Builder.
2. Initialize the NotificationManager in onCreate().

3. Create an integer constant that will be used for both notification id and the PendingIntent request code, and set it equal to 0.
4. Create a member variable for the PendingIntent that will contain the notification content intent, since it will be used for both notifications.
5. Create the content intent and initialize the corresponding PendingIntent in onCreate():
6. Download a vector asset to use as the notification icon. Since this is a parking meter app, the provided material icon of the car should do well: 
7. Change the icon color to white.
8. Create a notification in the notifyFinal() method with the following components (use the passed in Context to create the Builder):

|                |                                  |
|----------------|----------------------------------|
| Content Title  | "Parking Alert!"                 |
| Content Text   | "Parking is about to expire"     |
| Defaults       | NotificationCompat.DEFAULT_ALL   |
| Auto Cancel    | true                             |
| Small Icon     | R.drawable.ic_car                |
| Content Intent | mNotificationPendingIntent       |
| Priority       | NotificationCompat.PRIORITY_HIGH |

**Note:** Auto Cancel is a parameter that determines whether a notification is dismissed when it is clicked (the content intent is activated). It is set using the setAutoCancel() method.

9. Call notify() on the NotificationManager, using the notification id constant and the builder you just created.

For the pending notification, it is a little more complicated. Firstly, a PendingIntent that cancels the notification and alarm is needed for the notification action. Secondly, the content text of the pending notification needs to be updated with the remaining time, which needs to be calculated each time notifyPending() is called. Do the following to set up the pending notification in the notifyPending() method:

10. Create a variable to store the current time.
11. Create a member variable to keep track of the time that the parking meter expires. This variable will be set later on in the setAlarm() method, which will always be called before the notifications are delivered.
12. Create a variable for the time left in the meter by subtracting the current time from the parking expiration time:

```
long currentTime = Calendar.getInstance().getTimeInMillis();
long timeLeft = mParkingExpireTime - currentTime;
```

13. Create an intent with the ACTION\_CANCEL as its action. This will be delivered when the cancel action inside the notification is clicked.
14. Create a PendingIntent from the cancel intent using the getBroadcast() method, using the ONE\_SHOT flag since this PendingIntent won't be reused.
15. Create a string variable for the pending notification content text:

```
String notificationText = getString(R.string.notification_text,
    (TimeUnit.MILLISECONDS.toMinutes(timeLeft)+1));
```

- Note:** There is a minute added to the pending notification text since the seconds are not counted and 4:59 would be approximated as 4 instead of 5.
16. Download an icon to be used for the cancel action. The clear action is usually represented by the cross icon. Change the color of the icon to white.
  17. Use the builder to create a new notification with the following components:

|               |  |
|---------------|--|
| Content Title | "Parking Alert!"                                   |
| Content Text  | notificationString                                 |
| Priority      | NotificationCompat.PRIORITY_LOW                    |
| Ongoing       | true   |
| Small Icon    | R.drawable.ic_car                                  |
| Action        | R.drawable.ic_clear, "Cancel", cancelPendingIntent |

- Note:** Ongoing notifications cannot be dismissed by the user and remain in the notification tray. This feature is useful when there is an ongoing process (in this case, a parking timer) that the user may want to constantly check on until it is finished.
18. Call notify() on the NotificationManager.

## Task 3. Set up your alarms

Now that your user notifications are prepared, it is time to get the main component of your application: the AlarmManager. This is the class that will be used to deliver your notification broadcast intents (both the ongoing and final one) as well as periodically update the ongoing notification. AlarmManager has many kinds of alarms built into it, both one-time and periodic, exact and inexact. To learn more about the different kinds of alarms, look into [this guide](#).

### 3.1 Set up the broadcast intents

AlarmManager delivers a PendingIntent at the time and frequency set in one of the many "set" methods. Since you will set two different alarms, one repeating alarm to update the pending notification and one single alarm to display the final notification, you will need two

different PendingIntents. Do the following:

1. Create member variables for both PendingIntents.
2. Create the Intents in setAlarm() using the ACTION\_NOTIFY\_PENDING and ACTION\_NOTIFY\_FINAL.
3. Get both of the PendingIntents using the getBroadcast() method. Use the FLAG\_ONE\_SHOT for the final alarm intent, and FLAG\_UPDATE\_CURRENT on the pending alarm intent so that the time left on the alarm can be updated.

## 3.2 Implement setAlarm()

You are now ready to implement the setAlarm() method in MainActivity. There will be two different alarms:

- A single alarm that for the final notification.
- A repeating alarm for the pending notification.

Both of these alarms are set using the [AlarmManager](#) class. Create a member variable for the alarm manager class and initialize it in onCreate():

```
mAlarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
```

The single alarm will be set using the appropriately named set() method. On devices that are pre API 19, this will create an exact alarm that will be triggered at the specified time (either based on the real-time clock, or the elapsed time since the last boot). After API 19, these alarms are inexact in order to optimize the resources needed, potentially batching alarms together. For this reason, you will use the setWindow() method instead for devices running API 19+, allowing the system to better batch the alarms together.

1. Store the current time in milliseconds in a variable called currentTime.
2. Set the variable for the parking expiration time by adding the selected duration to the current time:

```
long currentTime = Calendar.getInstance().getTimeInMillis();
mParkingExpireTime = currentTime + mSelectedDuration;
```

3. Create an if block that checks if the API level of the device is greater than API level 19 (Kitkat):

The setWindow() method takes 4 arguments:

- The alarm type, which can be either real-time, or elapsed time since the boot. There is also a wakeup version of both of these alarms that are able to wake up the device if is locked. For this example you will use a real-time wakeup alarm.

- The start time of the window, which you will set to be one minute before the parking expiration time.
- The duration of the window, which will be 30 seconds.
- The pending intent to deliver, in this case the final alarm broadcast intent.

Call the `setWindow()` method on the `AlarmManager` if the device is running API 19+, otherwise call `set()` with the following arguments:

- The same alarm type (real-time wakeup).
- The trigger time, which will be 30 seconds before the parking expired time.
- The final alarm broadcast pending intent.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
    mAlarmManager.setWindow(AlarmManager.RTC_WAKEUP, mParkingExpireTime - TimeUnit.MINUTES.toMillis(1), TimeUnit.SECONDS.toMillis(30), mAlarmFinalIntent);
} else {
    mAlarmManager.set(AlarmManager.RTC_WAKEUP, mParkingExpireTime - TimeUnit.SECONDS.toMillis(30), mAlarmFinalIntent);
}
```

The second alarm is a repeating alarm, which is meant to update the remaining time in the pending notification. It is not important that the alarm be exact, since the alarm will be set to repeat every minute, and a few seconds of difference is acceptable for the pending alarm (but the final notification will always be on time). The second alarm will therefore use the `setInexactRepeating()` method. Do the following:

- Call `setInexactRepeating()` on the `AlarmManager`, starting the repeating alarm at the time the `setButton()` is pushed:

```
mAlarmManager.setInexactRepeating(AlarmManager.RTC_WAKEUP, currentTime, TimeUnit.MILLISECONDS.toMillis(1), mAlarmPendingIntent);
```

### 3.3 Implement cancelAlarm()

Cancelling the alarm is straightforward: each of the existing alarms need to be canceled, using the `PendingIntent` they are meant to deliver and the alarm manager's `cancel()` method. You must then remove the notification:

```
public void cancelAlarm(View view) {
    mAlarmManager.cancel(mAlarmPendingIntent);
    mAlarmManager.cancel(mAlarmFinalIntent);
    mNotificationManager.cancel(NOTIFICATION_ID);
}
```

### 3.4 Finish the broadcast receiver

Once the alarms are set, the application will begin delivering your pending alarm intent close to every minute. This repeating broadcast should trigger the broadcast receiver to call `notifyPending()`. Right before you timer expires, the application will send your final alarm broadcast, which should cancel the repeating alarm and display the final alarm notification.

Do the following:

1. In the `notifyFinal()` method, call `cancel()` on your `AlarmManager`, passing in the pending intent for the ongoing repeated alarm:

```
mAlarmManager.cancel(mAlarmPendingIntent);
```

2. Fill in each case statement in the `onReceive` method, calling the appropriate method depending on the incoming action:

```
@Override  
public void onReceive(Context context, Intent intent) {  
    switch (intent.getAction()) {  
        case ACTION_NOTIFY_PENDING:  
            notifyPending();  
            break;  
        case ACTION_NOTIFY_FINAL:  
            notifyFinal();  
            break;  
        case ACTION_CANCEL:  
            cancelAlarm(null);  
            break;  
    }  
}
```

## 3.5 Final Steps

Your parking timer app is almost complete. After running the app, you may notice a few things that are not quite right:

- When the alarm is set, the ongoing notification does not show up immediately. This is because the alarm used to post this notification uses inexact timing (you called `setInexactRepeating`) and although it is meant to start immediately, in practice the system may wait to deliver your broadcast. For this reason, you should trigger the notification immediately when the `setAlarm()` button is pressed, allowing the alarm to simply update the time left in the notification where the timing is not as important:
- To the user, there is no feedback when the alarm is set or when the alarm is canceled. The buttons can be pressed any number of times, and it is not clear whether this creates multiple alarms. In order to clarify this behavior for the user, as well as limit the application to a single parking timer at a time, you can toggle the enabled state of the

buttons. Do the following:

1. In the activity\_main.xml file, disable both of the buttons.
2. Enable the "Set Parking Timer" button once a duration is selected from the spinner.
3. Once the alarm is set, disable the "Set Parking Timer" button and enable the "Cancel Alarm" button.
4. In the cancelAlarm() method, reset the state of the buttons: enabled for the "Set Parking Timer" button and disabled for the "Cancel Alarm" button.

## Coding challenge

Note: All coding challenges are optional.

The AlarmManager class also handles alarm clocks in the usual sense, the kind that wake you up in the morning. On devices running API 21+, you can get information about the next alarm clock of this kind by calling getNextAlarmClock() on the alarm manager.

Add a button to your application that displays the time of next alarm clock that the user has set in a toast message.

## Conclusion

AlarmManager allows you to manage the timing of an intent by using one of its set() methods. It allows for these intents to be delivered using the PendingIntent framework, and can be set to rely on the real time clock or else the elapsed time since boot. This flexibility can be dangerous, as precise timing is resource heavy and is prone to draining battery. You should therefore use the built-in inexact timing mechanism, whereby the alarm manager can batch tasks together for greater efficiency.

## Resources

### Android Developer Documentation

#### Guides

- [Scheduling Repeating Alarms](#)

#### Reference

- [AlarmManager](#)

## Other Web resources

- Blog Post on choosing the correct alarm type

## 8.2 P: Job Scheduler

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [Task 1. Setup the PowerReceiver Project](#)
- [Task 2. Send and Receive a Custom Broadcast](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

You've seen that you can trigger events based on the real-time clock, or the elapsed time since boot using the `AlarmManager` class. Most tasks, however, do not require an exact time, but should rather be scheduled based on a combination of system requirements and your application logic. For example, a news app might like to have updated news ready in the morning, but could wait until the device is charging and connected to wifi, in order to preserve the user's data and system resources.

The `JobScheduler` class is meant precisely for this kind of scheduling: it allows you to set the parameters of your task, and calculates the best time to schedule it. The task to be run is implemented in a `JobService` class, and executed according to the added constraints.

`JobScheduler` is only available on devices running API 21+, and is currently not available in the support library. For backward compatibility, use the `GCMNetworkManager` (soon to be `FirebaseJobDispatcher`).

In this practical, you will create an app which schedules a notification to be posted when the user set parameters are fulfilled, and the system requirements are met.

### What you should already KNOW

- How to deliver a notification.
- How to get a integer value from a Spinner view.
- How to use Switch views for user input.
- How to create PendingIntents.

### What you will LEARN

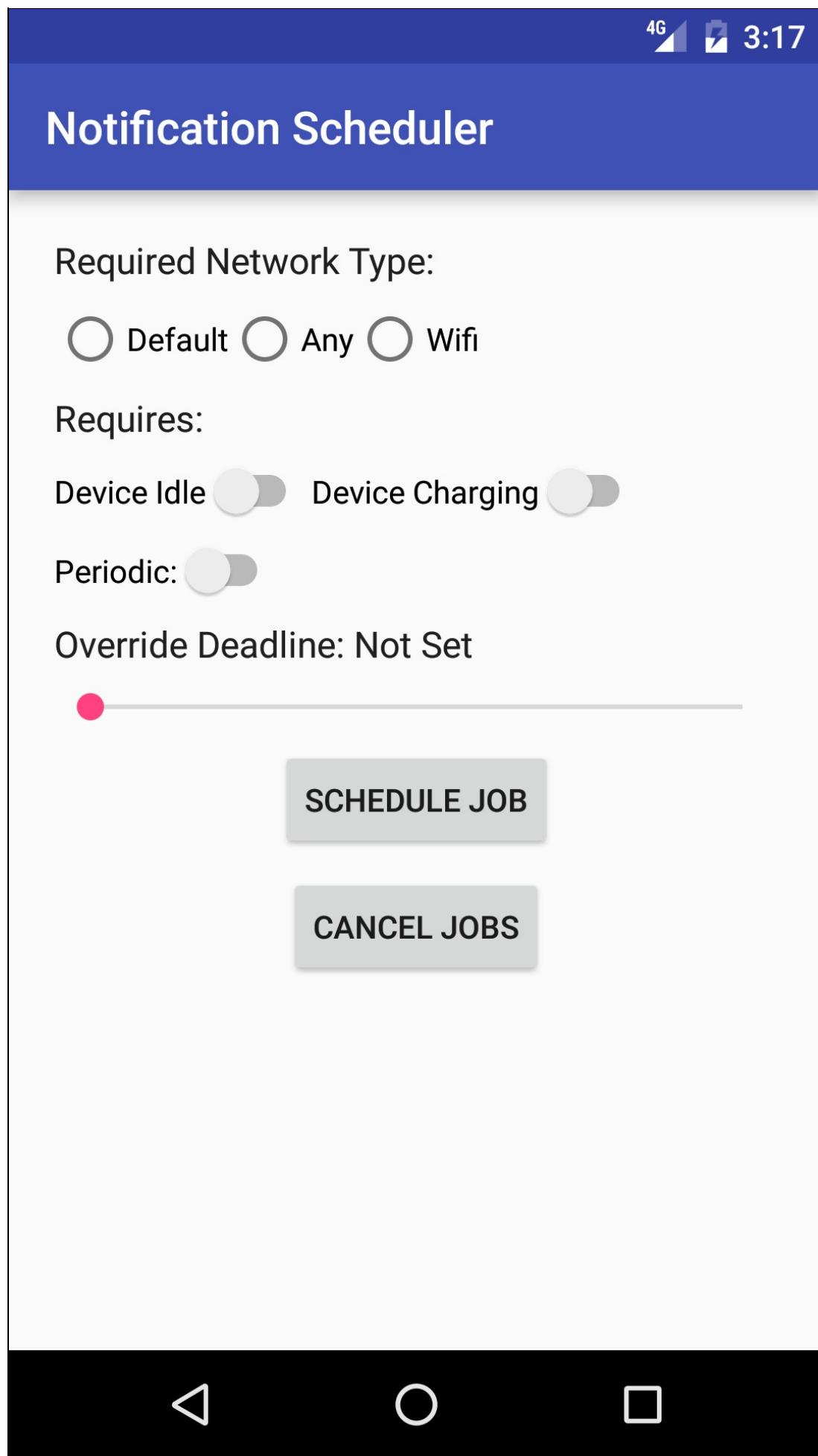
- How to implement a JobService.
- How to construct a JobInfo object with specific constraints.
- How to schedule a JobService based on the JobInfo object.

## What you will DO

- Implement a JobService that delivers a notification.
- Get user input to configure the constraints on the JobService you are scheduling.
- Schedule the job using JobScheduler.

## App Overview

Notification Scheduler demonstrates the JobScheduler framework by allowing the user to select constraints and schedule the job, which posts a notification when it is executed. The user will therefore be able to observe the timing of JobScheduler based on the explicit constraints.



# Task 1. Implement a JobService

To begin with, you must create a service that will be run at the time determined by the constraints. The JobService is automatically executed by the system, and the only parts you need to implement are:

- The onStartJob() callback, which is called when the system determines that your task should be run. This is where the job to be done is implemented.

**Note:** onStartJob() is executed on the main thread, and therefore any long-running tasks must be offloaded to a different thread. In this case, you are simply posting a notification, which can be done on the main thread.

- The onStopJob(), which is called if the constraints are no longer met while your work is being performed, meaning that the job must be stopped.

Both of these callbacks return boolean values that determine their behavior.

- For onStartJob(), the boolean indicates whether the job is fully completed in the onStartJob() callback. If true, the work is offloaded to a different thread, and you must call jobFinished() to indicate that the job is complete. If false, the framework knows that the job is completed by the end of onStartJob() and it will automatically call jobFinished() on your behalf.
- For onStopJob(), the boolean determines the retry policy in the case where the job cannot finish due to changing constraints. If true, the job will be rescheduled, otherwise, it will be dropped.

## 1.1 Create the Project and the NotificationJobService

To begin, create a new project called "Notification Scheduler", make sure to select API 21 as the minimum SDK (JobScheduler won't work before then) and use the empty template .

Then do the following:

1. Create a new Java class called NotificationJobService and make it extend JobService.
2. Implement the required methods: onStartJob() and onStopJob().
3. Navigate to your AndroidManifest.xml file and register your JobService with the following permission:

```
<service  
    android:name=".NotificationJobService"  
    android:permission="android.permission.BIND_JOB_SERVICE"/>
```

## 1.2 Implement onStartJob()

1. In onStartJob(), get the context of the application and store it in a variable.
2. Download a notification icon to be used with the "Job Running" notification.
3. Create a PendingIntent to launch the MainActivity of your app in onStartJob(), to be used as the content intent for your notification.
4. In onStartJob(), construct and deliver a notification with the following attributes:

|                |                                  |
|----------------|----------------------------------|
| Content Title  | "Job Service"                    |
| Content Text   | "Your Job is running!"           |
| Content Intent | contentPendingIntent             |
| Small Icon     | R.drawable.ic_job_running        |
| Priority       | NotificationCompat.PRIORITY_HIGH |
| Defaults       | NotificationCompat.DEFAULT_ALL   |
| AutoCancel     | true                             |

5. Make sure onStartJob() return false, since all of the work is completed in that callback;
6. Make onStopJob() return true, so that the job is rescheduled if it fails.

## Task 2. Implement the constraints

Now that you have your JobService, it is time to determine the constraints the system will use to schedule it. To begin, will create a group of radio buttons to determine the network type required for this job.

### 2.1 Implement the network constraint

One of the available constraints is a network required constraint: limiting the JobService to be executed only when the network conditions are met. The options are:

- Default, which will run in any network conditions.
- Any, which will run as long as a network is available.
- Unmetered, which will run as long as you are connected to wifi.

Do the following:

1. In your activity\_main.xml file, change the rootview element to a vertical LinearLayout.
2. Change the TextView to have the following attributes:

| Attribute              | Value                                     |
|------------------------|---|
| android:layout_width   | "wrap_content"                            |
| android:layout_height  | "wrap_content"                            |
| android:text           | "Network Type Required: "                 |
| android:textAppearance | "@style/TextAppearance.AppCompat.Subhead" |
| android:layout_margin  | "4dp"                                     |

3. Add a RadioGroup container element below the TextView with the following attributes:

| Attribute             | Value                 |
|-----------------------|-----------------------|
| android:layout_width  | "wrap_content"        |
| android:layout_height | "wrap_content"        |
| android:orientation   | "horizontal"          |
| android:id            | "@+id/networkOptions" |
| android:layout_margin | "4dp"                 |

Note: Using a radio group ensures that only one of its children can be selected at a time. For more information on Radio Buttons see [this guide](#).

4. Add three RadioButtons as children to the RadioGroup with their layout height and width set to "wrap\_content" and the following text and id's:

|                      |                       |
|----------------------|-----------------------|
| <b>RadioButton 1</b> |                       |
| android:text         | "Default"             |
| android:id           | "@+id/defaultNetwork" |
| <b>RadioButton 2</b> |                       |
| android:text         | "Any"                 |
| android:id           | "@+id/anyNetwork"     |
| <b>RadioButton 3</b> |                       |
| android:text         | "Wifi"                |
| android:id           | "@+id/wifiNetwork"    |

5. Add two buttons below the radio button group with height and width set to "wrap content" with the following attributes:

|                        |                     |
|------------------------|---------------------|
| <b>Button 1</b>        |                     |
| android:text           | "Schedule Job"      |
| android:onClick        | "scheduleJob"       |
| android:layout_gravity | "center_horizontal" |
| android:layout_margin  | "4dp"               |
| <b>Button 2</b>        |                     |
| android:text           | "Cancel Jobs"       |
| android:onClick        | "cancelJobs"        |
| android:layout_gravity | "center_horizontal" |
| android:layout_margin  | "4dp"               |

6. Implement both of the `onClick()` methods in `MainActivity`.
7. Create a member variable for the `JobScheduler`, and initialize it in `scheduleJob()` using `getSystemService()`:

```
mScheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
```

8. In `scheduleJob()`, find the `RadioGroup` by id.
9. Create a class constant for the job id, and set it equal to 0 (you will only have one job at time).
10. Create a `JobInfo.Builder` object in `scheduleJob()` passing in the job id and the `ComponentName` for the `JobService` you created:

```
JobInfo.Builder builder = new JobInfo.Builder(JOB_ID, new ComponentName(getPackageName(), NotificationJobService.class.getName()))
```

11. Call `setRequiredNetworkType()` on the `JobInfo.Builder` object, passing in the id of the checked radio button:

```
.setRequiredNetworkType(networkOptions.getCheckedRadioButtonId());
```

12. Call `schedule()` on the `JobScheduler` object, passing in the `JobInfo` object with the `build()` method:

```
mScheduler.schedule(builder.build());
```

13. Show a `Toast` message, letting the user know the job was scheduled.
14. In the `cancelJobs()` method, check if the `JobScheduler` object is null, and if not, call `cancelAll()` on it to remove all pending jobs, reset the `JobScheduler` to be null, and show a `Toast` message to let the user know the job was canceled:

```

if (mScheduler!=null){
    mScheduler.cancelAll();
    mScheduler = null;
    Toast.makeText(MainActivity.this, "Jobs Canceled", Toast.LENGTH_SHORT).show();
}

```

- Run the app, you can now set tasks with network restrictions and see how long it takes for them to be executed.

## 2.2 Implement the Device Idle and Device Charging constraints

JobScheduler includes the ability to wait until the device is charging, or in an idle state (screen off, CPU gone to sleep) to execute your JobService. You will now add switches to your app to toggle these constraints on your JobService. Do the following:

- In your activity\_main.xml file, copy the network type label TextView and paste it below the RadioGroup.
- Change the `android:text` attribute to "Requires:".
- Below this textview, insert a horizontal LinearLayout with a 4dp margin.
- Create two Switch views as children to the horizontal LinearLayout with height and width set to "wrap\_content" and the following attributes:

| <b>Switch 1</b> |                       |
|-----------------|-----------------------|
| android:text    | "Device Idle"         |
| android:id      | "@+id/idleSwitch"     |
| <b>Switch 2</b> |                       |
| android:text    | "Device Charging"     |
| android:id      | "@+id/chargingSwitch" |

- Create member variables for the switches and initialize them in `onCreate()`.
- In the `scheduleJob()` method, add the following methods to the `JobInfo.Builder` to set the constraints on the JobScheduler based on the user selection in the switches:

```

.setRequiresDeviceIdle(mDeviceIdle.isChecked())
.setRequiresCharging(mDeviceCharging.isChecked())

```

- Run your app, now with the additional constraints. Waiting until the device is idle and plugged in is a common pattern for battery intensive tasks such as downloading or uploading large files.

## 2.3 Implement the Override Deadline constraint

Up to this point, there is no way to know when the framework will execute your task. This is precisely so that it can take into account effective resource management, but it also does not guarantee that your task will run on time. For example, the news app may want to wait until wifi is available and the device is plugged in, but a careless user may forget to do so and will be disappointed when they wake up to yesterday's news. For this reason, the JobScheduler API includes the ability to set a hard deadline that will override the previous constraints.

In this step you will implement a Seekbar that will allow the user to set a deadline between 0 and 100 seconds to execute your task. Do the following:

1. Copy the horizontal LinearLayout of switches and paste it directly below the first one.
2. The SeekBar will have two labels: a static one just like label for RadioGroup of buttons, and dynamic one that will be updated with the value from the SeekBar. Change the Switch views to TextViews and modify the attributes it the following way:

| TextView 1             |   |
|------------------------|---|
| android:text           | "Override Deadline: "                     |
| android:id             | "@+id/overrideLabel"                      |
| android:textAppearance | "@style/TextAppearance.AppCompat.Subhead" |
| TextView 2             |   |
| android:text           | "Not Set"                                 |
| android:id             | "@+id/seekBarProgress"                    |
| android:textAppearance | "@style/TextAppearance.AppCompat.Subhead" |

3. Add a SeekBar view below the LinearLayout with the following attributes:

| Attribute             | Value                  |
|-----------------------|------------------------|
| android:layout_height | "wrap_content"         |
| android:layout_width  | "match_parent"         |
| android:id            | "@+id/overrideSeekBar" |
| android:layout_margin | "4dp"                  |

4. Create member variables for both the SeekBar and the TextViews, and initialize them in `onCreate()`.
5. Call `setOnSeekBarChangeListener()` on the SeekBar, passing in a new `OnSeekBarChangeListener` (Android Studio should generate the required methods):

```
mOverrideSeekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener()
{
    @Override
    public void onProgressChanged(SeekBar seekBar, int i, boolean b) {}

    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {}

    @Override
    public void onStopTrackingTouch(SeekBar seekBar) {}
});
```

6. In the onProgressChanged() callback, check if the integer value is greater than 0, and if it is, set the SeekBar progress label to the integer value and the seconds unit:

```
if (i > 0){
    mSeekBarProgress.setText(String.valueOf(i) + " s");
}
```

7. Otherwise, set the TextView to read "Not Set":

```
else {
    mSeekBarProgress.setText("Not Set");
}
```

8. The override deadline should only be set if the integer value of the SeekBar is greater than 0. In the scheduleJob() method, create a boolean variable that is true if the SeekBar has an integer value greater than 0:

```
boolean overrideSet = mOverrideSeekBar.getProgress() > 0;
```

9. If this boolean is true, call setOverrideDeadline() on the JobInfo.Builder, passing in the integer value from the SeekBar multiplied by 1000 (the parameter is in milliseconds, you want the user to set the deadline in seconds):

```
if(overrideSet){
    builder.setOverrideDeadline(mOverrideSeekBar.getProgress() * 1000);
}
```

10. Run the app. The user can now set a hard deadline by which time the JobService must be run!

## 2.4 Implement the Periodic constraint

JobScheduler also includes the possibility of scheduling a repeated task, much like AlarmManager. This option has a few caveats:

- The task is not guaranteed to run in the given period (the other conditions may not be met, or there might not be enough system resources).
- Using this constraints prevents you from also setting an override deadline or a minimum latency (see [JobInfo.Builder](#)) documentation for information on these methods), since these options do not make sense for repetitive tasks.

Do the following:

1. In activity\_main.xml, add a Switch view between the two horizontal LinearLayouts. Use the following attributes:

| Attribute             | Value                 |
|-----------------------|-----------------------|
| android:layout_height | "wrap_content"        |
| android:layout_width  | "wrap_content"        |
| android:text          | "Periodic"            |
| android:id            | "@+id/periodicSwitch" |
| android:layout_margin | "4dp"                 |

Because the override deadline and periodic constraints are mutually exclusive, you will use the switch to toggle the functionality and label of the SeekBar to represent either the override deadline, or the periodic interval, respectively.

2. Refactor mOverrideSeekBar and mOverrideLabel to the more generic mSeekBar and mLabel, and the boolean in scheduleJob() to seekBarSet (since they will now be used for multiple constraints).
3. Call `setOnCheckedChangeListener()` on the periodic switch, passing in a new `OnCheckedChangeListener`.
4. If checked, set the label to "Periodic Interval: ", otherwise to "Override Deadline":

```

@Override
public void onCheckedChanged(CompoundButton compoundButton, boolean b) {
    if (b){
        mLabel.setText("Periodic Interval: ");
    } else {
        mLabel.setText("Override Deadline: ");
    }
}

```

All that remains now is to implement the logic in the `scheduleJob()` method to properly set the constraints on the `JobInfo` object. This is how the application stands:

5. If the periodic option is turned **on** and the SeekBar has a **nonzero value**, you can set

- the constraint by calling `setPeriodic()` on the `JobInfo.Builder` object,
- 6. If the periodic option is turned **on** but the SeekBar has a **value of 0**, you should show a `Toast` message asking the user to set a periodic interval with the SeekBar.
  - 7. If the periodic option is turned **off** and the SeekBar has a **nonzero value**, the user has set an override deadline which must be applied using the `setOverrideDeadline()` option.
  - 8. If the periodic option is turned **off** and the SeekBar has a **value of 0**, the user has simply not specified an override deadline or a periodic task, so nothing should be added to the `JobInfo.Builder` object.

In code, this logic looks like this:

```
if (mPeriodic.isChecked()) {
    if (seekBarSet) {
        builder.setPeriodic(mSeekBar.getProgress()*1000);
    } else {
        Toast.makeText(MainActivity.this, "Please set a periodic interval", Toast.LENGTH_SHORT).show();
    }
} else {
    if (seekBarSet) {
        builder.setOverrideDeadline(mSeekBar.getProgress()*1000);
    }
}
```

## Coding challenge

Note: All coding challenges are optional.

In this example, the `JobService` that was scheduled based on the constraints was simple: it delivered a notification. Most of the time, however, `JobScheduler` is used for more robust background tasks such as updating the weather or syncing with a database. These kinds of tasks require a more thought since the burden falls on the developer to off - load the task as well as notify the framework when it is complete by calling `jobFinished()`.

Implement a `JobService` that starts an `AsyncTask` when the given constraints are met. The `AsyncTask` should sleep the thread for 5 seconds. This will require you to call `jobFinished()` once the task is complete. If the constraints are no longer met while the thread is sleeping, show a `Toast` message saying that the job failed and reschedule the job.

## Conclusion

JobScheduler provides a flexible framework to accomplish intelligent background services. It has three major components:

- JobScheduler, the manager class responsible for scheduling the task.
- JobInfo, created using the JobInfo.Builder class, which contains the constraints.
- JobService, the task that needs to be run, processed on the main thread.

JobScheduler batches tasks together to maximize the efficiency of system resources, so you do not have exact control of when it will be executed.

## Resources

### Android Developer Documentation

#### Reference

- [JobScheduler](#)
- [JobInfo](#)
- [JobInfo.Builder](#)
- [JobService](#)
- [JobParameters](#)

# 9.1 P: Connect to the Internet using an AsyncTask

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [Task 1. Explore the Books API](#)
- [Task 2. Create the "Who Wrote It?" App](#)
- [Task 3. Final Touches](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

In this practical you will use an AsyncTask to start a background task to get data from the Internet using a simple REST API. You will use the [Google API Explorer](#) to learn how to query the Book Search API, implement this query in a worker thread using AsyncTask, and display the result in your UI.

## What you should already KNOW

- Creating an activity.
- Adding a TextView to the layout for the activity.
- Implementing onClick functionality to a button in your layout.
- Implementing an AsyncTask and displaying the result in your UI.

## What you will LEARN

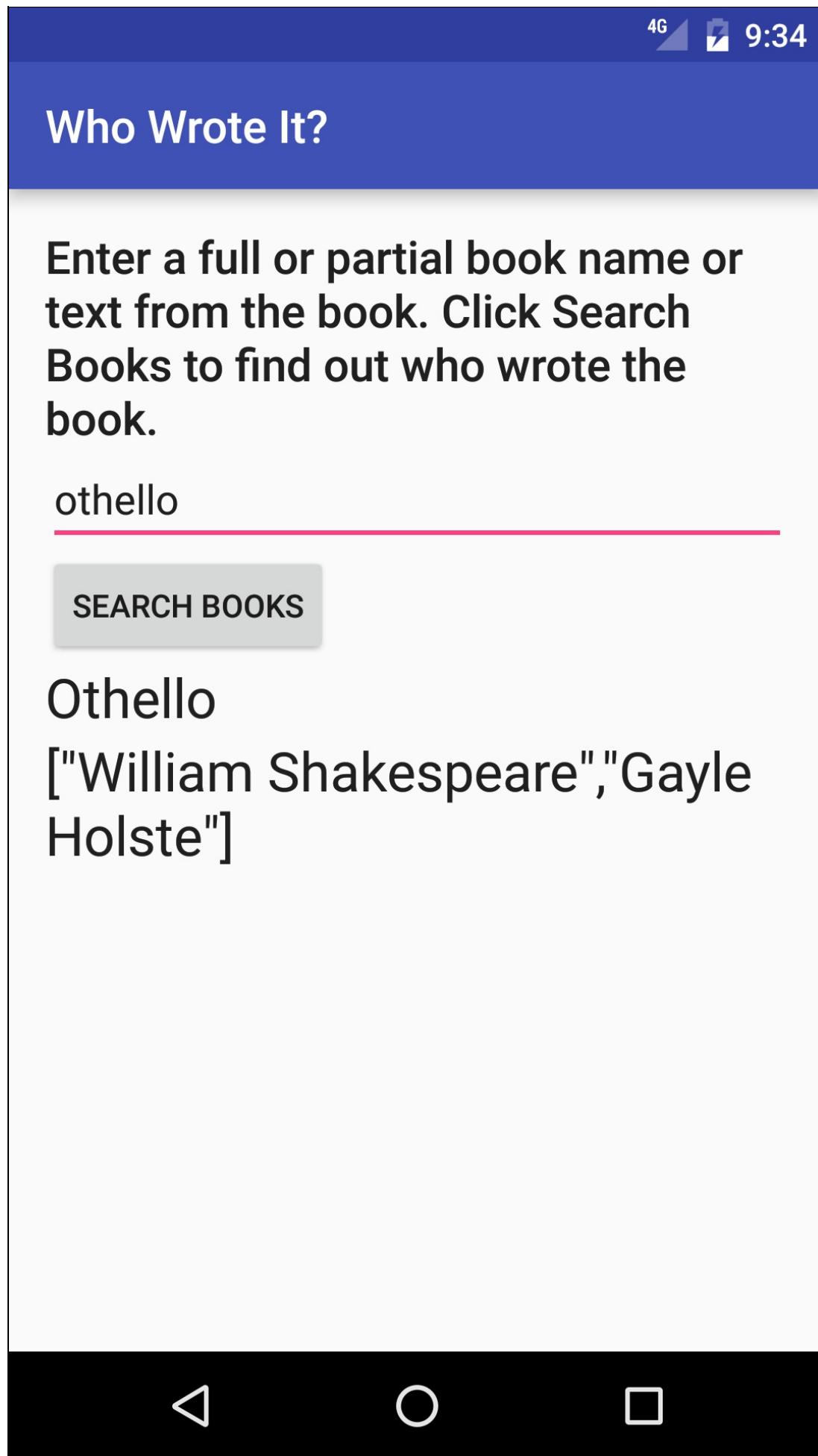
- How to use the Google APIs Explorer to investigate Google APIs and to view JSON responses to http requests.
- How to use the Books API as an example of how to retrieve data over the Internet and keep the UI fast and responsive. You won't learn the Books API in detail in this practical. Your app will only use the simple book search function. To learn more about the Books API see the [Books API reference documentation](#).
- How to parse the JSON results from your API query.

## What you will DO

- Use the Google API Explorer to learn about the Books API.
- Create a simple application that queries the Books API query using a worker thread and displays the result in the UI.

## App Overview

You will build an app that contains an EditText and a Button. The user enters the name of the book in the EditText and clicks a button. This button launches a worker thread using the AsyncTask helper class that queries the Google Book Search API to find a title and author of a book search and displays the result in a TextView below the button.



# Task 1. Explore the Books API

In this practical you will use the Google Books API to search for information about a book, such as the author(s), and the titles. The Google Books API provides programmatic access to the Google Book Search service using REST APIs. This is the same service used behind the scenes when you manually execute a search on [Google Books](#). You can use the Google API Explorer and Google Book Search in your browser to verify that your Android app is getting the expected results.

## 1.1 Send a Books API Request

1. Go to the Google APIs Explorer at <https://developers.google.com/apis-explorer/>
2. Click **Books API**.
3. Find (Ctrl-F / Cmd-F) **books.volumes.list** and click that function name. You should see a webpage that illustrates the various parameters of the Books API function that performs book searches.
4. In the **q** field enter a book name, or partial book name. The **q** parameter is the only required field. You will also use the **maxResults** and **printType** fields, to limit the results to the top 10 matching books that were printed. The **maxResults** field takes an integer value that limits the amount of results per query. The **printType** field takes one of three string arguments: "all", which does not limit the results by type at all, "books", returning only books in print, and "magazines" which returns only magazines.
5. Make sure that the "Authorize requests using OAuth 2.0" switch at the top of the form is turned off. Click "**Execute without OAuth**" at the bottom of the form.
6. Scroll down to see the Request and Response.

The Request field is an example of a Uniform Resource Identifier (URI). A URI is a string that names or locates a particular resource. URLs are a certain type of URI for identifying and locating a web resource by its primary access mechanism. For the Books API, the request is a URL that contains your search as a parameter (following the **q** parameter). Notice the API key field after the query field. Usually, to access a public API, you must obtain an API key and include it in your Request for security reasons. However, this specific API does not require a key, so you can omit that portion of the request URL in your app.

## 1.2 Analyze the Books API Response

Scroll further down to the Response for the text you entered in the **q** field. The response format is called [JSON](#), which is a common format for an API query responses. In the API Explorer web page, the JSON code is nicely formatted so that it is easily readable by

humans. In your application it will appear as a single string, and you will need to parse that string to extract the information you need.

1. Find the value for the "title" key. Notice that this result has a single key and value.
2. Find the value for the "authors" key. Notice that this one can contain an array of values.
3. In this practical, you will only return the title and authors of the first item with both of these defined in the response.

## Task 2. Create the "Who Wrote It?" App

Now that you are familiar with the Books API method that you will be using, it is time to set up the layout of your application.

### 2.1 Create the project and user interface

1. Create an app project called "Who Wrote It?" with one Activity, using the "Empty Activity" Template.
2. Add the following UI elements in the XML file, using a vertical LinearLayout as your RootView, i.e. the view that contains all of your other views inside a layout XML file.  
Make sure the LinearLayout uses `android:orientation="vertical"` :

| View     | Attributes  | Values   |
|----------|---|--|
| TextView | android:layout_width<br>android:layout_height<br><br>android:id<br><br>android:text<br><br>android:textAppearance | wrap_content<br>wrap_content<br><br>@+id/instructions<br><br>@string/instructions<br><br>@style/TextAppearance.AppCompat.Title |
| EditText | android:layout_width<br>android:layout_height<br><br>android:id<br><br>android:inputType<br><br>android:hint      | match_parent<br>wrap_content<br><br>@+id/bookInput<br><br>text<br><br>@string/input_hint                                       |
| Button   | android:layout_width<br>android:layout_height<br><br>android:id<br><br>android:text<br><br>android:onClick        | wrap_content<br>wrap_content<br><br>@+id/searchButton<br><br>@string/button_text<br><br>searchBooks                            |
| TextView | android:layout_width<br>android:layout_height<br><br>android:id<br><br>android:textAppearance                     | wrap_content<br>wrap_content<br><br>@+id/titleText<br><br>@style/TextAppearance.AppCompat.Headline                             |
| TextView | android:layout_width<br>android:layout_height<br><br>android:id<br><br>android:textAppearance                     | wrap_content<br>wrap_content<br><br>@+id/authorText<br><br>@style/TextAppearance.AppCompat.Headline                            |

3. In the string.xml file, add these string resources:

```
<string name="instructions">Enter a book name, or part of a book name, or just som  
e text from a book to find  
the full book title and who wrote the book!</string>  
<string name="button_text">Search Books</string>  
<string name="input_hint">Enter a Book Title</string>
```

4. Create a method called searchBooks() in MainActivity.java to handle the onClick button action. As all onClick methods, it should take a View as a parameter.

## 2.2 Set up the Main Activity

To query the Books API, you need to get the user input from the EditText.

1. In MainActivity.java, create member variables for the EditText, the author TextView and the title TextView.
2. Initialize these variables by id in onCreate().
3. In the searchBooks method, get the text from the EditText widget and convert to a String, assigning it to a string variable.

```
String queryString = mBookInput.getText().toString();
```

**Note:** mBookInput.getText() returns an “Editable” datatype which needs to be converted into a string.

## 2.3 Create an empty AsyncTask

You are now ready to connect to the Internet and the Book Search REST API. Network connectivity can be sometimes be sluggish or experience delays. This may cause your app to behave erratically or become slow so you should not make a network connection on the UI thread. If you attempt a network connection on the UI thread, the Android runtime may raise a [NetworkOnMainThreadException](#) to warn you it is a bad idea. It is possible to disable this warning, but that’s a topic for an advanced Android class. Use an AsyncTask to make network connections:

1. Create a new Java class called FetchBook in app/java and have it extend AsyncTask.

An AsyncTask requires three arguments: the input parameters, the progress indicator and the result type. The generic type parameters for the task will be `<String, Void, String>` since the AsyncTask takes a String as the first parameter (the query), Void since there is no progress update, and String since it returns a string as a result (the JSON response).

2. Implement the required method (`dolnBackground()`) by placing your cursor on the red underlined text, pressing **Alt + Enter** (**Opt + Enter** on a Mac) and selecting **Implement methods**. Choose `dolnBackground()` and press **OK**. Make sure the parameters and return types are the correct type (It takes a String array and returns a String).
3. Click the **Code** menu and choose **Override methods** (or pressing **Ctrl + O**). Select the `onPostExecute()` method. The `onPostExecute()` method should take a String as a parameter and return void.
4. To display the results in the TextViews, you must have access to those TextViews inside the AsyncTask. Create member variables in the FetchBook AsyncTask for the two TextViews that show the results, and assign them in a constructor. You will use this

- constructor in your MainActivity to pass along the TextViews to your AsyncTask.
5. Create a unique LOG\_TAG variable to be used throughout your AsyncTask for logging.

### Solution Code for FetchBook:

```
public class FetchBook extends AsyncTask<String,Void,String>{  
    private TextView mTitleText;  
    private TextView mAuthorText;  
  
    private static final String LOG_TAG = FetchBook.class.getSimpleName();  
  
  
    public FetchBook(TextView mTitleText, TextView mAuthorText) {  
        this.mTitleText = mTitleText;  
        this.mAuthorText = mAuthorText;  
    }  
  
    @Override  
    protected String doInBackground(String... params) {  
        return null;  
    }  
  
    @Override  
    protected void onPostExecute(String s) {  
        super.onPostExecute(s);  
    }  
}
```

## 2.4 Build the URI

In this step, you will open an Internet connection and query the Books API. This section has quite a lot of code so remember to visit the developer documentation for [Connecting to the Network](#) if you get stuck.

1. In the doInBackground() method of the FetchBook class, get the query string from the params variable and assign it to a String variable called queryString. The parameter of the doInBackground() method is a String array generated by the Android runtime system and created from the arguments passed into the execute() method when it is triggered. In this case, you will only pass in a single string (the query string), so it will be stored in the first slot of the params array.

```
String queryString = params[0];
```

2. Create the following two local variables in doInBackground() that will be needed later to help connect and read the incoming data.

```
HttpURLConnection urlConnection = null;  
BufferedReader reader = null;
```

3. Create another local variable in doInBackground() to contain the raw response from the query (the JSON string you examined earlier) and return it at the end of the method:

```
String bookJSONString = null;  
return bookJSONString;
```

4. Create a skeleton try/catch/finally block. This is where you will make your HTTP request. All of the following code will go in the try block. The catch block is used to handle any problems with making the HTTP request and the finally block is for closing the network connection after you've finished receiving the JSON data and returning the result.

```
try {  
    ...  
} catch (Exception ex) {  
    ...  
} finally {  
    return bookJSONString;  
}
```

5. If you recall the request URI from the Books API webpage, you will notice all of the requests begin with the same base URI. After this base URI, there are appended query parameters that specify the type of resource you are looking for. It is common practice to separate all of these query parameters into constants, and combine them using a [URI.Builder](#) so that they can be reused for different URI's. The Uri class has a convenient method, Uri.buildUpon() that returns a URI.Builder that we can use.

For this application, we limit the number and type of results in order to increase the query speed, and only look for books that are printed. In the try block, build the desired URL:

```
final String BOOK_BASE_URL = "https://www.googleapis.com/books/v1/volumes?"; // Base  
URI for the Books API  
final String QUERY_PARAM = "q"; // Parameter for the search string  
final String MAX_RESULTS = "maxResults"; // Parameter that limits search results  
final String PRINT_TYPE = "printType"; // Parameter to filter by print type
```

1. Build up your request URI:

```
//Build up your query URI, limiting results to 10 items and printed books
Uri builtURI = Uri.parse(BOOK_BASE_URL).buildUpon()
    .appendQueryParameter(QUERY_PARAM, queryString)
    .appendQueryParameter(MAX_RESULTS, "10")
    .appendQueryParameter(PRINT_TYPE, "books")
    .build();
```

## 2. Convert your URI to a URL:

```
URL requestURL = new URL(builtURI.toString());
```

## 2.5 Make the Request

Making an API request over the Internet is a common pattern. You may want to create a utility class with this functionality or develop a useful subclass for your own convenience. This pattern uses the [HttpURLConnection](#) class in combination with an [InputStream](#) and a [StringBuffer](#) to obtain the JSON response from the web. If at any point the process fails and [InputStream](#) or [StringBuffer](#) are empty, it returns null signifying that the query failed.

### 1. In the try block of the `doInBackground()` method, open the URL connection and make the request:

```
urlConnection = (HttpURLConnection) requestURL.openConnection();
urlConnection.setRequestMethod("GET");
urlConnection.connect();
```

### 2. Read the response using an [InputStream](#) and a [StringBuffer](#), then convert it to a String:

```
InputStream inputStream = urlConnection.getInputStream();
StringBuffer buffer = new StringBuffer();
if (inputStream == null) {
    // Nothing to do.
    return null;
}
reader = new BufferedReader(new InputStreamReader(inputStream));
String line;
while ((line = reader.readLine()) != null) {
    /* Since it's JSON, adding a newline isn't necessary (it won't affect
    parsing) but it does make debugging a *lot* easier if you print out the
    completed buffer for debugging. */
    buffer.append(line + "\n");
}
if (buffer.length() == 0) {
    // Stream was empty. No point in parsing.
    return null;
}
bookJSONString = buffer.toString();
```

3. Close the try block and log the exception in the catch block.

```
catch (IOException e) {  
    e.printStackTrace();  
    return null;  
}
```

4. Close both the urlConnection and the reader variables in the finally block:

```
finally {  
    if (urlConnection != null) {  
        urlConnection.disconnect();  
    }  
    if (reader != null) {  
        try {  
            reader.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

**Note:** Each time the connection fails, this code returns null. This means that onPostExecute() will have to check for a null string and let the user know that connection failed. This error handling strategy is overly simple, as the user has no idea why the connection failed. A better solution would be to handle each point of failure differently so that the user can get the appropriate feedback.

5. Log the value of the bookJSONString variable before returning it. We are now done with the doInBackground() method.

```
Log.d(LOG_TAG, bookJSONString);
```

6. Now that your AsyncTask is set up, you need to launch it from the MainActivity using the execute() method. Add the following code to your searchBooks method in MainActivity.java to launch the AsyncTask:

```
new FetchBook(mTitleText, mAuthorText).execute(mQueryString);
```

7. Run your app. Execute a search. Your app will crash. Look at your Logs, what is causing the error? You should see the following line:

```
Caused by: java.lang.SecurityException: Permission denied (missing INTERNET permission?)
```

This error indicates that you have not included the permission to access the internet in your AndroidManifest.xml file. Connecting to the internet introduces new security

concerns, which is why by default your apps will not have connectivity, and you must add it manually in the form of a permission in the Manifest.

## 2.6 Add the Internet permission

1. Open the AndroidManifest.xml file.
2. Add the following xml tag outside of the application tag:

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

This is where all the permissions that your app needs will go. Android has a specific design philosophy when it comes to permissions which you will learn about later, but for now just understand that to connect to the internet, your app requires this Internet permission declared in the manifest.

3. Build and run your app again. Running a query should now result in a JSON string being printed to the Log.

## 2.7 Parse the JSON string

Now that you have the correct response to your query, you must parse the results to extract the information you want to display in the UI. Fortunately, Java has existing classes that aid in the parsing and handling of JSON type data. This process, as well as updating the UI, will happen in the onPostExecute() method.

As previously mentioned, there is chance that the doInBackground() method does not return the expected JSON string (the try catch fails and throws an exception, the network times out or a number of other possible errors). In that case, the Java JSON methods will fail to parse the data and another exception will be thrown. This is why the parsing must also be done in a try block, and the case where incorrect or incomplete data is return must be handled in the catch block.

To parse the JSON data and handle possible exceptions, do the following:

1. In onPostExecute(), add a try/catch block below the call to super.
2. Use the built in Java JSON classes (JSONObject and JSONArray) to obtain the JSON array of results items in the try block.

```
JSONObject jsonObject = new JSONObject(s); //Convert the response into a JSON object  
JSONArray itemsArray = jsonObject.getJSONArray("items"); // Get the JSON Array of book items
```

3. Iterate through the itemsArray, checking each book for title and author information. If both are not null, exit the loop and update the UI; otherwise, continue looking through the list. This way, only entries with both a title and authors will result in a result being displayed.

```
//Iterate through the results
for(int i = 0; i<itemsArray.length(); i++){
    JSONObject book = itemsArray.getJSONObject(i); //Get the current item
    String title=null;
    String authors=null;
    JSONObject volumeInfo = book.getJSONObject("volumeInfo");

    // Try to get the author and title from the current item, catch if either field is
    // empty and move on
    try {
        title = volumeInfo.getString("title");
        authors = volumeInfo.getString("authors");
    } catch (Exception e){
        e.printStackTrace();
    }

    //If both a title an authors exist, update the textviews and return
    if (title != null && authors != null){
        mTitleText.setText(title);
        mAuthorText.setText(authors);
        return;
    }
}
```

4. If no results with valid titles and authors are found (if the loop finishes executing), set the title textView to read "No Results Found", and clear the authors textView.
5. In the catch block, print the error to the log, set the title TextView to "No Results Found", and clear the author textView.

**Solution Code:**

```
//Method for handling the results on the UI thread
@Override
protected void onPostExecute(String s) {
    super.onPostExecute(s);
    try {
        JSONObject jsonObject = new JSONObject(s); //Convert the response into a JSON object
        JSONArray itemsArray = jsonObject.getJSONArray("items"); // Get the JSON Array of book items
        //Iterate through the results
        for(int i = 0; i<itemsArray.length(); i++){
            JSONObject book = itemsArray.getJSONObject(i); //Get the current item
            String title=null;
            String authors=null;
            JSONObject volumeInfo = book.getJSONObject("volumeInfo");

            // Try to get the author and title from the current item, catch if either field is empty and move on
            try {
                title = volumeInfo.getString("title");
                authors = volumeInfo.getString("authors");
            } catch (Exception e){
                e.printStackTrace();
            }

            //If both a title an authors exist, update the textviews and return
            if (title != null && authors != null){
                mTitleText.setText(title);
                mAuthorText.setText(authors);
                return;
            }
        }

        // If none are found, update the UI to show failed results
        mTitleText.setText("No Results Found");
        mAuthorText.setText("");

    } catch (Exception e){
        // If onPostExecute does not receive a proper JSON string, update the UI to show failed results
        mTitleText.setText("No Results Found");
        mAuthorText.setText("");
        e.printStackTrace();
    }
}
```

## Task 3. Final Touches

You now have a functioning app that uses the Books API to execute a book search.

However, there are a few things that do not behave as expected:

- When the user clicks "Search Books", the keyboard does not disappear, and there is no indication to the user that the query is being executed.
- If there is no network connection, or the search field is empty, the app still tries to query the API and fails without properly updating the UI.
- If you rotate the screen during a query, the AsyncTask becomes disconnected from the Activity, and it is not able to update the UI with the results.

You will fix these issues in the following section.

### 3.1 Hide the Keyboard and Update the TextView

The user experience of searching is not intuitive when the button is pushed: the keyboard remains visible and there is no way to know that the query is in progress. One solution is to programmatically hide the keyboard and update one of the result textviews to read "Loading..." while the query is being performed. Do the following:

1. Add the following code to the searchBooks() method to hide the keyboard when the button is pushed:

```
InputMethodManager inputManager = (InputMethodManager) getSystemService(Context.INPUT_METHOD_SERVICE);
inputManager.hideSoftInputFromWindowgetCurrentFocus().getWindowToken(), InputMethodManager.HIDE_NOT_ALWAYS);
```

2. Add a line of code below the call to execute the FetchBook task that changes the title textview to read "Loading..." and clears the author textview.
3. Extract your String resources.

### 3.2 Network state management and the empty search field case

Whenever your application uses the network, prepare for the possibility that a network connection is unavailable. Before attempting to connect to the network in your AsyncTask, your app should check the state of network connection.

1. Add the following permission to your Android Manifest to enable access to the network connection state:

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

2. Modify your searchBooks() method to check both the connection and if there is anything

- in the search field before executing the FetchBook task.
3. Update the UI in the case that there is no connection or no search query, to prompt the user to fix the error.

### Solution Code:

```
public void searchBooks(View view) {  
    //Gets the search string from the input field  
    String queryString = mBookInput.getText().toString();  
  
    //Hides the keyboard when the button is pushed  
    InputMethodManager inputManager = (InputMethodManager)  
        getSystemService(Context.INPUT_METHOD_SERVICE);  
    inputManager.hideSoftInputFromWindowgetCurrentFocus().getWindowToken(),  
        InputMethodManager.HIDE_NOT_ALWAYS);  
  
    //Checks the status of the network connection  
    ConnectivityManager connMgr = (ConnectivityManager)  
        getSystemService(Context.CONNECTIVITY_SERVICE);  
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();  
  
    //If the network is active and the search field is not empty, starts a FetchBook As  
    yncTask  
    if (networkInfo != null && networkInfo.isConnected() && queryString.length()!=0) {  
        new FetchBook(mTitleText, mAuthorText).execute(queryString);  
        mAuthorText.setText("");  
        mTitleText.setText(R.string.loading);  
    }  
    // Otherwise updates the TextView to tell the user there is no connection or no sea  
    rch term  
    else {  
        if (queryString.length() == 0) {  
            mAuthorText.setText("");  
            mTitleText.setText("Please enter a search term");  
        } else {  
            mAuthorText.setText("");  
            mTitleText.setText("Please check your network connection and try again.");  
        }  
    }  
}
```

## 3.3 Configuration changes and more

As in the previous AsyncTask example, rotating the screen (or any other configuration changes that cause the activity to be recreated) disassociates the UI from the task. When this happens the the results never get displayed. To work around this problem, Android

provides a set of classes called [Loaders](#), which work with the Activity lifecycle to preserve and manage your data. In the next chapter you will migrate your FetchBook task to use a more useful class (AsyncTaskLoader) instead of an AsyncTask.

For now, go through your code and [extract your resources](#) to finalize this version of Who Wrote It!

## Coding challenge

Explore the the specific API you are using in greater detail and find a search parameter that restricts the results to books that are downloadable in the epub format. Add this parameter to your request and view the results.

Note: All coding challenges are optional.

## Summary

- Network connectivity should not be executed on the UI thread
- The Android runtime usually defaults to StrictMode which will raise an Exception if you attempt network connectivity or file access on the UI thread.
- The Google API Explorer is a tool that helps you explore various Google APIs interactively
- The Books Search API is a set of RESTful APIs to access Google Books programmatically
- An API request to Google Books is in the form of a URL
- The response to that API request returns in the form of a JSON string
- Text from an EditText view is retrieved via `getText()`. It can be converted into a simple String by using `toString()`.
- The Uri class has a convenient method, `Uri.buildUpon()` that returns a `URI.Builder` that can be used to construct a URI string
- An AsyncTask is a high-level abstraction (a helper class) for programmers so they don't have to implement difficult Thread state logic.
- An AsyncTask can be started via `execute()`, i.e., a 0-length varargs
- You must configure network permission in the Android manifest file to connect to the Internet: `<uses-permission android:name="android.permission.INTERNET" />`
- Use the built in Java JSON classes (`JSONObject` and `JSONArray`) to create and parse JSON strings

## Resources

# Android Developer Documentation

## Guides

- [Connecting to the Network](#)
- [Managing Network State](#)

## Reference

- [AsyncTask](#)

## 9.2 P: Using an AsyncTaskLoader

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [Task 1. AsyncTask vs AsyncTaskLoader](#)
- [Coding challenge](#)
- [Summary](#)
- [Resources](#)

In the previous lesson, you learned how to use an `AsyncTask` to perform work that needs to happen outside the main UI thread. However, as you've seen, `AsyncTask` has a significant drawback: whenever the device's configuration changes (e.g., rotating the device from portrait to landscape), `AsyncTask` recreates the objects in the Activity. Since the newly created UI objects in your Activity are different objects from when the `AsyncTask` was created, the worker thread in your `AsyncTask` can not update the UI.

You should use an `AsyncTaskLoader`, instead of an `AsyncTask`, to run a task asynchronously in the background when loading data. This `AsyncTaskLoader` will take care of associating the task to the appropriate activity as necessary.

### What you should already KNOW

- Displaying data in a `RecyclerView`.
- How to pass information between Activities as Extras.

### What you will LEARN

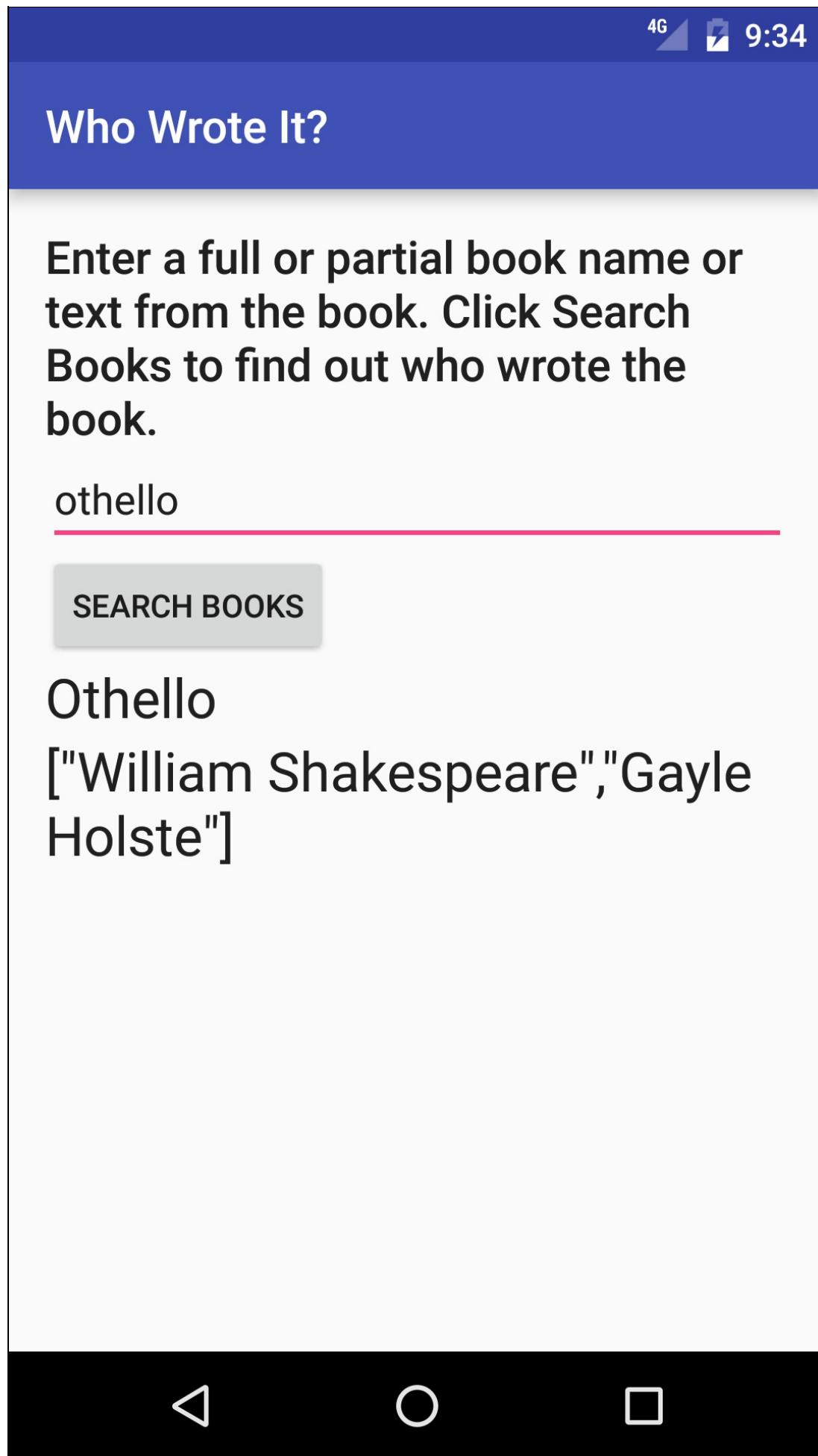
- How to implement an `AsyncTaskLoader` that preserves data on configuration changes.
- How to update your UI using the Loader Callbacks.

### What you will DO

- Modify the "Who Wrote it?" app to use an `AsyncTaskLoader` instead of an `AsyncTask`.

## App Overview

The improved "Who Wrote It?" app can now function properly through orientation changes.



# Task 1. AsyncTask vs AsyncTaskLoader

When using an `AsyncTask`, the UI could not be updated if a configuration change happened while the background task was running. The Android SDK provides a set of classes called loaders meant specifically for loading data asynchronously. If you use a loader, you don't have to worry about the loader losing the ability to update the UI in the activity that initially called it. The Android system does the work for you of reconnecting the loader to the appropriate Activity if the device changes state. This means if you rotate the device while the task is still running, the results will still be displayed correctly in the Activity.

In this practical you will be using a specific loader called an `AsyncTaskLoader`. An `AsyncTaskLoader` is a subclass of `Loader` that uses an `AsyncTask` to make it loading data in the background much easier.

When you use an `AsyncTask`, you implement the `onPostExecute()` method ***in the AsyncTask*** to display the results on the screen. When you use an `AsyncTaskLoader`, you define callback methods ***in the Activity*** to display the results.

Loaders provide a lot of additional functionality beyond just running tasks and reconnecting to the Activity. For example, you can attach a loader to a data source and have it automatically update the UI elements when the underlying data changes. Loaders can also be programmed to resume loading if interrupted.

So when should you use an `AsyncTask` if an `AsyncTaskLoader` is so much more robust? The answer is that it depends on the context. If the background task is likely to finish before any configuration changes occur, and it is not crucial that it updates the UI, an `AsyncTask` may be sufficient. The Loader framework can be inflexible, and actually uses an `AsyncTask` behind the scenes to work its magic. A good rule of thumb is to check whether a screen rotation can happen in the time that your task is running, and if so, use a loader instead of an `AsyncTask`.

In this chapter, you will learn how to use a `AsyncTaskLoader` instead of an `AsyncTask` to run your Books API query. You will learn more about the other uses of loaders in a later lesson.

Implementing a Loader requires the following components:

1. An extension of a loader class (in this case, `AsyncTaskLoader`).
2. An Activity that implements the `LoaderManager.LoaderCallbacks` class.
3. An instance of the `LoaderManager`.

## 1.1 Create an `AsyncTaskLoader`

Begin by [copying the WhoWroteIt project](#), in order to preserve the result of the previous practical. Call the copied project WhoWroteItLoader. Then, do the following:

1. Create a new class in your java directory called BookLoader.
2. Have your BookLoader class extend AsyncTaskLoader with parameterized type . **Make sure you import the loader from the support library.**
3. Implement the required method (loadInBackground()). Notice the similarity between this method and the initial doInBackground() method from AsyncTask.
4. In Android Studio, it is likely the class declaration will still be underlined in red. With your text cursor on the class declaration line, press **Alt + Enter (Option + Enter on a Mac)** and choose **Create constructor matching super**. This will create a constructor with the Context as a parameter.
5. Press **Ctrl + O** to open the Override methods menu, and select **onStartLoading**. This method is called by the system when you start your loader.

The loader will not actually start loading the data until you call the `forceLoad()` method.

6. Inside the onStartLoading method stub, call `forceLoad()` to start the loadInBackground() method once the Loader is created.
7. Create a member variable `mQueryString` that will hold the query String called, and modify the constructor to take a String as an argument and assign it to the `mQueryString` variable.
8. Copy all of the code from the doInBackground() method in the FetchBook class to the loadInBackground() method in your BookLoader class. Delete the code that get the query String from the params variable:

```
String queryString = params[0];
```

9. Change the reference to the `queryString` variable inside the URI builder to `mQueryString`.
10. Create the `LOG_TAG` string for your log statements.

```
//Connects to the network and makes the Books API request on a Background thread
@Override
public String loadInBackground() {
    //Sets up variables to be closed
    HttpURLConnection urlConnection = null;
    BufferedReader reader = null;
    String bookJSONString = null;

    //Attempts to query the Books API
    try {
        final String BOOK_BASE_URL = "https://www.googleapis.com/books/v1/volumes?"; /
        // Base URI for the Books API
        final String QUERY_PARAM = "q"; // Parameter for the search string
```

```
final String MAX_RESULTS = "maxResults"; // Parameter that limits search results

final String PRINT_TYPE = "printType"; // Parameter to filter by print type

//Build up your query URI, limiting results to 5 items and printed books
Uri builtURI = Uri.parse(BOOK_BASE_URL).buildUpon()
    .appendQueryParameter(QUERY_PARAM, mQueryString)
    .appendQueryParameter(MAX_RESULTS, "10")
    .appendQueryParameter(PRINT_TYPE, "books")
    .build();

URL requestURL = new URL(builtURI.toString());

//Opens the connection
urlConnection = (HttpURLConnection) requestURL.openConnection();
urlConnection.setRequestMethod("GET");
urlConnection.connect();

//Gets the InputStream and reads the response string into a StringBuffer
InputStream inputStream = urlConnection.getInputStream();
StringBuffer buffer = new StringBuffer();
if (inputStream == null) {
    // Nothing to do.
    return null;
}
reader = new BufferedReader(new InputStreamReader(inputStream));

String line;
while ((line = reader.readLine()) != null) {
    /* Since it's JSON, adding a newline isn't necessary (it won't affect parsing
     * but it does make debugging a *lot* easier if you print out the completed buffer
     * for debugging. */
    buffer.append(line + "\n");
}

if (buffer.length() == 0) {
    // Stream was empty. No point in parsing.
    // return null;
    return null;
}
bookJSONString = buffer.toString();

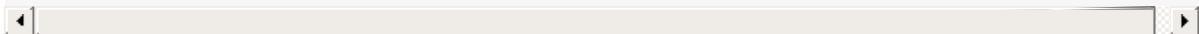
// Catches errors
} catch (IOException e) {
    e.printStackTrace();
    return null;
}

//Closes the connections
} finally {
    if (urlConnection != null) {
        urlConnection.disconnect();
    }
}
```

```

        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    //Returns the raw response
    return bookJSONString;
}

```



## 1.2 Modify your MainActivity

You must now implement the [Loader Callbacks](#) in your MainActivity to handle the results of your loadInBackground() AsyncTaskLoader method.

1. Add the LoaderManager.LoaderCallbacks implementation to your Main Activity class declaration, parameterized with the String type:

```
public class MainActivity extends AppCompatActivity implements LoaderManager.LoaderCallbacks<String>{
```

2. Implement all of the required methods. Place your text cursor on the class signature line and enter **Alt + Enter (Option + Enter on a Mac)**. Make sure all the methods are selected. Also check your imports to ensure you are importing the LoaderManager class from the support library package, to match the loader that you also imported from the support library.

**Note:** If the imports for Loader and LoaderManager in MainActivity do not match the import for the AsyncTaskLoader for the BookLoader class, you will have some type errors in the callbacks. Make sure all of these import from the support library.

3. Loaders use [Bundle](#) class to pass information from the calling activity to the LoaderCallbacks. You can add primitive data to a bundle with the appropriate putType() method.

To start a loader, you have two options:

4. initLoader(): this method creates a new loader if one does not exist already, passing in the arguments Bundle. If a loader exists, the calling Activity is re-associated with it without updating the Bundle.
5. restartLoader(): this method is the same as initLoader() except that if it finds an existing loader, it destroys and recreates it with the new Bundle.

Instead of executing the FetchBook AsyncTask in response to the button press, replace the call to execute the FetchBook task with a call to `restartLoader()` (you want to start up your loader with a new Bundle each time the button is pressed), passing in the query string you got from the EditText in the Bundle:

```
Bundle queryBundle = new Bundle();
queryBundle.putString("queryString", queryString);
getSupportLoaderManager().restartLoader(0, queryBundle, this);
```

The `restartLoader` method takes three arguments:

6. A loader id (useful if you implement more than one Loader in your activity)
7. An arguments Bundle (this is where any data needed by the loader goes)
8. The instance of LoaderCallbacks you implemented in your activity. If you want the loader to deliver the results to the MainActivity, specify `this` as the third argument.

Examine the Override methods from the LoaderCallbacks class. These are where you instantiate your Loader (`onCreateLoader`), update your UI with the results (`onLoadFinished`), and clean up any remaining resources (`onLoaderReset`). You will only be using the first two methods, since your current data model is a simple string that does not need extra care when the loader is reset.

9. In `onCreateLoader`, return an instance of the BookLoader class, passing in the `queryString` obtained from the passed in args Bundle:

```
return new BookLoader(this, args.getString("queryString"));
```

10. Update `onLoadFinished()` to process your result: the raw JSON String response from the BooksAPI.
  - i. Copy the code from `onPostExecute()` in your FetchBook class to `onLoadFinished()` in your MainActivity, excluding the call to `super.onPostExecute()`.
  - ii. Replace the argument to the `JSONObject` constructor with the passed in data String.
  - iii. Create a `LOG_TAG` for the MainActivity.
11. Run your app.
12. You should have the same functionality as before, but now in a Loader! One thing still does not work. When the phone is rotated, the View data is lost. That is because when the Activity is created (or recreated), the Activity does not know there is a loader running. An `initLoader()` method is needed in `onCreate()` of MainActivity to reconnect to the loader. Add the following code in `onCreate` to reconnect to the Loader if it already exists:

```
if(getSupportLoaderManager().getLoader(0)!=null){  
    getSupportLoaderManager().initLoader(0,null,this);  
}
```

Note: This pattern may appear to be a little counter intuitive. You are only initializing the loader if it already exists, and not the other way around. You only want to reattach the loader to the activity if a query has already been executed. In the initial state of the app, no data is loaded so there is none to preserve.

13. Run your app again and rotate the device. The LoaderManager now holds on to your data across device configurations!
14. Remove the FetchBook class as it is no longer used.

## Coding challenge

The response from the Books API contains as many results as you set with the maxResults parameter, but in this implementation you are only returning the first valid Book result. Modify your app so that the Data is displayed in a RecyclerView that has maxResults amount of entries.

Note: All coding challenges are optional.

## Summary

- A Loader allows asynchronous loading of data in an Activity
- A Loader can be used to re-establish communication to the UI when an Activity is prematurely terminated (e.g., by device rotation).
- An AsyncTaskLoader is a Loader that uses an AsyncTask helper class to make it easier handling Threads
- Loaders are managed by a LoaderManager
- A LoaderManager managers one or more Loader instances
- An AsyncTask may experience issues if the Activity it is controlling prematurely terminates (e.g., device rotation).
- An AsyncTaskLoader can be used to re-associate a worker thread with a newly created Activity (ie, after a device rotation when an Activity might be re-created).
- The LoaderManager allows you to associate a newly created Activity with a Loader using `getSupportLoaderManager().initLoader()`.

## Resources

# Android Developer Documentation

## Guides

- [Loaders](#)
- [Managing Network State](#)

## Reference

- [AsyncTaskLoader](#)

# 10.0 P: Shared Preferences

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Explore HelloSharedPrefs](#)
- [Task 2. Save and restore data to shared preferences](#)
- [Task 3. Add a Reset button](#)
- [Coding challenge](#)
- [Conclusion](#)
- [Resources](#)

Shared preferences allow you to read and write small amounts of primitive data (as key/value pairs) to a file on the device storage. The `SharedPreferences` class provides APIs for getting a handle to a preference file and for reading, writing, and managing this data. The shared preferences file itself is managed by the framework, and accessible to (shared with) all the components of your app. That data is not, however, shared with or accessible to any other apps.

Shared preferences are different from the activity instance state you learned about earlier. Instance state only preserves state data across activity instances in the same user session. Shared preferences persist across user sessions, even if your app is killed and restarted.

Android also includes a set of Preference APIs, which are different from shared preferences. The Preference APIs can be used to build user interface for a settings page, although they do use shared preferences for their underlying implementation. See [Settings](#) for more information on settings and the Preference APIs.

Use shared preferences only when you have a small amount of simple key/value pairs you want to save. To manage larger amounts of persistent app data use the other methods such as SQL databases, which you will learn about in a later chapter.

## What you should already KNOW

From the previous practicals you should be familiar with:

- Creating, building, and running apps in Android Studio.

- Designing layouts with buttons and text views.
- Using styles and themes.
- Saving and restoring activity instance state.

## What you will LEARN

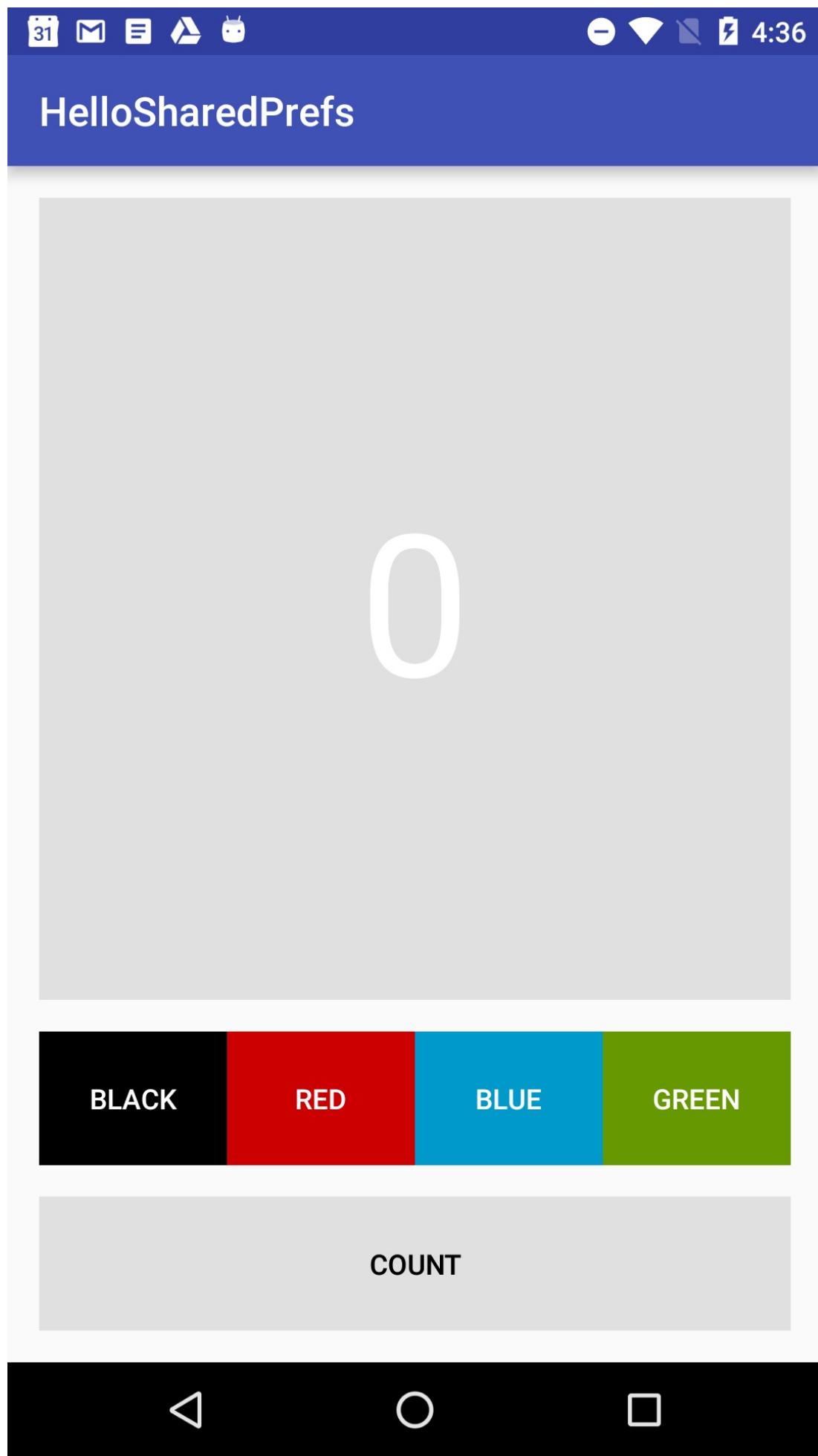
- What shared preferences are.
- How to create a shared preferences file for your app.
- How to save data to shared preferences, and read those preferences back again.
- How to clear the data in the shared preferences.

## What you will DO

- You will add shared preferences to a starter app.
- You will add a "Reset" button to the app that clears both the app state and the preferences for the app.

## App Overview

The HelloSharedPrefs app is an adaptation of the HelloToast app you created in Lesson 1. It includes a more flexible layout and additional buttons for changing the background color. The app also uses themes and styles and saves its instance state on rotation.



You'll start with the starter app in this practical and add shared preferences to the main activity code. You'll also add a reset button that sets both the count and the background color to the default, and clears the preferences file.

## Task 1. Explore HelloSharedPrefs

The complete starter app project for HelloSharedPrefs is available at [HelloSharedPrefs-start.zip](#). In this task you will load the project into Android Studio and explore some of the app's key features.

### 1.1 Open and Run the HelloSharedPrefs Project

1. Download the HelloSharedPrefs app and unzip the file.
2. Open the app in Android Studio.
3. Build and run the project in Android Studio. Try these things:
  - Click the Count button to increment the number in the main text view.
  - Click any of the color buttons to change the background color of the main text view.
  - Rotate the device and note that both background color and count are preserved.
4. Force-quit the app using one of these methods:
  - In Android Studio, select Run > Stop 'app' or click the Stop Icon  in the toolbar.
  - On the device, click the Recents button (the square button in the lower right corner). Swipe the card for the HelloSharedPrefs app to quit, or click the X in the right corner. If you quit the app in this manner, wait a few seconds before starting it again so the system can clean up.
5. Re-run the app.

The app restarts with the default appearance -- the count is 0, and the background color is grey.

### 1.2 Explore the Activity code

1. Open MainActivity (`java/com.example.android.simplecalc/MainActivity`).
2. Examine the code and note these things:
  - The count (`mCount`) is defined by an integer. The `countUp()` click handler method increments this value and updates the main textview.
  - The color (`mCurrentColor`) is also an integer that is initially defined as grey in the `colors.xml` resource file as `default_background`.
  - The `changeBackground()` click handler method gets the background color of the button that was clicked and then sets the background color of the main textview.
  - Both the count and color integers are saved to the instance state bundle in

onSaveInstanceState(), and restored in onCreate().

## Task 2. Save and restore data to shared preferences

In this task you'll save the state of the app to a shared preferences file, and read that data back in when the app is restarted. Most of this code is boilerplate you can add to any app.

### 2.1 Initialize the preferences

1. In MainActivity, import the SharedPreferences class:

```
import android.content.SharedPreferences;
```

2. Add member variables to the MainActivity class to hold the name of the shared preferences file, and a reference to a SharedPreferences object.

```
private SharedPreferences mPreferences;  
private String sharedPrefFile = "com.example.android.hellosharedprefs";
```

You can name your shared preferences file anything you want to, but conventionally it has the same name as the package name of your app.

3. In the onCreate() method, initialize the shared preferences. Make sure you insert this code before the `if` statement.:

```
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
```

The `getSharedPreferences()` method opens the file at the given file name (`sharedPrefFile`) with the mode `MODE_PRIVATE`.

**Note:** Older versions of Android had other modes that allowed you to create a world-readable or world-writable shared preferences file. These modes were deprecated in API 17, and are now **strongly discouraged** for security reasons. If you need to share data with other apps, use a service or a content provider.

#### Solution Code (Main Activity - partial)

```

public class MainActivity extends AppCompatActivity {
    private int mCount = 0;
    private TextView mShowCount;
    private int mCurrentColor;

    private SharedPreferences mPreferences;
    private String sharedPrefFile = "com.example.android.hellosharedprefs";

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

        mShowCount = (TextView) findViewById(R.id.textview);
        mCurrentColor = ContextCompat.getColor(this, R.color.default_background);
        mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);

        // ...
    }
}

```

## 2.2 Save preferences in onPause()

Saving preferences is a lot like saving the instance state -- both operations set aside the data you're interested in as a key/value pair. For shared preferences, however, save that data in the onPause() lifecycle callback, and you need a shared editor editor object ([SharedPreferences.Editor](#)) to write to the shared preferences object.

1. Click the last line of the MainActivity class, just before the closing bracket.
2. Select Code > Generate, then select Override Methods.
3. Type "onPause", select the method signature for the onPause() method, and click OK.

A skeleton onPause() method is added to the insertion point.

4. Get an editor for the SharedPreferences object:

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
```

A shared preferences editor is required to write to the shared preferences object. Add this line to onPause() after the call to super.onPause().

5. Use the putInt() method to put both the mCount and mCurrentColor integers into the shared preferences with the appropriate keys:

```
preferencesEditor.putInt("count", mCount);
preferencesEditor.putInt("color", mCurrentColor);
```

6. Call apply() to save the preferences:

```
preferencesEditor.apply();
```

The apply() method saves the preferences asynchronously, off of the UI thread. The shared preferences editor also has a commit() method to synchronously save the preferences. The commit() method is discouraged as it can block other operations.

### Solution Code (Main Activity - onPause() method)

```
@Override
protected void onPause(){
    super.onPause();

    SharedPreferences.Editor preferencesEditor = mPreferences.edit();
    preferencesEditor.putInt("count", mCount);
    preferencesEditor.putInt("color", mCurrentColor);
    preferencesEditor.apply();
}
```

## 2.3 Restore preferences in onCreate()

As with the instance state, your app reads any saved shared preferences in the onCreate() method. For our particular app we only want to read the preferences if there is no savedInstanceState. You can use the savedInstanceState argument to onCreate() to determine this. If savedInstanceState is null, this is the first onCreate(). If the activity is being reconstructed due to a configuration change such as rotation, the instance state bundle contains data.

1. Locate the part of the onCreate() method that tests if the savedInstanceState argument is null:

```
if (savedInstanceState != null) { //...
```

2. Add an else block to that if statement:

```
if (savedInstanceState != null) {
    // ... (deleted for brevity)
}
else { // bundle is null; this is application startup
    // ... add preference code here
}
```

3. Inside the else block, get the count from the shared preferences and update mCount:

```
mCount = mPreferences.getInt("count", 0);
```

Note that the getInt() method takes two arguments: one for the key, and the other for the default value if the key cannot be found. With the default argument you don't have to test whether the preference exists in the file.

4. Update the main text view to display the count:

```
mShowCount.setText(String.format("%s", mCount));
```

5. Get the color from the shared preferences and update mcurrentColor:

```
mCurrentColor = mPreferences.getInt("color", mCurrentColor);
```

6. Update the main text view to display the background color:

```
mShowCount.setBackgroundColor(mCurrentColor);
```

7. Run the app. Click the count button and change the background color to update the instance state and the preferences.

8. Force-quit the app using one of these methods:

- In Android Studio, select Run > Stop 'app.'
- On the device, click the Recents button (the square button in the lower right corner). Swipe the card for the HelloSharedPrefs app to quit, or click the X in the right corner.

9. Re-run the app. The app restarts and loads the preferences, maintaining the state.

### Solution Code (Main Activity - onCreate())

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    mShowCount = (TextView) findViewById(R.id.textview);  
    mCurrentColor = ContextCompat.getColor(this, R.color.default_background);  
    mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);  
  
    /* Restore the saved state. See onSaveInstanceState() for what gets saved. */  
    if (savedInstanceState != null) {  
  
        mCount = savedInstanceState.getInt("count");  
        if (mCount != 0) {  
            mShowCount.setText(String.format("%s", mCount));  
        }  
  
        mCurrentColor = savedInstanceState.getInt("color");  
        mShowCount.setBackgroundColor(mCurrentColor);  
    }  
    else {  
        // Restore preferences  
        mCount = mPreferences.getInt("count", 0);  
        mShowCount.setText(String.format("%s", mCount));  
        mCurrentColor = mPreferences.getInt("color", mCurrentColor);  
        mShowCount.setBackgroundColor(mCurrentColor);  
    }  
}
```

## Task 3. Add a Reset button

The HelloSharedPrefs app automatically saves both the instance state and the preferences any time the activity is paused or restarted. In this task we'll add a button to the app that resets the count and the background color, and clears the preferences.

### 3.1 Update the layout

In the current app layout the Count button at the bottom of the screen fills the full width of the display. To add a new button for Reset we'll add both the new button and a new linear layout to display the Count and Reset buttons side by side.

1. Open the layout file for the main activity (`activity_main.xml`).
2. Scroll down to definition of the Count button at the end of the layout. It looks like this:

```
<Button
    android:id="@+id/button5"
    style="@style/AppTheme.Button"
    android:onClick="countUp"
    android:text="@string/count_button" />
```

3. Copy the definition for the Count button, and paste a new copy just below that button.
4. Modify the new button to have these attributes:

Attribute	Value
android:id	"@+id/button6"
android:onClick	"reset"
android:text	"reset"

Extract the string resource.

5. Nest the two buttons inside a LinearLayout definition:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="10"
    android:gravity="center"
    android:orientation="horizontal">
```

6. Add layout\_marginRight and layout\_marginEnd attributes to the Count button:

```
    android:layout_marginRight="@dimen/activity_horizontal_margin"
    android:layout_marginEnd="@dimen/activity_horizontal_margin"
```

7. Open styles.xml and change these items in the AppTheme.Button style:

Item Name	Value
"android:layout_width"	"wrap_content"
"android:layout_height"	"wrap_content"
"android:layout_weight"	(delete this entire item)

### Solution Code (activity\_main.xml - partial)

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="10"
    android:gravity="center"
    android:orientation="horizontal">

    <Button
        android:id="@+id/button5"
        style="@style/AppTheme.Button"
        android:layout_marginEnd="@dimen/activity_horizontal_margin"
        android:layout_marginRight="@dimen/activity_horizontal_margin"
        android:onClick="countUp"
        android:text="@string/count_button" />

    <Button
        android:id="@+id/button6"
        style="@style/AppTheme.Button"
        android:onClick="reset"
        android:text="@string/reset_button" />
</LinearLayout>
```

### Solution Code (styles.xml - partial)

```
<style name="AppTheme.Button" parent="Widget.AppCompat.Button">
    <item name="android:background">@color/grey_background</item>
    <item name="android:textColor">@android:color/black</item>
    <item name="android:layout_width">wrap_content</item>
    <item name="android:layout_height">wrap_content</item>
</style>

<style name="AppTheme.Button.Colored" parent="AppTheme.Button">
    <item name="android:textColor">@android:color/white</item>
    <item name="android:layout_width">0dp</item>
    <item name="android:layout_height">match_parent</item>
    <item name="android:layout_weight">1</item>
</style>
```

### Solution Code (strings.xml)

```
<resources>
    <string name="app_name">HelloSharedPrefs</string>
    <string name="blue_button">Blue</string>
    <string name="green_button">Green</string>
    <string name="red_button">Red</string>
    <string name="black_button">Black</string>
    <string name="default_count">0</string>
    <string name="count_button">Count</string>
    <string name="reset_button">Reset</string>
</resources>
```

## 3.2 Implement the reset() click handler

1. In the layout file, place your cursor on "reset" in the onClick attribute for the reset button.
2. Press **Alt-Enter (Option-Enter on the Mac)**, and select **Create onClick event handler**.

Android Studio inserts a skeleton reset() method in your MainActivity.java file.

3. Reset the mCount variable to 0 and update the main text view:

```
mCount = 0;
mShowCount.setText(String.format("%s", mCount));
```

4. Reset the mCurrentColor variable to the default color (from color.xml) and update the main text view:

```
mCurrentColor = ContextCompat.getColor(this,
R.color.default_background);
mShowCount.setBackgroundColor(mCurrentColor);
```

5. Get a preferences editor:

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
```

6. Delete all the shared preferences:

```
preferencesEditor.clear();
```

7. Apply the changes:

```
preferencesEditor.apply();
```

**Solution Code (reset() method) :**

```
public void reset(View view) {
    mCount = 0;
    mShowCount.setText(String.format("%s", mCount));

    mCurrentColor = ContextCompat.getColor(this, R.color.default_background);
    mShowCount.setBackgroundColor(mCurrentColor);

    SharedPreferences.Editor preferencesEditor = mPreferences.edit();
    preferencesEditor.clear();
    preferencesEditor.apply();
}
```

## Coding challenge

Modify the HelloSharedPreferences app so that instead of automatically saving the state to the preferences file, there is a second activity to change, reset, and save the preferences.

Add an item to the action bar for Settings, and a second activity to hold those settings. The settings activity should:

- Read the count and color items stored in the shared preferences.
- Display UI elements such as toggle buttons and spinners to modify those preferences.
- Include Save and Reset buttons for saving and clearing the preferences.

## Conclusion

In this chapter, you learned about using shared preferences in your app, including how to get a shared preferences file for your app, how to write data to and read data from the shared preferences, and how to clear the shared preferences.

## Resources

- [Saving Data](#) (Android Guides)
- [Storage Options](#) (Android Guides)
- [Saving Key-Value Sets](#) (Android Training)
- [SharedPreferences](#) (Android API Reference)
- [SharedPreferences.Editor](#) (Android API Reference)
- [How to use SharedPreferences in Android to store, fetch and edit values](#) (Stack Overflow)
- [onSavedInstanceState vs. SharedPreferences](#) (Stack Overflow)



# 11.1 P: SQLite Database

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 0. Download and run the base code
- Task 1. Create a data model for word list data
- Task 2: Extend SQLiteOpenHelper to create and populate the database
- Task 3: Display the data in the RecyclerView
- Task 4: Edit words in the UI and store changes in the database
- Task 5: Create UI Elements
- Task 6: Handle Clicks
- Summary
- Resources

An SQLite database is a good storage solution when you have structured data that you need to store persistently and access, search, and change frequently.

When you use an SQLite database, all interactions with the database are through an instance of the [SQLiteOpenHelper](#) class which executes your requests and manages your database for you.

In this series of practicals, you will create a SQLite database for a set of data, display retrieved data in a RecyclerView, and add functionality to add, delete, and edit data in the RecyclerView and store it in the database.

**Note:** Adding a database to persistently store your data, and abstracting your data into a data model, are sufficient for small apps with minimal complexity. In later chapters, you will learn to architect your app using loaders and content providers to further separate data from the user interface and move work off the UI thread, all with the goal of making the user's experience as smooth and natural as possible, and retain the developer's ability to extend and maintain the app.

**Important:** In this practical, the SQLiteOpenHelper executes database operations in the main thread. In a production app, where database operations might take quite some time, you would perform these operations on a background thread, for example, using AsyncTask and CursorLoader. Check out the Coding Challenge if you want to tackle this.

# What you should already KNOW

For this practical you should be familiar with:

- Creating, building, and running apps in Android Studio.
- Displaying data in a RecyclerView.
- Using adapters as intermediaries between data and views.
- Adding onClick event handlers to views. Creating onClick handlers dynamically.
- Starting and returning data from a second activity.
- Passing data between activities using intent extras.
- Using an EditText view to get data entered by the user.

You also need a basic understanding of SQL databases, how they are organized into tables of rows and columns, and the SQL language. See the SQLite Primer

# What you will LEARN

- How to create and manage a SQLite database with an SQLiteOpenHelper.
- How to implement insert, delete, update, and query functionality through your open helper and connect it with the user interface through an adapter and custom click handler.

# What you will DO

You start with a an app that is the same as the RecyclerView word list app you created previously, with additional user interface elements already added for you, so that you can focus on the database code.

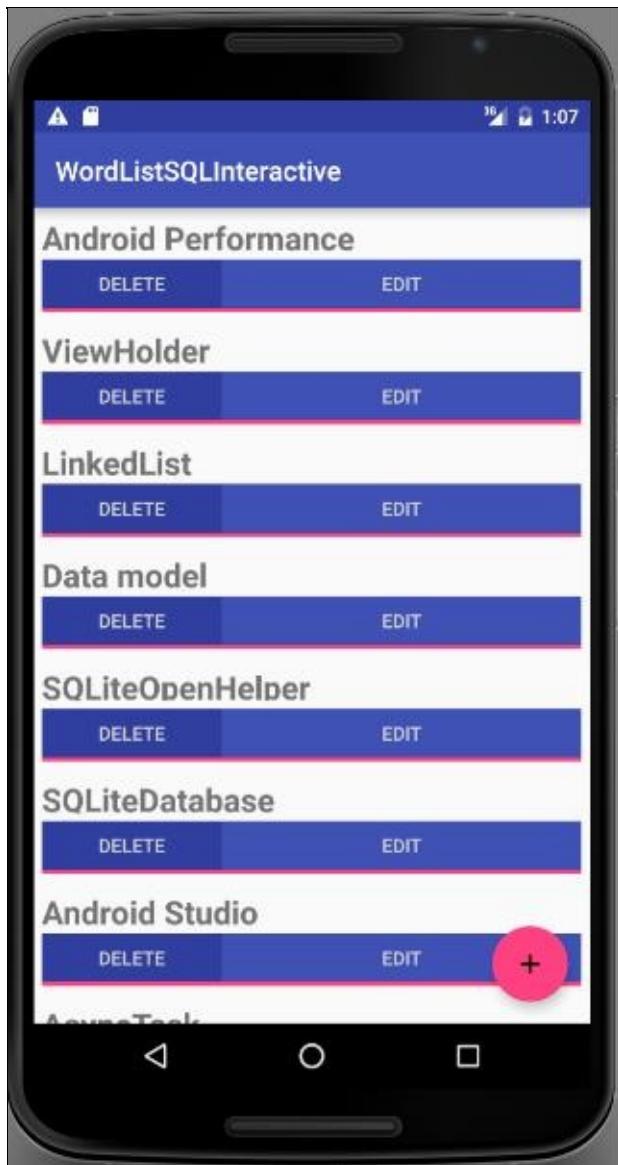
You will extend and modify the base app to:

- Implement a custom class to model your data.
- Create a subclass of SQLiteOpenHelper that creates and manages your app's database.
- Display data from the database in the RecyclerView.
- Implement functionality to add, modify, and delete data in the UI, and store the changes in the database.

# App Overview

Starting from a skeleton app, you will add functionality to:

- Display words from an SQLite database in a RecyclerView.
- Each word can be edited or deleted.
- You can add new words and store them in the database.



Minimum SDK Version is API15: Android 4.0.3 IceCreamSandwich and \*target\* SDK is the current version of Android (version 23 as of the writing of this book).

## Task 0. Download and run the starter code

In order to save you some work, in particular writing database-unrelated activities and user interface code, you need to get the starter code for this practical.

1. Download the starter code at XYZ .
2. Open the app in Android Studio.
3. Run the app. You should see the UI as shown in the previous screenshot. All the displayed words should be "placeholder". Clicking the buttons does nothing.

# Task 1. Extend SQLiteOpenHelper to create and populate a database

Android apps can use standard SQLite databases to store data. This practical does not teach SQLite, but shows how to use it in an Android app. For info on learning about SQLite, see the SQL Primer.

[SQLiteOpenHelper](#) is a utility class in the Android SDK for interacting with an [SQLite database](#) object. It includes `onCreate()` and `onUpdate()` methods that you must implement, and `insert`, `delete`, `update`, and `query` methods for all your database interactions.

The `SQLOpenHelper` class takes care of opening the database if it exists, creating it if it does not, and upgrading it as necessary. Transactions are used to make sure the database is always in a sensible state.

**Note:** You can have more than one database per app, and more than one open helper managing them; but if possible, instead of multiple databases, consider creating multiple tables in the same database.

## 1.1 Create a skeleton WordListOpenHelper class

The first step in adding a database to your code is always to create a subclass of `SQLiteOpenHelper` and implement its methods.

1. Create a new Java class `WordListOpenHelper` with the following signature.

```
public class WordListOpenHelper extends SQLiteOpenHelper {}
```

2. In the code editor, hover over the error, then click the light bulb and select **Implement methods**. Make sure both methods are highlighted and click **OK**.
3. Add the missing constructor for `WordListOpenHelper`. (You will define the undefined constants next.)

```
public WordListOpenHelper(Context context) {  
    super(context, DATABASE_NAME, null, DATABASE_VERSION);  
}
```

## 1.2. Add database constants to WordListOpenHelper

1. At the top of the `WordListOpenHelper` class, define the constants for the tables, rows, and columns as shown in the code below. This should get rid of all the errors.

```

// It's a good idea to always define a log tag like this.
private static final String TAG = WordListOpenHelper.class.getSimpleName();

// has to be 1 first time or app will crash
private static final int DATABASE_VERSION = 1;
private static final String WORD_LIST_TABLE = "word_entries";
private static final String DATABASE_NAME = "wordlist";

// Column names...
private static final String KEY_ID = "_id";
private static final String KEY_WORD = "word";

// ... and a string array of columns.
private static final String[] COLUMNS = { KEY_ID, KEY_WORD };

```

2. Run your code to make sure it has no more errors.

### 1.3. Build the SQL query and code to create the database

SQL queries can become quite complex. It is a best practice to construct the queries separately from the code that uses them. This increases code readability and helps with debugging.

Continue adding code to WordListOpenHelper.java:

1. Below the constants, add the following code to construct the query. Refer to the SQLite Primer if you need help understanding this query.

```

// Build the SQL query that creates the table.
private static final String WORD_LIST_TABLE_CREATE =
    "CREATE TABLE " + WORD_LIST_TABLE + " (" +
        KEY_ID + " INTEGER PRIMARY KEY, " +
        // id will auto-increment if no value passed
        KEY_WORD + " TEXT );";

```

2. Add instance variables for the references to writable and readable databases. Storing these references saves you the work of getting a database reference every time you need to read or write.

```

private SQLiteDatabase mWritableDatabase;
private SQLiteDatabase mReadableDB;

```

3. In the onCreate method, add code to create a database and the table (The helper class does not create another database, if one already exists.)

```

@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL(WORD_LIST_TABLE_CREATE);
}

```

4. Fix the error by renaming the method argument from sqLiteDatabase to db.

## 1.4 Create the database in onCreate of the MainActivity

To create the database, create an instance of the WordListOpenHelper class you just wrote.

1. Open MainActivity.java and add an instance variable for the open helper:

```
private WordListOpenHelper mDB;
```

2. In onCreate, initialize mDB with an instance of WordListOpenHelper. This calls onCreate of the WordListOpenHelper, which creates the database.

```
mDB = new WordListOpenHelper(this);
```

3. Add a breakpoint, run the app with the debugger, and check that mDB is an instance for WordListOpenHelper.

## 1.5 Add data to the database

The list of words for your app could come from many sources. It could be completely user created, or downloaded from the internet, or generated from a file that's part of your APK. For this practical, you will seed your database with a small amount of hard-coded data.

Note that acquiring, creating, and formatting data is a whole separate topic that is not covered in this book.

1. Open WordListOpenHelper.java.
2. In onCreate, after creating the database, add a function call to

```
fillDatabaseWithData(db);
```

Next, implement the `fillDatabaseWithData()` method in WordListOpenHelper.

3. Implement the method stub.

```
private void fillDatabaseWithData(SQLiteDatabase db){}
```

4. Declare a string of words as your mock data.

```
String[] words = {"Android", "Adapter", "ListView", "AsyncTask",
    "Android Studio", "SQLiteDatabase", "SQLOpenHelper",
    "Data model", "ViewHolder", "Android Performance",
    "OnClickListener"};
```

5. Create a container for the data. The insert method that you will call next requires the values to fill a row as an instance of [ContentValues](#). A ContentValues stores the data for one row as key-value pairs, where the key is the name of the column and the value is the value to set,

```
// Create a container for the data.
ContentValues values = new ContentValues();
```

6. Add key/value for the first row to values, then insert that row into the database. Repeat for all the words in your array of words.

- `db.insert()` is a [SQLiteDatabase](#) convenience method to insert one row into the database. (It's a convenience method, because you do not have to write the SQL query yourself.)
- The first argument to `db.insert` is the table name, `WORD_LIST_TABLE`.
- The second argument is a `String nullColumnHack`. It's a hack that allows you to insert empty rows. See [the documentation for insert\(\)](#). Use null for this argument.
- The third argument must be a [ContentValues](#) container with values to fill the row. This sample only has one column "words" as represented by the constant `KEY_WORD` set earlier; for tables with multiple columns, add the values for each column to this container.

```
for (int i=0; i < words.length; i++) {
    // Put column/value pairs into the container.
    // put() overrides existing values.
    values.put(KEY_WORD, words[i]);
    db.insert(WORD_LIST_TABLE, null, values);
}
```

7. After adding this code, you need to destroy the current empty database and then recreate it so that the database is initialized with the seed data. You can uninstall the app from your device, or you can clear all the data in the app from **Settings > Apps > WordList > Clear Data**.

8. Run your app. You will not see any changes in the user interface.
  - Check the logs and make sure there are no errors before you continue. If you encounter errors, read the logcat messages carefully and use resources, such as Stack Overflow, if you get stuck.
  - You can also check in settings, that the app users storage.

## Task 2. Create a data model for a single word

A data model is a class that encapsulates a complex data structure and provides an API for accessing and manipulating the data in that structure. You need this to pass data retrieved from the database to the UI.

For this practical, the data model only contains the word and its id. While the unique id will be generated by the database, you need a way of passing the id to the user interface, so that you can identify the word that the user is changing.

### 2.1. Create a data model for your word data

1. Create a new class and call it `WordItem`.
2. Add the following class variables.

```
private int mId;  
private String mWord;
```

3. Add an empty constructor.
4. Add getters and setters for the id and word.
5. Run your app. You will not see any visible UI changes, but there should be no errors.

#### Solution:

```
public class WordItem {  
  
    private int mId;  
    private String mWord;  
  
    public WordItem() {}  
  
    public int getId() {return this.mId;}  
  
    public String getWord() {return this.mWord;}  
  
    public void setId(int id) {this.mId = id;}  
  
    public void setWord(String word) {this.mWord = word;}  
}
```

## Task 3. Implement the query() method in WordListOpenHelper

The query() method retrieves rows from the database as selected by an SQL query. The most generalized query method would allow any SQL.

For this sample, in order to display words in the RecyclerView, we need to get them from the database, one at a time, as needed. The word needed is identified by its position in the view. We don't need any other functionality.

As such, the query methods has a parameter for the requested position, and returns a WordItem.

### 3.1. Implement the query() method

1. Create a query method that takes an integer position argument and returns a WordItem.

```
public WordItem query(int position) {  
}
```

2. Construct a query that returns only the nth row of the result. Use LIMIT with position as the row, and 1 as the number of rows.

```
String query = "SELECT * FROM " + WORD_LIST_TABLE +  
    " ORDER BY " + KEY_WORD + " ASC " +  
    "LIMIT " + position + ",1";
```

3. Instantiate a Cursor variable to null to hold the result from the database.

```
Cursor cursor = null;
```

The SQLiteDatabase always presents the results as a [Cursor](#) in a table format that resembles of a SQL database.

A cursor is a pointer into a row of structured data. You can think of it as an array of rows. The Cursor class provides methods for moving the cursor through that structure, and methods to get the data from the columns of each row.

4. Instantiate a WordItem entry.

```
WordItem entry = new WordItem();
```

5. Add a try/catch/finally block.

```
try {} catch (Exception e) {} finally {}
```

6. Inside the try block,

- i. get a readable database if it doesn't exist.

```
if (mReadableDB == null) {  
    mReadableDB = getReadableDatabase();  
}
```

- ii. send a raw query to the database and store the result in a cursor.

```
cursor = mReadableDB.rawQuery(query, null);
```

The open helper query method can construct an SQL query string and send it as a rawQuery to the database which returns a cursor. If your data is supplied by your app, and under your full control, you can use raw query().

- iii. Move the cursor to the first item.

```
cursor.moveToFirst();
```

- iv. set the id and word of the WordItem entry to the values returned by the cursor.

```
entry.setId(cursor.getInt(cursor.getColumnIndex(KEY_ID)));  
entry.setWord(cursor.getString(cursor.getColumnIndex(KEY_WORD)));
```

7. In the catch block, log the exception.

```
Log.d(TAG, "EXCEPTION! " + e);
```

8. In the finally block, close the cursor and return the WordItem entry.

```
cursor.close();  
return entry;
```

## Solution:

```

public WordItem query(int position) {
    String query = "SELECT * FROM " + WORD_LIST_TABLE +
        " ORDER BY " + KEY_WORD + " ASC " +
        "LIMIT " + position + ",1";

    Cursor cursor = null;
    WordItem entry = new WordItem();

    try {
        if (mReadableDB == null) {
            mReadableDB = getReadableDatabase();
        }
        cursor = mReadableDB.rawQuery(query, null);
        cursor.moveToFirst();
        entry.setId(cursor.getInt(cursor.getColumnIndex(KEY_ID)));
        entry.setWord(cursor.getString(cursor.getColumnIndex(KEY_WORD)));
    } catch (Exception e) {
        Log.d(TAG, "EXCEPTION! " + e);
    } finally {
        cursor.close();
        return entry;
    }
}

```

## 3.2. The onUpgrade method

Every SQLiteOpenHelper must implement the onUpgrade method, which determines what happens if the database version number changes. The customary default action is to delete the current database and recreate it.

**Important:** While it's OK to drop the table in a sample app, In a production app you need to carefully migrate the user's data so that it's not lost.

You can use the code below to implement the onUpgrade() method for this sample.

**Boilerplate code for onUpgrade():**

```

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    Log.w(WordListOpenHelper.class.getName(),
        "Upgrading database from version " + oldVersion + " to "
        + newVersion + ", which will destroy all old data");
    db.execSQL("DROP TABLE IF EXISTS " + WORD_LIST_TABLE);
    onCreate(db);
}

```

## Task 4. Display data in the RecyclerView

You now have a database, with data. Next, you will update the WordListAdapter and MainActivity to fetch and display this data.

## 4.1. Update WordListAdapter to display WordItems

1. Open WordListAdapter.
2. In onBindViewHolder replace the code that displays mock data with code to get an item from the database and display it. You get an error on mDB.

```
WordItem current = mDB.query(position);
holder.wordItemView.setText(current.getWord());
```

3. Declare mDB as an instance variable.

```
WordListOpenHelper mDB;
```

4. To get the value for mDB, change the constructor for WordListAdapter and add a second argument for the WordListOpenHelper.
5. Assign the value of the argument to mDB. Your constructor should look like this:

```
public WordListAdapter(Context context, WordListOpenHelper db) {
    mInflater = LayoutInflater.from(context);
    mContext = context;
    mDB = db;
}
```

This generates an error in MainActivity, because you added an argument to the WordListAdapter constructor.

6. Open MainActivity and add the missing mDB argument.

```
mAdapter = new WordListAdapter (this, mDB);
```

7. Open MainActivity and add the missing mDB argument.
8. Run your app. You should see all the words from the database.

## Task 5. Add new words to the database

When the user clicks the FAB, an activity opens that lets them enter a word that gets added to the database when they click save.

The starter code provides you with the click listener and the EditWordActivity. You will add the database specific code and tie the pieces together, from the bottom up, like you just did with the query method.

## 5.1. Write the insert() method

In WordListOpenHelper:

1. Create the insert() method with the following signature. The user supplies a word, and the method returns the id for the new entry. Generated id's can be big, so insert returns a number of type long.

```
public long insert(String word){}
```

2. Declare a variable for the id. If the insert operation fails, the method returns 0.

```
long newId = 0;
```

3. As before, create a ContentValues value for the row data.

```
ContentValues values = new ContentValues();
values.put(KEY_WORD, word);
```

4. Put your database operation into a try/catch block.

```
try {} catch (Exception e) {}
```

5. Get a writable database if one doesn't already exist.

```
if (mWritableDatabase == null) {
    mWritableDatabase = getWritableDatabase();
}
```

6. Insert the row.

```
newId = mWritableDatabase.insert(WORD_LIST_TABLE, null, values);
```

7. Log the exception.

```
Log.d(TAG, "EXCEPTION! " + e);
```

8. Return the id.

```
return newId;
```

**Solution:**

```

public long insert(String word){
    long newId = 0;
    ContentValues values = new ContentValues();
    values.put(KEY_WORD, word);
    try {
        if (mWritableDatabase == null) {
            mWritableDatabase = getWritableDatabase();
        }
        newId = mWritableDatabase.insert(WORD_LIST_TABLE, null, values);
    } catch (Exception e) {
        Log.d(TAG, "EXCEPTION! " + e);
    }
    return newId;
}

```

## 5.2. Get the word to insert from the user and update the database

The starter code comes with an `EditWordActivity` that gets a word from the user and returns it to the main activity. In `MainActivity`, you have to fill in the `onActivityResult` method.

1. Check that the result is from the right activity and get the word that the user entered from the extras.

```

if (requestCode == WORD_EDIT) {
    if (resultCode == RESULT_OK) {
        String word = data.getStringExtra(EditWordActivity.EXTRA_REPLY);
    }
}

```

2. If the word is not empty, check whether we have been passed an id with the extras. If there is no id, insert a new word. In the next task, you will update the existing word if an id is passed.

```

if (!TextUtils.isEmpty(word)) {
    int id = data.getIntExtra(WordListAdapter.EXTRA_ID, -99);
    if (id == WORD_ADD) {
        mDB.insert(word);
    }
}

```

3. Update the UI.

```
mAdapter.notifyDataSetChanged();
```

4. If the word is empty because the user didn't enter anything, show a toast letting them know. And don't forget to close all the parentheses.

```
    } else {
        Toast.makeText(
            getApplicationContext(),
            R.string.empty_not_saved,
            Toast.LENGTH_LONG).show();
    }
}
```

## Solution:

```
if (requestCode == WORD_EDIT) {  
    if (resultCode == RESULT_OK) {  
        String word = data.getStringExtra(EditWordActivity.EXTRA_REPLY);  
        // Update the database  
        if (!TextUtils.isEmpty(word)) {  
            int id = data.getIntExtra(WordListAdapter.EXTRA_ID, -99);  
            if (id == WORD_ADD) {  
                mDB.insert(word);  
            }  
            // Update the UI  
            mAdapter.notifyDataSetChanged();  
        } else {  
            Toast.makeText(  
                getApplicationContext(),  
                R.string.empty_not_saved,  
                Toast.LENGTH_LONG).show();  
        }  
    }  
}
```

### 5.3. Implement getItemCount()

In order for the new items to be displayed properly, getItemCount in WordListAdapter has to return the actual number of entries in the database instead of the number of words in the starter list of words.

1. Change getItemCount to the code below, which will trigger an error.

```
return (int) mDB.count();
```

2. Open WordListOpenHelper and implement count() to return the number of entries in the database.

```

public long count(){
    if (mReadableDB == null) {
        mReadableDB = getReadableDatabase();
    }
    return DatabaseUtils.queryNumEntries(mReadableDB, WORD_LIST_TABLE);
}

```

3. Run your app and add some words.

## Task 6. Delete words from the database

To implement the delete functionality you need to:

- Add a click handler to the DELETE button in WordListAdapter
- Implement the delete() method in WordListOpenHelper

### 6.1. Write the delete() method

You use the delete() method on SQLiteDatabase to delete an entry in the database.

Add a method delete to the WordListOpenHelper that:

1. Create the method stub for delete(), which takes an int argument for the id of the item to delete, and returns the number of rows deleted.

```
public int delete(int id) {}
```

2. Declare a variable to hold the result.

```
int deleted = 0;
```

3. As for insert, add a try block.

```
try {} catch (Exception e) {}
```

4. Get a writable database, if necessary.

```

if (mWritableDatabase == null) {
    mWritableDatabase = getWritableDatabase();
}

```

5. Call delete on the WORD\_LIST\_TABLE, selecting by KEY\_ID and passing the value of the id as the argument.

```
deleted = mWritableDatabase.delete(WORD_LIST_TABLE,
    KEY_ID + " =? ", new String[]{String.valueOf(id)});
```

6. Print a log message for exceptions.

```
Log.d (TAG, "EXCEPTION! " + e);
```

7. Return the number of rows deleted.

```
return deleted;
```

### Solution:

```
public int delete(int id) {
    int deleted = 0;
    try {
        if (mWritableDatabase == null) {
            mWritableDatabase = getWritableDatabase();
        }
        deleted = mWritableDatabase.delete(WORD_LIST_TABLE, //table name
            KEY_ID + " =? ", new String[]{String.valueOf(id)});
    } catch (Exception e) {
        Log.d (TAG, "EXCEPTION! " + e);
    }
    return deleted;
}
```

## 6.2. Add a click handler to DELETE button

You can now add a click handler to the DELETE button that calls the delete() method you just wrote.

Take a look at the MyButtonClickListener class in your starter code. The MyButtonClickListener class implements a click listener that stores the position in the view, the id, and the word that you need to make changes to the database.

Each view holder, when bound in onBindViewHolder, attaches a click listener to the DELETE button, passing the current position, id, and word to the constructor. These values are then used by the onClick handler to delete the correct item and notify the adapter, which item has been removed.

### Solution:

```
// Attach a click listener to the DELETE button.
holder.delete_button.setOnClickListener(new MyButtonOnClickListener(
    position, current.getId(), null) {
    @Override
    public void onClick(View v) {
        Log.d(TAG, " " + position + " " + id);
        int deleted = mDB.delete(id);
        if (deleted >= 0)
            notifyItemRemoved(position);
    }
});
```

## Task 7. Update words in the database

To update existing words you have to:

- Add an update() method to WordListOpenHelper.
- Add a click handler to the EDIT button of your view.

### 7.1. Write the update() method

You use the update() method on SQLiteDatabase to update an existing entry in the database.

1. Add a method to the WordListOpenHelper that:

- Takes an integer id and a String word for its arguments and returns an integer.

```
public int update(int id, String word)
```

- Initializes int mNumberOfRowsUpdated to -1.

```
int mNumberOfRowsUpdated = -1;
```

2. Inside a try block:

3. Get a writable SQLiteDatabase db if there isn't one already.

```
if (mWritableDatabase == null) {
    mWritableDatabase = getWritableDatabase();
}
```

4. Create a new instance of ContentValues and at the KEY\_WORD word to it.

```
ContentValues values = new ContentValues();
values.put(KEY_WORD, word);
```

5. Call db.update using the following arguments:

```
mNumberOfRowsUpdated = db.update(WORD_LIST_TABLE,
    values, // new values to insert
    // selection criteria for row (the _id column)
    KEY_ID + " = ?",
    //selection args; value of id
    new String[]{String.valueOf(id)});
```

6. In the catch block, print a log message if any exceptions are encountered.

```
Log.d (TAG, "EXCEPTION: " + e);
```

7. Return the number of rows updated, which should be -1 (fail), 0 (nothing updated), or 1 (success).

```
return mNumberOfRowsUpdated;
```

### Solution:

```
public int update(int id, String word) {
    int mNumberOfRowsUpdated = -1;
    try {
        if (mWritableDatabase == null) {
            mWritableDatabase = getWritableDatabase();
        }
        ContentValues values = new ContentValues();
        values.put(KEY_WORD, word);
        mNumberOfRowsUpdated = mWritableDatabase.update(WORD_LIST_TABLE,
            values,
            KEY_ID + " = ?",
            new String[]{String.valueOf(id)});
    } catch (Exception e) {
        Log.d (TAG, "EXCEPTION! " + e);
    }
    return mNumberOfRowsUpdated;
}
```

## 7.2. Add a click listener to the EDIT button

And here is the code for the click listener. This has nothing database specific. It starts EditWordActivity and passes it the current id, position, and word.

If you get an error on the EXTRA\_POSITION constant, add it with a value of "POSITION",

### Solution:

```

// Attach a click listener to the EDIT button.
holder.edit_button.setOnClickListener(new MyButtonOnClickListener(
    position, current.getId(), current.getWord()) {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(mContext, EditWordActivity.class);
        intent.putExtra(EXTRA_ID, id);
        intent.putExtra(EXTRA_POSITION, position);
        intent.putExtra(EXTRA_WORD, word);

        // Start an empty edit activity.
        ((Activity) mContext).startActivityForResult(intent, MainActivity.WORD_EDI
T);
    }
});
```

## 7.3. Add updating to onActivityResult

As implemented, clicking edit starts an activity that shows the user the current word, and they can edit it. To make the update happen,

1. Add one line of code to onActivityResult.

```

else if (id >= 0) {
    mDB.update(id, word);
}
```

2. Run your app and play with it!

## 7.4. Design and error considerations

- The methods you wrote to add, update and delete entries in the database all assume that their input is valid. This is acceptable for sample code because the purpose of this sample code is to teach you the basic functionality of an SQLite database, and so not every edge case is considered, not every value is tested, and everybody is assumed to be well behaved.
- In a production app, you must catch specific exceptions and handle them appropriately.
- You tested the correct functioning of the app by running it. For a production app with real data, you will need more thorough testing, for example, using unit and interface testing.
- For this practical, you created the the database schema/tables from the SQLiteOpenHelper class. This is sufficient for a simple example, like this one. For a more complex app, it is a better practice to separate the schema definitions from the rest of the code in a helper class that cannot be instantiated. You will learn how to do

that in the chapter on content providers.

- As mentioned above, some database operations can be lengthy and should be done on a background thread. Use AsyncTask for operations that take a long time. Use loaders to load large amounts of data - you will learn about loaders in a later chapter.

## Coding challenges

- Extend the app to have a definition for each word.

## Summary

In this chapter, you learned how to

- Use a [SQLiteDatabase](#) to store user data persistently.
- Work with an [SQLiteOpenHelper](#) to manage your database.
- Retrieve and display data from the database
- Edit data in the user interface and reflect those changes in the database

## Resources

### Developer Documentation:

- [Storage Options](#)
- [Saving Data in SQL Databases](#)

# 11.2 P: Searching an SQLite Database

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 0. Download and run the base code
- Task 1. Add a Search Menu Item
- Task 2: Add a search activity
- Task 3: Display the data in the RecyclerView
- Conclusion
- Resources

## What you should already KNOW

For this practical you should be familiar with:

- SQLite database
- Writing basic SQLite queries

## What you will LEARN

- How to add search functionality to your app via the options menu, and search the database for words that match a substring entered by the user.
- How to build search queries for the SQLite database from user input.

## What you will DO

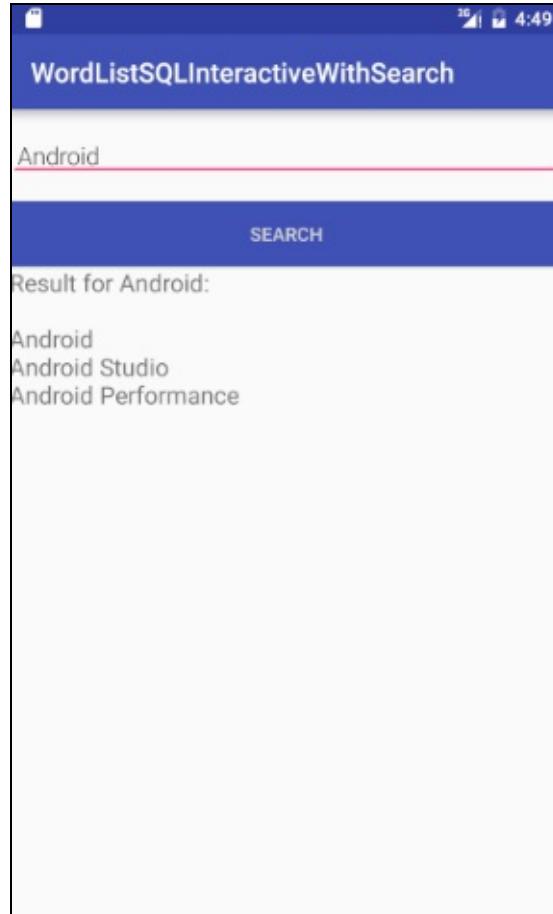
In this practical you will add an options menu item for searching, and an activity that allows users to enter a search string and displays the result of the search in a text view.

**Why:** Users should always be able to search the data on their own terms.

Note that our concern is not building a spiffy search UI, but showing you how to query the database.

# App Overview

Starting from the WordListSQLInteractive app, you will add an activity that lets users search for partial and full words in the database. For example, entering "Android" will return all



entries that contain the substring "Android".

## Task 0. Download and run the base code

In order to save you some work, this practical will build on an app you have already built. In addition, rearchitecting existing application code to add features or fix problems is a common developer task.

### 0.1. Create your project

1. Download the WordListSQLInteractive base app. You can find the code at XYZ. You can use your own app, or download the base app. As long as the app uses an SQLite database, you can use these instructions to extend it.
2. Load a copy of the app into Android Studio. Refer to the appendix for information on copying a project.
3. Rename the package using Refactor > Rename.

4. Change the package name in your build.gradle file.
5. Run the app.

## Task 1. Add Search

### 1.1. Add an Options Menu with Search item

Use the OptionsMenuSample code from the previous practicals if you need an example of how to do this.

1. In your project, create an Android Resource directory and call it menu with "menu" as the resource type (**res > menu**).
2. Add a main\_menu.xml menu resource file to res > menu.
3. Create a menu with one item **Search**. Reference the code snippet for values.

```
<menu
    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"
    tools:context="com.android.example.wordlistsqsearchable.MainActivity">

    <item
        android:id="@+id/action_search"
        android:title="Search..."
        android:orderInCategory="1"
        app:showAsAction="never" />
</menu>
```

4. In MainAcvitiy, inflate the menu by overriding onCreateOptionsMenu.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}
```

5. Override onOptionsItemSelected method. Switch on action\_search, and just return true.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_search:
            return true;
    }
    return super.onOptionsItemSelected(item);
}
```

6. Run your app. You should see the dots for the options menu. When you click it, you should see one menu item for search that does nothing.

## 1.2. Create the layout for the search activity

This layout is similar to activity\_edit\_word, so you can take advantage of existing code and copy it.

1. Create a copy of activity\_editword and call it activity\_search.xml.
2. In activity\_search.xml., change the id's and strings to be representative of searching.
3. Change the onClick method for the button to showResult.
4. Add a TextView with an id of search\_result, at least 300dp height, and 18sp font size.
5. Run your app. You should notice no difference.

## 1.3. Add an Activity for searching

1. Create a new activity, SearchActivity. If you create it by New > Android > Activity then DON'T generate the layout file because we created it in the previous task.
2. Add a private TextView class variable mTextView.
3. Add a private EditText class variable mEditWordView.
4. Add a private TextView class variable mTextView.
5. Add a private WordListOpenHelper variable mDB.
6. In onCreate, initialize mDB with a new WordListOpenHelper(this).
7. In onCreate, initialize mTextView and mEditWordView to their respective views.

```
public class SearchActivity extends AppCompatActivity {

    private static final String TAG = EditWordActivity.class.getSimpleName();

    private TextView mTextView;
    private EditText mEditWordView;
    private WordListOpenHelper mDB;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_search);

        mEditWordView = ((EditText) findViewById(R.id.search_word));
        mTextView = ((TextView) findViewById(R.id.search_result));
        mDB = new WordListOpenHelper(this);
    }
}
```

8. Add the activity to the AndroidManifest.

```
<activity
    android:name="com.android.example.wordlistsqllsearchable.SearchActivity">
</activity>
```

## 1.4. Trigger SearchActivity from the menu

1. To start SearchActivity when the menu item is selected, insert code to start SearchActivity into the switch statement in the onOptionSelected() method in MainActivity.

```
Intent intent = new Intent(getApplicationContext(), SearchActivity.class);
startActivity(intent);
```

2. Build and run your app to make sure the activity is fired off when the menu item is selected.
3. Enter a search string and press "Search". Your app crashes.
4. Find out why the app has crashed, then move to the next task.

## 1.5. Implement the onClick handler for the Search button

Your app crashed, because the onClick handler set for the Search button in the XML code doesn't exist yet. So you will build showResult next.

When the Search button is pressed, several things need to happen:

- The event handler calls public void showResult(View view) in SearchActivity.
- Your app has to get the current value from the mEditText, which is your search string.
- You print the "Result for" and the word in mTextView.
- You call the (not yet written) search function on mDB (mDB.search(word)) and get back a cursor. You will implement the search function in the next task.
- You process the cursor and add the result to mTextView.
- In SearchActivity, create the showResult function. It is public, takes a View argument, and returns nothing.
- Create a `String word` and initialize it with the contents of the input edit text view, `mEditText`.
- Show the search term in the search results text view; something like

```
"Search term: " + word
```

- Search the database and get the cursor.

```
Cursor cursor = mDB.search(word);
```

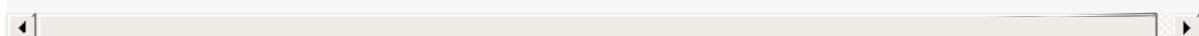
- To process the cursor, you need to do the following:
  1. Make sure the cursor is not null.
  2. Move the cursor to the first entry.
  3. Iterate over the cursor processing the current entry, then advancing the cursor.
  4. Extract the word.
  5. Display the word in the text view.
- Close the cursor.
- Check the annotated code for additional details.

```
public void showResult(View view){
    String word = mEditWordView.getText().toString();
    mTextView.setText("Result for " + word + ":\\n\\n");

    // Search for the word in the database.
    Cursor cursor = mDB.search(word);

    // Only process a non-null cursor with rows.
    if (cursor != null & cursor.getCount() > 0) {
        // You must move the cursor to the first item.
        cursor.moveToFirst();
        int index;
        String result;
        // Iterate over the cursor, while there are entries.
        do {
            // Don't guess at the column index. Get the index for the named column.
            index = cursor.getColumnIndex(WordListOpenHelper.KEY_WORD);
            // Get the value from the column for the current cursor.
            result = cursor.getString(index);
            // Add result to what's already in the text view.
            mTextView.append(result + "\\n");
        } while (cursor.moveToNext()); // This call returns true or false
        cursor.close();
    } // You should add some handling of null case. Right now, nothing happens.

}
```



Your app will not run without at least a stub for search() implemented. Android Studio will create the stub for you. In the light bulb, choose create method.

- Open WordListOpenHelper.
- Implement a stub for search, with a String parameter, that returns a null cursor.
- Run your app and fix any errors you may have. Note that most of the code in

`showResult()` is not exercised yet.

## 1.6. Implement the search method in WordListOpenHelper

The final step is to implement the actual searching of the database.

Inside the `search()` method, you need to build a query with the search string and send the query to the database.

The most secure way to do this is by using parameters for each part of the query.

**WHY:** In the previous practical, for the query in `WordListOpenHelper`, you could build the query string directly and submit it as a `rawQuery()`, because you had full control over the contents of the query. As soon as you are handling user input, you must assume that it could be malicious. You should always validate user input even before you build your query!

You will learn more about security in the Security chapter and [Security Tips](#).

The SQL query for searching for all entries in the wordlist matching a substring has this form:

```
SELECT * FROM WORD_LIST_TABLE WHERE KEY_WORD LIKE %searchString%;
```

The parametrized form of the query method you will call looks like this:

```
Cursor query (String table, // The table to query
              String[] columns, // The columns to return
              String selection, // WHERE statement
              String[] selectionArgs, // Arguments to WHERE
              String groupBy, // Grouping filter. Not used.
              String having, // Additional condition filter. Not used.
              String orderBy) // Ordering. Setting to null uses default.
```

See the [SQLite Database Android](#) and `query()` documentation for more on `query()`.

For the query in the `search()` method, you need to assign only the first four arguments.

1. The table is already defined as the `WORD_LIST_TABLE` constant.
2. In `search()`, create a variable for the columns. You need only the value from the `KEY_WORD` column.

```
String[] columns = new String[]{KEY_WORD};
```

3. Add the % to the `searchString` parameter.

```
searchString = "%" + searchString + "%";
```

4. Create the where clause. Omit "WHERE" as it's implied. Use a question mark for the argument to LIKE. Make sure you have the correct spaces.

```
String where = KEY_WORD + " LIKE ?";
```

5. Specify the argument to the where clause, which is the searchString.

```
String[]whereArgs = new String[]{searchString};
```

6. Add a Cursor cursor variable and initialize it to null.

7. In a try/catch block.

- i. Get a readable database if mReadable is not set yet.
- ii. Query the database using the above form of the query. Pass null for the unused parameters.
- iii. Handle the exception. You can just log it.

8. Return the cursor.

9. Run your app and search for some strings.

**Note:** You can send any SQLite query to the database in this way and receive the reply as a cursor.

**Here is the solution for the complete method:**

```
public Cursor search (String searchString) {
    String[] columns = new String[]{KEY_WORD};
    searchString = "%" + searchString + "%";
    String where = KEY_WORD + " LIKE ?";
    String[]whereArgs = new String[]{searchString};

    Cursor cursor = null;

    try {
        if (mReadableDB == null) {mReadableDB = getReadableDatabase();}
        cursor = mReadableDB.query(WORD_LIST_TABLE, columns, where, whereArgs, null, null, null);
    } catch (Exception e) {
        Log.d(TAG, "EXCEPTION! " + e);
    }

    return cursor;
}
```

## Coding challenges

- Most of the code samples use the default AppBar that comes with the Empty Template. In some of the previous chapters, you learned about the Toolbar, for example, when using the Basic Template.

Change the app to use the Toolbar and SearchView and show the search icon on the toolbar.

<https://developer.android.com/training/search/setup.html>

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

- As written, this app is not very secure. Consider how to add basic input validation for the search string. See and [Security Tips](#).
- Try different types of queries and other forms of the query method.

## Summary

In this chapter, you learned how to:

- Add an options menu for searching.
- Construct an SQLite search query from user input.
- Use the query() method to search that database for matching words.
- Display the result to the user.

## Resources

### Developer Documentation:

- [Storage Options](#)
- [Saving Data in SQL Databases](#)

# 12.1 P: Implement a Minimalist Content Provider

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 0. Create a MainActivity](#)
- [Task 1. Create a class that extends ContentProvider](#)
- [Task 2: Create a Contract](#)
- [Task 3: Implement getType\(\)](#)
- [Task 4: Edit words in the UI and store changes in the database](#)
- [Task 5: Create UI Elements](#)
- [Task 6: Handle Clicks](#)
- [Summary](#)
- [Resources](#)

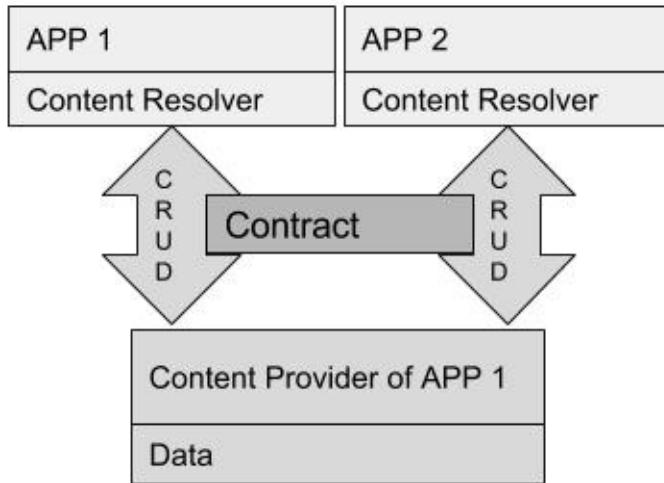
The content provider is a layer between an app's data storage backend and the user-facing parts of the app.

Content providers are useful for:

- With a content provider, you can allow multiple apps to securely access, use, and modify a single data source. Examples: Contacts, game scores, spell-checking dictionary.
- Access control. You can specify levels of permissions for your content provider.
- Storing data independently from the app. Having a content provider between your frontend and your backend allows you to change how the data is stored. For example, you can build a prototype using mock data, then use an SQL database for the real app. You could even store some of your data in the cloud and some locally, and it would be all the same to your users.
- Another benefit of separating data from the user interface is that development teams can work independently on the frontend and backends of your app. For larger, complex apps it is very common that the user interface and the data backend are developed by different teams, and they can even be separate apps. That is, it is not required that the app with the content provider has a user interface.
- Using other classes that expect to interact with a content provider. Example: You must

have a content provider to use a loader.

The following diagram shows and summarizes the parts of the content provider architecture.



**Data:** The data repository. APP 1 owns the data and specifies what permissions other apps have to work with the data.

The data is often stored in an SQLite database, but this is not mandatory. Typically, the data is presented to the content provider as tables, similar to database tables, where each row represents one entry, and each column represents an attribute for that entry. For example, each row contains one contact, and may have columns for email addresses and phone numbers.

**Content Provider of APP1:** The content provider implemented by APP 1 provides a standard CRUD (create, read, update, delete) interface to APP 1's data. In addition, it provides a public and secure interface to the data, so that other apps, such as APP 2, can access the data with the appropriate permissions.

**Contract:** The contract is a public class that exposes important information about APP 1's content provider to other apps. This usually includes the URI schemes, important constants, and the structure of the data that will be returned.

**URI scheme:** Apps send requests to the content provider using content [Uniform Resource Identifiers or URIs](#). A content URI for content providers has this general form:

- scheme (for content URI, this is always content://)
- authority (represents the domain, and for content providers customarily ends in .provider )
- path (this represents the path to the data)
- ID (uniquely identifies the data set to search; such as a file or table)

The following URI could be used to request all the entries in the "words" table:

```
content://com.android.example.wordcontentprovider.provider/words
```

Designing URI schemes is a topic in itself and not covered in this practical.

**Content Resolver:** The ContentResolver object provides query(), insert(), update(), and delete() methods for accessing data from a content provider. Thus, the content resolver mirrors the content providers CRUD API and manages all interaction with the content provider for you. In most situations, you can just use the default content resolver provided by the Android system.

If your app does not share data with other apps, then your app does not require a content provider. However, because the content provider cleanly separates the implementation of your backend from the user interface, it can also be useful for architecting complex applications.

In this practical, you will build a basic content provider from scratch. You will create and process mock data so that you can focus on understanding content provider architecture. Likewise, the user interface to display the data is minimalist. In the next practical, you will add a content provider to the WordList app, using this minimalist app as your template.

## What you should already KNOW

For this practical you should be familiar with:

- Creating, building and running interactive apps in Android Studio.
- Displaying data in a RecyclerView using an adapter.
- Abstracting and encapsulating data with data models.
- Creating, managing, and interacting with a SQLite database using an SQLiteOpenHelper.

## What you will LEARN

- Architecture and anatomy of a content provider.
- How to build a minimalist content provider that you can use as a template for creating other content providers.

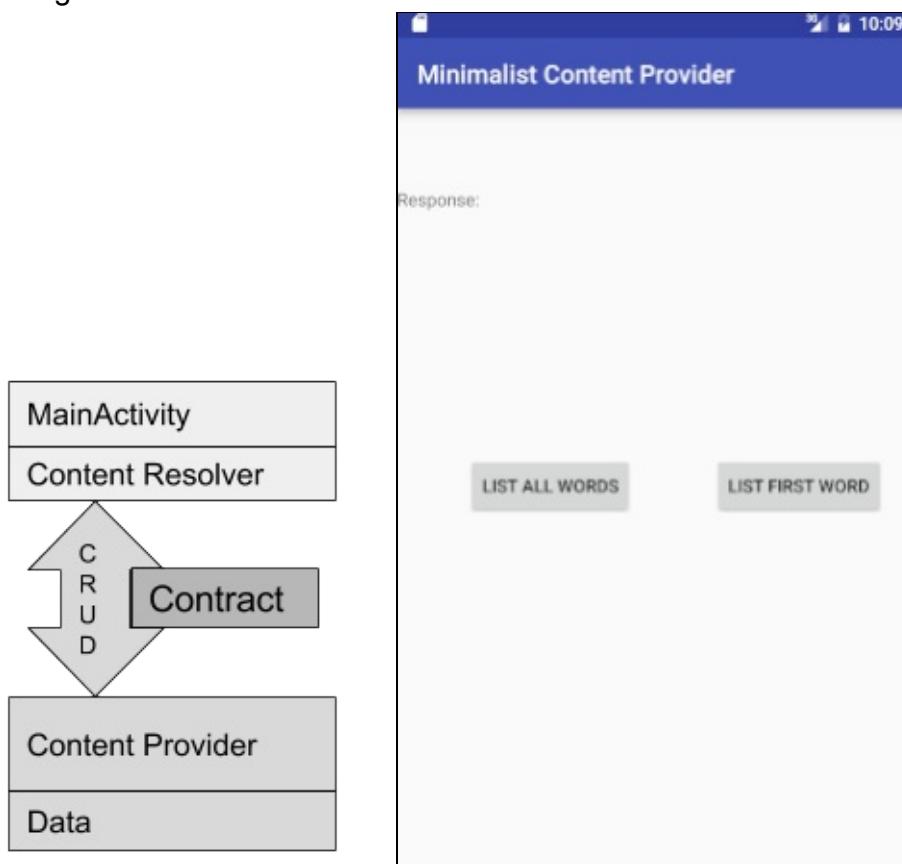
## What you will DO

You will build a minimalist stand-alone app to learn the mechanics of building a content provider.

**WHY:** One of the challenges of building a more complex app is to first thoroughly understand each piece. One way of building that understanding is by building a standalone, minimalist version around that concept, then use it as a reference for building the real thing. This technique has broad application whenever you need to learn something new that is non-trivial.

## App Overview

- The backend of this app generates mock data and stores it in a linked list, call "words".
- The app requests data through a content resolver and displays it. The UI is minimal, consisting of one activity with a TextView and two Buttons.
- In between the user interface and the backend, a content provider abstracts and manages the interaction between the backend and the frontend.



Minimum SDK Version is API15: Android 4.0.3 IceCreamSandwich and \*target\* SDK is the current version of Android (version 23 as of the writing of this book).

## Task 1. Create the MinimalistContentProvider project

By now, you are expected to be very familiar with the basics of app creation.

## 1.1 Create a project within the given constraints

Create an app with one activity that shows one text view and two buttons, one for showing the first word in the list, and the other for listing all words. Both buttons call `onClickDisplayEntries()` when they are clicked. For now, this method just logs a statement that the button was clicked.. Only the mandatory elements are listed below. You should keep it simple.

App name	MinimalistContentProvider
One Activity	Name: MainActivity private static final String TAG = MainActivity.class.getSimpleName(); public void onClickDisplayEntries (View view){Log.d (TAG, "Yay, I was clicked!");}
TextView	@+id/textview android:text="response"
Button	@+id/button_display_all android:text="List all words" android:onClick="onClickDisplayEntries"
Button	@+id/button_display_first android:text="List first word" android:onClick="onClickDisplayEntries"

## 1.2. Complete the basic setup

Complete the basic setup of the user interface:

1. In the `MainActivity`, create a member variable for the text view and initialize it in `onCreate()`.
2. In `onClickDisplayEntries()`, use a switch statement to check which button was pressed. Use the view id to distinguish the buttons. Print a log statement for each case.
3. In `onClickDisplayEntries()`, at the end append some text to the `textview`.
4. As always, extract the string resources and run the app.

Your `MainActivity` should be similar to this solution.

**Solution:**

```
package android.example.com.minimalistcontentprovider;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    private static final String TAG = MainActivity.class.getSimpleName();

    TextView mTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mTextView = (TextView) findViewById(R.id.textview);
    }

    public void onClickDisplayEntries(View view) {
        Log.d(TAG, "Yay, I was clicked!");

        switch (view.getId()) {
            case R.id.button_display_all:
                Log.d(TAG, "Yay, " + R.id.button_display_all + " was clicked!");
                break;
            case R.id.button_display_first:
                Log.d(TAG, "Yay, " + R.id.button_display_first + " was clicked!");
                break;
            default:
                Log.d(TAG, "Error. This should never happen.");
        }
        mTextView.append("Thus we go! \n");
    }
}
```

## Task 2. Create a Contract class, a URI scheme, and mock data

The contract contains information about the data that apps need to build queries, in particular, the names of the selectors.

### Why:

- Contract is public and includes important information for other apps that want to connect to this content provider.

- The URI scheme shows how to build URIs to access the data. It's the API for the data.
- Separates design/definition from the implementation.
- Allows to define shared constants, making it easier to maintain the application.
- Makes information easy to find, because it is in one place.
- The contract contains information about the data that apps need to build queries, in particular, the names of the selectors.

## 2.1. Create the Contract class

1. Create a new public **Java class Contract** with the following signature. Make sure to make it final.

```
public final class Contract {}
```

2. To prevent someone from accidentally instantiating the Contract class, give it an empty private constructor. This is a standard pattern.

```
private Contract() {}
```

## 2.2. Create the URI scheme

Other apps use content URIs to query the content provider. See the chapter introduction for a quick summary on URIs.

The URI scheme for the content provider is defined in the Contract so that it is available to any app that wants to query this content provider. Customarily, this is done by defining constants for AUTHORITY, CONTENT\_PATH, and CONTENT\_URI.

1. In the Contract class, create a constant for AUTHORITY. Customarily, to make Authority unique, it's the package name extended with "provider."

```
public static final String AUTHORITY = "com.android.example.minimalistcontentprovider.provider";
```

2. Create a constant for the CONTENT\_PATH. The content path is an abstract semantic identifier of the data you are interested in. It does not predict or presume in what form the data is stored or organized in the background. As such, "words" could resolve in the name of a table, the name of a file, or in this example, the name of the list.

```
public static final String CONTENT_PATH = "words";
```

3. Create a constant for the CONTENT\_URI. This is a content:// style URI that points to one set of data. If you have multiple "data containers" in the backend, you would create

a content URI for each. [Uri](#) is a helper class for building and manipulating URIs.

```
public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + C  
ONTENT_PATH);
```

4. Create a convenience constant for ALL\_ITEMS. This means you don't have to reveal the implementation details, and you can change it without breaking your clients' apps. (The value is -2 because that's the first lowest value not returned by a method call.)

```
static final int ALL_ITEMS = -2;
```

5. Create a convenience constant for WORD\_ID.

```
static final String WORD_ID = "id";
```

## 2.3. Add the MIME Type

Content providers require the `getType()` method to be implemented, returning a [MIME type](#) for one item and multiple items. The MIME type identifies the content type for this content provider. You can use the MIME type information to find out if your application can handle data that the provider offers, and choose how you want to process received data based on the MIME type. You usually need the MIME type when you are working with a provider that contains complex data structures or files.

MIME types are of the form `type/subtype`, such as `text/html` for HTML pages. For your content provider, you need to define a vendor-specific MIME type for the kind of data your content provider returns. The type of vendor-specific Android MIME types is always:

- `vnd.android.cursor.item` for one data item/record
- `vnd.android.cursor.dir` for a set of multiple records.

The subtype can be anything, but it is a good practice to make it informative to what it is a MIME type for. For example:

- vnd—a vendor MIME type
- com.example—the domain
- provider—it's for a content provider
- words—the name of the table

Read [Implementing ContentProvider MIME types](#) for details.

For this practical, you can return one of two MIME types as follows:

1. Declare the MIME time for one data item.

```
static final String SINGLE_RECORD_MIME_TYPE = "vnd.android.cursor.item/vnd.com.example.provider.words";
```

2. Declare the MIME type for multiple records.

```
static final String MULTIPLE_RECORD_MIME_TYPE = "vnd.android.cursor.dir/vnd.com.example.provider.words";
```

## 2.5. Create the mock data

The content provider always presents the results as a [Cursor](#) (see practical on storing data) in a table format that resembles a SQL database. This is independent of how the data is actually stored. To keep it simple for this example and focus your efforts on the content provider itself, this app uses a string array of words.

In strings.xml, add a short list of words:

```
<string-array name="words">
    <item>Android</item>
    <item>Activity</item>
    <item>ContentProvider</item>
    <item>ContentResolver</item>
</string-array>
```

## Task 3. Implement the MiniContentProvider class

### 3.1. Create the MiniContentProvider class

1. Create a class MiniContentProvider extending ContentProvider.
2. Implement the methods ([Code > Implement methods](#)).
3. Add a log tag.
4. Add a member variable for the mock data.

```
public String[] mData;
```

5. In onCreate(), create initialize `mData` from the array of words, and return true.

```

@Override
public boolean onCreate() {
    Context context = getContext();
    mData = context.getResources().getStringArray(R.array.words);
    return true;
}

```

6. Add an appropriate logging message to the insert, delete, and update methods. You will not implement these methods for this practical.

```
Log.e(TAG, "Not implemented: update uri: " + uri.toString());
```

## 3.2. Publish the content provider by adding it to the Android manifest

In order to access the content provider, your app and other apps need to know that it exists. Add a declaration for the content provider to the Android manifest inside a <provider> tag. Use the existing one, if there is already one. The declaration contains the name of the content provider and the authorities (its unique identifier).

1. In the AndroidManifest, inside the application tag, after the activity closing tag, add:

```

<provider
    android:name=".MiniContentProvider"
    android:authorities="android.example.com.minimalistcontentprovider.provider" />

```

2. Run your code to make sure it compiles.

## 3.3 Set up URI matching

In order to take appropriate action depending on the request URI, the content provider needs to analyze the URI. [UriMatcher](#) is a helper class for processing the accepted Uri schemes for a given content provider.

- You create an instance of UriMatcher.
- Add each URI that your content provider recognizes to the UriMatcher.
- Assign each URI a numeric constant. Having a numeric constant for each URI is convenient when you are processing incoming URIs because you can use a switch/case statement on the numeric values to work through the URIs.

Make the following changes in the MiniContentProvider class.

1. In the MiniContentProvider class, create a private static variable for a new UriMatcher.

The argument specifies the value to return if there is no match. As a best practice, use UriMatcher.NO\_MATCH.

```
private static UriMatcher sUriMatcher =
    new UriMatcher(UriMatcher.NO_MATCH);
```

2. Create a method for initializing the URI matcher.

```
private void initializeUriMatching(){}
```

3. In the initializeUriMatching() method, add the URIs that your content provider accepts to the matcher and assign them an integer code. These are the URIs based on the authority and content paths specified in the contract.

The # symbol matches a string of numeric characters of any length. In this app, it refers to the index of the word in the string array. In a production app, this could be the id of an entry in a database. Assign this URI a numeric value of 1.

```
sUriMatcher.addURI(Contract.AUTHORITY, Contract.CONTENT_PATH + "/#", 1);
```

4. The second URI is the one you specified in the contract for returning all items. Assign it a numeric value of 0.

```
sUriMatcher.addURI(Contract.AUTHORITY, Contract.CONTENT_PATH, 0);
```

Note that if your app is more complex and uses more URIs, use constants for the codes, as shown in the [UriMatcher documentation](#).

### Solution:

```
private void initializeUriMatching(){
    sUriMatcher.addURI(Contract.AUTHORITY, Contract.CONTENT_PATH + "/#", 1);
    sUriMatcher.addURI(Contract.AUTHORITY, Contract.CONTENT_PATH, 0);
}
```

## 3.4 Implement the getType() method

The getType() method returns the MIME type for each of the specified URIs.

Unless you are doing something special in your code, this method implementation is going to be very similar for any content provider and does the following:

1. Match the URI.

2. Switch on the returned code.
3. Return the appropriate MIME type.

Learn more in the [UriMatcher documentation](#).

### Solution:

```
public String getType(Uri uri) {  
    switch (sUriMatcher.match(uri)) {  
        case 0:  
            return Contract.MULTIPLE_RECORD_MIME_TYPE;  
        case 1:  
            return Contract.SINGLE_RECORD_MIME_TYPE;  
        default:  
            // Alternatively, throw an exception.  
            return null;  
    }  
}
```

## 3.5 Implement the query() method

The purpose of the query method is to match the URI, convert it to a query, execute the query, and return the result in a [Cursor](#) object. For example, commonly this could be an SQL query sent to an SQLite database.

### The query() method

Make sure you completely understand this method and its arguments.

**WHY:** Because if you understand this method, implementing content providers is considerably more straightforward.

The query method has the following signature:

```
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionA  
rgs, String sortOrder){}
```

The arguments to this method represent the parts of an SQL query. Even if you are using another kind of backend, you must still accept a query in this style and handle the arguments appropriately. (In the next task you will build a query in the `MainActivity` to see how the arguments are used.)

uri	The complete URI queried. This cannot be null.
projection	Indicates which columns/attributes you want to access
selection	Indicates which rows/records of the objects you want to access
selectionArgs	The binding parameters to the previous selection argument. For security reasons, the arguments are processed separately.
sortOrder	Whether to sort, and if so, whether ascending or descending. If this is null, the default sort or no sort is applied.
@return	Cursor of any kind, with the response data inside.

1. Identify the following processing steps in the query() method code shown below in the solutions section.

Query processing always consists of these steps:

2. Match the URI.
3. Switch on the returned code.
4. Process the arguments and build a query appropriate for the backend.
5. Get the data and (if necessary) drop it into a Cursor.
6. Return the cursor.
7. Identify portions of the code that need to be different in a real-world application.

The query implementation for this basic app takes some shortcuts.

- Error handling is minimal.
- Because the app is using mock data, the cursor can be directly populated.
- Because the URI scheme is simple, this method is rather short.
- Identify at least one design decision that makes it easier to understand and maintain the code.
- Analyzing the query and executing it to populate a cursor are separated into two methods.
- The code contains more comments than executable code.
- Add the code to your app. You will get an error for the populateCursor() method, and will address this in the next step.

### Annotated Solution Code for the query() method in MiniContentProvider.java

```
@Nullable
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[] selection
Args, String sortOrder) {
    int id = -1;
    switch (sUriMatcher.match(uri)) {
        case 0:
            // Matches URI to get all of the entries.
            id = Contract.ALL_ITEMS;
            // Look at the remaining arguments to see whether there are constraints.
            // In this example, we only support getting a specific entry by id. Not ful
l search.
            // For a real-life app, you need error-catching code; here we assume that t
he
            // value we need is actually in selectionArgs and valid.
            if (selection != null){
                id = parseInt(selectionArgs[0]);
            }
            break;

        case 1:
            // The URI ends in a numeric value, which represents an id.
            // Parse the URI to extract the value of the last, numeric part of the path
            ,
            // and set the id to that value.
            id = parseInt(uri.getLastPathSegment());
            // With a database, you would then use this value and the path to build a q
uery.
            break;

        case UriMatcher.NO_MATCH:
            // You should do some error handling here.
            Log.d(TAG, "NO MATCH FOR THIS URI IN SCHEME.");
            id = -1;
            break;
        default:
            // You should do some error handling here.
            Log.d(TAG, "INVALID URI - URI NOT RECOGNIZED.");
            id = -1;
    }
    Log.d(TAG, "query: " + id);
    return populateCursor(id);
}
```

## 3.6 Implement the `populateCursor()` method

Once the `query()` method has identified the URI, it calls the `populateCursor()` with the last segment of the path, which is the id (index) of the word to retrieve.

The query method must return a Cursor, so populateCursor has to create, populate, and return a cursor.

- If your data is stored in a SQLite database, executing the query will return a Cursor.
- If you are not using a data storage method that returns a cursor, such as files or the mock data, you can use a simple [MatrixCursor](#) to hold the data to return.

The populateCursor() method connects the content provider to the data source. It does the following:

1. Receives the id extracted from the URI.
2. Creates a cursor to store received data (because the mock data received is not a cursor). For this sample, this gets the string at the index `id` from the string array. In a more realistic app, this could execute a query to a database.
3. Executes the query.
4. Adds the result to the cursor.
5. Returns the cursor.

```
private Cursor populateCursor(int id) {
    MatrixCursor cursor = new MatrixCursor(new String[] { Contract.CONTENT_PATH });
    // If there is a valid query, execute it and add the result to the cursor.
    if (id == Contract.ALL_ITEMS) {
        for (int i = 0; i < Contract.mWordList.size(); i++) {
            String word = Contract.mWordList.get(i);
            cursor.addRow(new Object[]{word});
        }
    } else if (id >= 0) {
        // Execute the query to get the requested word.
        String word = Contract.mWordList.get(id);
        // Add the result to the cursor.
        cursor.addRow(new Object[]{word});
    }
    return cursor;
}
```

## Task 4. Use MiniContentProvider to get data

With the content provider in place, the onClickDisplayEntries() method in the MainActivity can be expanded to query and display data to the UI. This requires the following steps:

1. Create the SQL-style query, depending on which button was pressed.
2. Use a content resolver to interact with the content provider to execute the query and return a Cursor.
3. Process the results in the Cursor.

Let's look at each of these.

## 4.1. Get the content resolver

The content resolver parses the query and all arguments and interacts with the content provider on your behalf. If you provide a well-formed query, the results should always be civilized. The magic behind this is explained in the next practical. You don't need to create your own content resolver. You can use the one provided by the framework.

1. Remove all code from inside onClickDisplayEntries.
2. Add this code to onClickDisplayEntries in MainActivity.

```
Cursor cursor = getContentResolver().query(Uri.parse(queryUri), projection, selectionClause, selectionArgs, sortOrder);
```

The query part of this call takes arguments URI, projection, selectionClause, selectionArgs, and sortOrder, which should be familiar from the query method in the MiniContentProvider class. You must define those arguments next.

## 4.3. Declare the parts of the query

One of the advantages of content providers is that you do not need to know how the backend is implemented. You always build a SQL-like query in one format that fits all. The Contract for each content provider documents the types of queries you can make for a given content provider.

In order for this call to work, you need to declare and assign values to all the arguments.

1. **URI:** Declare the URI that identifies the content provider and the table. Note: Remember that from the perspective of the app, there is always a table at the backend. You get the information for the correct URI from the contract.

```
String queryUri = Contract.CONTENT_URI.toString();
```

2. **Projection:** A string array with the names of the columns to return for each row. Setting this to null returns all columns. When there is only one column, as in the case of this example, setting this explicitly is optional, but can be helpful for documentation purposes.

```
String[] projection = new String[] {Contract.CONTENT_PATH}; // Only get words.
```

3. **selectionClause:** Argument clause for the selection criteria for which rows to return. Formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null returns all rows for the given URI. Since this will vary depending on which button was pressed, declare it now and set this later.

```
String selectionClause;
```

4. **selectionArgs:** Argument values for the selection criteria. If you include ?s in selection, they are replaced by values from selectionArgs, in the order that they appear.

IMPORTANT: It is a best security practice to always separate selection and selectionArgs.

```
String selectionArgs[];
```

5. **sortOrder:** The order in which to sort the results. Formatted as an SQL ORDER BY clause (excluding the ORDER BY keyword). Usually ASC or DESC; null requests the default sort order, which could be unordered.

```
String sortOrder = null; // For this example, accept the order returned by the response.
```

## 4.2. Decide on selection criteria

The selectionClause and selectionArgs values depend on which button was pressed.

- To display all the words, set both arguments to null.
- To get the first word, query for the word with the ID of 0. (This assumes that word IDs start at 0 and are created in order. You know this, because the information is exposed in the contract. For a different content provider, you may not know the ids, and may have to search in a different way.)
- Replace the existing switch block with the following code in onClickDisplayEntries, before you get the content resolver.

```
switch (view.getId()) {  
    case R.id.button_display_all:  
        selectionClause = null;  
        selectionArgs = null;  
        break;  
    case R.id.button_display_first:  
        selectionClause = Contract.WORD_ID + " = ?";  
        selectionArgs = new String[] {"0"};  
        break;  
    default:  
        selectionClause = null;  
        selectionArgs = null;  
}
```

## 4.3. Process the Cursor

After getting the content resolver, you have to process the result from the Cursor.

- If there is data, display it in the text view.
- If there is no data, report errors.
- Examine the code. If you have difficulty understanding this code:
- Revisit the chapter on storing data.
- Read the documentation on [Cursors](#).
- Identify what specifically you don't understand and find answer on [developers.android.com](#) and developer community sites.

```
if (cursor != null) {  
    if (cursor.getCount() > 0) {  
        cursor.moveToFirst();  
        int columnIndex = cursor.getColumnIndex(projection[0]);  
        do {  
            String word = cursor.getString(columnIndex);  
            mTextView.append(word + "\n");  
        } while (cursor.moveToNext());  
    } else {  
        Log.d(TAG, "onClickDisplayEntries " + "No data returned.");  
        mTextView.append("No data returned." + "\n");  
    }  
} else {  
    Log.d(TAG, "onClickDisplayEntries " + "Cursor is null.");  
    mTextView.append("Cursor is null." + "\n");  
}  
cursor.close();
```

- Insert this code at the end of `onClickDisplayEntry()`.
- Run your app.
- Click the buttons to see the retrieved data in the text view.

## 4.4. Feel proud

If you understand everything in this practical, and have a running app, feel proud, because you have grasped one of the more advanced and complex design patterns of the Android framework.

In the next practical, you will add a content provider to the WordListSQL app. You can use the code you created here as a reference and template.

## Coding challenges

### Implement missing methods

Implement the insert, delete, and update methods for the MinimalistContentProvider app. Provide the user with a way to insert, delete, and update data.

**Hint:** If you don't want to build out the user interface, create a button for each action and hardwire the data that is inserted, updated, and deleted. The point of this exercise is to work on the content provider, not the user interface.

**Why:** You will implement the fully functioning content provider with UI in the next practical, when you will add a content provider to the WordListSQL app.

## Add Unit Tests for the content provider

After you implemented the content provider, there was no way for you to know whether or not the code would work. In this sample, you built out the front-end and by watching it work, assumed the app worked correctly. In a real-life app, this is not sufficient, and you may not even have access to a front-end. The appropriate way for determining that each method acts as expected, write a set of unit tests for MiniContentProvider.

## Summary

In this chapter, you learned how to

- Create a Contract to expose your content provider's API to other app. Note that contracts can be useful beyond content providers, for example, for databases. See coding challenge in the data storing chapter.
- Define a URI scheme so that other apps can access data through your content provider.
- Implement a minimalist content provider to help you understand how content providers work.
- Use a content resolver to request data from a content provider and display it to the user.

## Resources

### Developer Documentation:

- [Uniform Resource Identifiers or URIs](#)
- [MIME type](#)
- [MatrixCursor and Cursors](#)
- [Content Providers](#)



# 12.2 P: Implementing a Content Provider for WordListSQL

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 0. Introducing App Architecture
- Task 1. Downloading and run the base code
- Task 2: Adding a Contract class to WordListSQLInteractive
- Task 3: Creating a Content Provider
- Task 4: Implementing Content Provider methods
- Coding Challenge
- Summary
- Resources

Content providers in real apps are more complex than the bare-bones version you built in the previous practical.

In the real world:

- The backend is a database, file system, or other persistent storage option.
- The front-end displays the data in a pleasing UI and allows users to manipulate the data.

Your job will rarely be to build an app from scratch. More often, you will be asked to debug, refactor, or extend an existing application.

In this practical, you will take the WordListSQL app and refactor and extend it to use a content provider as a layer between the SQL database and the RecyclerView.

This is not an easy task, and you can go about it in two ways.

- Refactor and extend the WordListSQL app. This involves changing the app architecture and refactoring code.
- Start from scratch and re-use code from WordListSQL and MinimalistContentProvider.

The practical will demonstrate how to refactor the existing WordListSQL app, because it's what you are more likely to encounter on the job..

# What you should already KNOW

For this practical you should be familiar with:

- Creating, building and running apps in Android Studio.
- Displaying data in a RecyclerView
- Using adapters as intermediaries between data and views.
- Adding onClick event handlers to views. Creating onClick handlers dynamically.
- Abstracting and encapsulating data with data models.
- Starting and returning from a second activity.
- Passing data between activities using Extras.
- Using a EditText view to get data from the user.
- Creating, managing, and interacting with a SQLite database using an SQLiteOpenHelper.
- Understanding the architecture of a simple content provider, such as minimalist content provider you built in the previous practical.
- Applying abstraction to application design.
- Researching topics in the Android documentation and developer communities.
- Working with [CRUD](#) interfaces.

# What you will LEARN

- How to create a fully developed content provider for an existing application.
- How to refactor an application to accommodate a content provider.
- How to handle cascading changes and debug cascading errors.

# What you will DO

This practical is more advanced and requires setup that is more typical for real-word app development.

You start with the WordListSQLInteractive app you created in a previous practical, which displays words from a SQLite database in a RecyclerView, and users can create, edit, and delete words.

You will extend and modify this app:

- Implement a Contract class to expose your app's interface to other apps.
- Implement a ContentProvider.
- Refactor the MainActivity, WordListAdapter, and WordListOpenHelper classes to work

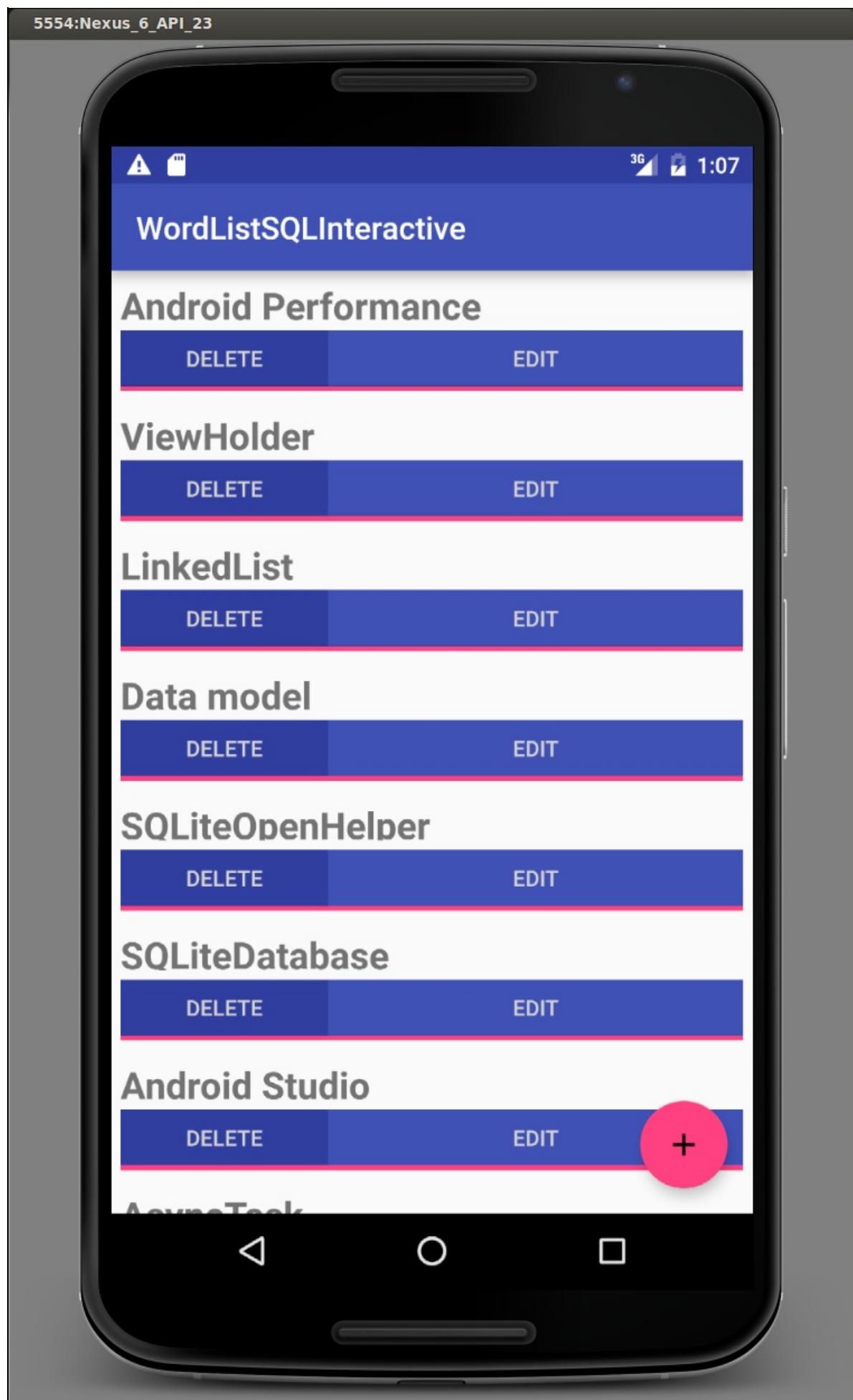
with the content provider.

## App Overview

The completed WordListSQLWithContentProvider app will have the following features:

- Separated backend and frontend by using a content provider and content resolver.
- Unchanged user interface and functionality.
- Public API through the content provider as specified in a Contract.

Your app will look that same as at the end of the data storage practical.



# Task 0. App Architecture

When apps get more complex it helps to have a plan, and for that plan to follow established practices. And to draw it out in diagrams.

- Keeping it all in your head gets really difficult really fast. Even with a simple app as WordListSQL, if you are new to app development, not having a plan is going to result in inconsistencies, which inevitably leads to more mistakes, which are harder to debug.
- Understanding, and thus changing, an existing app is much easier if you know what its architecture is.
- If you have a plan, and it is written, you can communicate it to your co-workers, your investors, and especially, to your technical writers.
- Having a roadmap makes development easier, because you make fewer decisions on the fly.

The roadmap for an application has at least these parts.

- The software architecture of the app. What the different pieces are, and how they relate to each other.
- The API. The public classes, methods, and functions that other applications can use, along with their signatures.
- The user interface, and how the user is guided through the workflow.

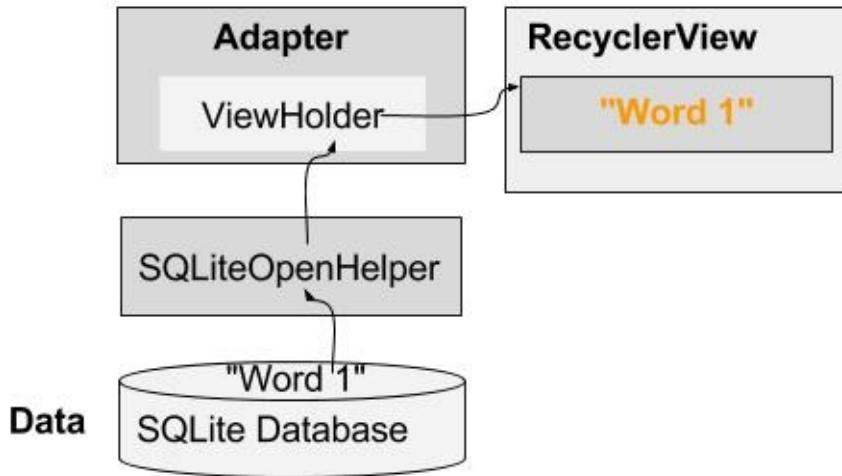
In this task, you are taking a closer look at the software architecture and API.

## 0.1. WordListSQLInteractive architecture and API diagrams

1. Open WordListSQLInteractive in Android Studio. You can download the finished app [HERE](#).
2. Inspect the architecture and API diagrams and relate them to your code. In particular, take note how the APIs are CRUD-like and match up through the stack.

### Architecture

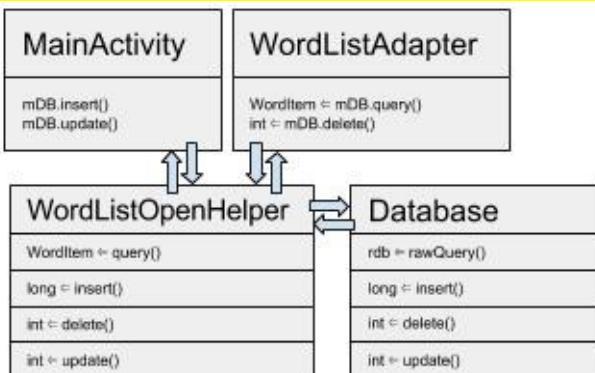
This is an architecture diagram for WordListSQL that you built in the previous chapter.



## API

The following diagram shows the relationship between **MainActivity**, **WordListAdapter**, **WordListOpenHelper**, and the **Database** through their APIs. Note that only the CRUD methods are listed. Which other methods could be included in this diagram?

[TODO]: Resize Original image



**Solution:** `count()`

## 0.2. Architecture and API diagrams for WordListSQLWithContentProvider

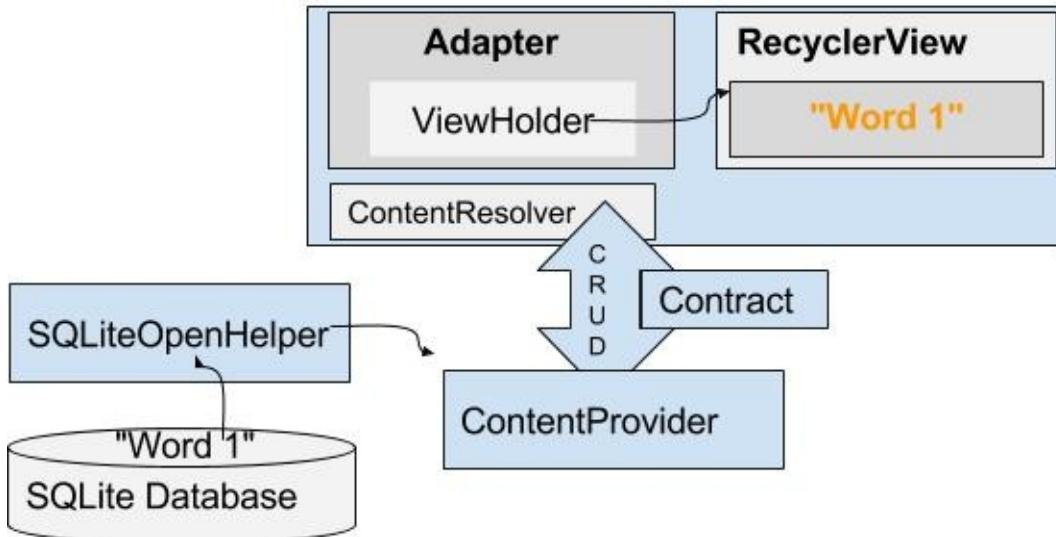
Following are the architecture diagrams for the finished app with a content provider.

1. Inspect the architecture diagram.
2. Write down at least three things about the app you are going to build.
  - What additional classes you might need?
  - Which classes you might need to change?
  - What you might want look up in the documentation and textbook before you start.
3. Inspect the API diagram.
4. Write down at least three things about the API you are going to build.

- Does anything in particular stand out when you look at the diagram?

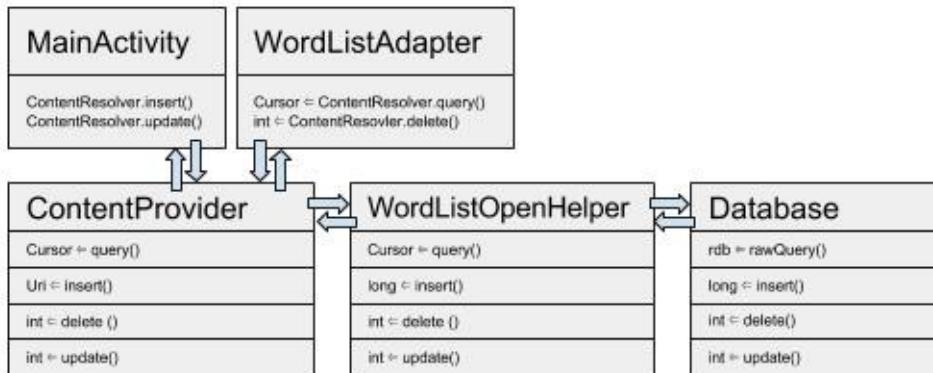
## Architecture with content provider

The blue boxes indicate components you will change.



## APIs with content provider

TODO: Resize Original image



## Solution:

Some of the things you might have noticed:

- You will not need to change the ViewHolder class, because it is wrapped by an adapter.
- All the functions follow the CRUD architecture, and they have the same signature through the application stack.
- Designing your base app well, and with expansion in mind, using an established API pattern, is going to make it easier to "insert" a content provider.
- New Classes: Contract, ContentProvider
- Classes that change: WordListOpenHelper, MainActivity, WordListAdapter
- Classes that should not change: WordItem, MyButtonOnClickListener, ViewHolder

# Task 1. Download and run the base code

This practical builds on the WordListSQLInteractive and MinimalistContentProvider apps that you built previously. You can start from your own code, or download the base apps.

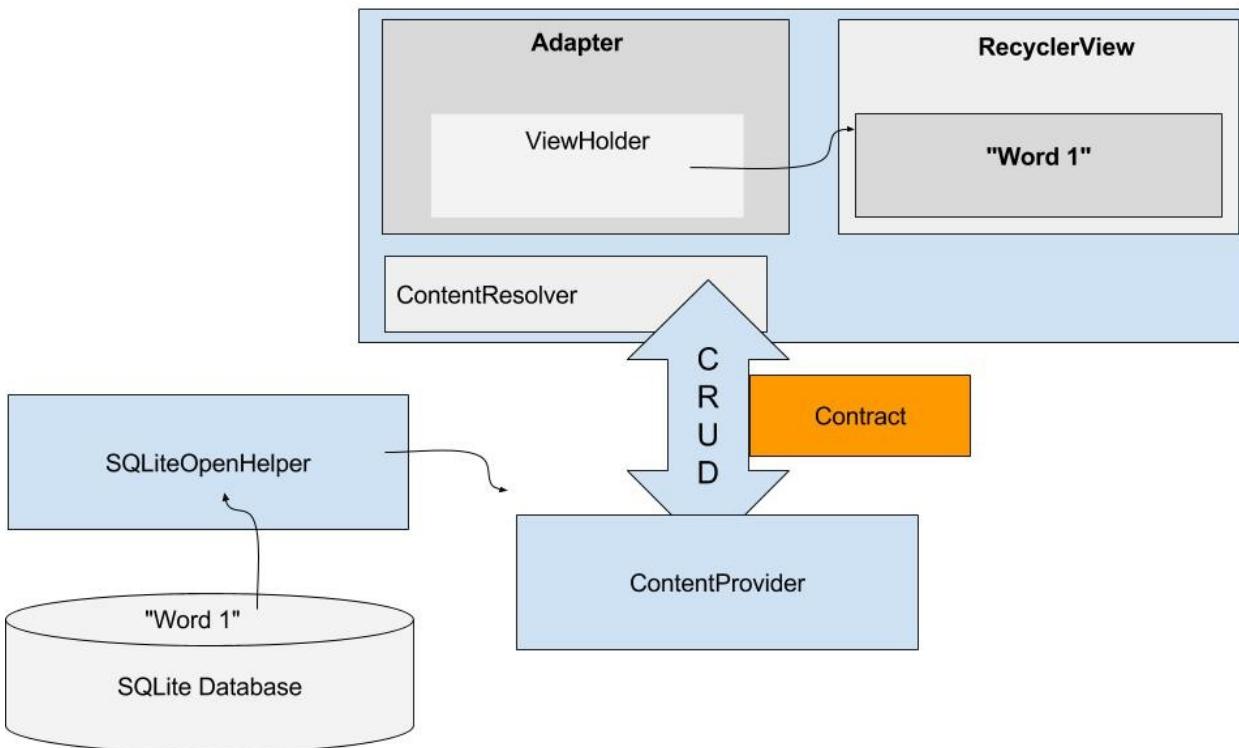
- WordListSQLInteractive
- MinimalistContentProvider

Load both apps into Android Studio, and run them.

You will extend WordListSQLInteractive and refer to MinimalistContentProvider.

# Task 2. Add a Contract class to WordListSQLInteractive

You will start by creating a contract class that defines public database constants, URI constants, and the MIME types. You will use these constants in all the other classes.



## 2.1 Add a Contract class

1. Study the [Define a Schema and Contract documentation](#).
2. Add a new public final class to your project and call it Contract.

This contract contains all the information that any app needs to use your app's content provider.

```
public final class Contract {}
```

3. To prevent the Contract class from being instantiated, add a private, empty constructor.

This is a standard pattern for classes that are used to hold meta information and constants for an app.

```
private Contract() {}
```

## 2.2 Move database constants into Contract

Move the constants for the database that another app would need to know out of WordListOpenHelper into the contract and make them public.

1. Move DATABASE\_NAME.

```
public static final String DATABASE_NAME = "wordlist";
```

A common way of organizing a contract class is to put definitions that are global to your database into the root level of the class. Then, create a static abstract inner class for each table with the column names. This inner class commonly implements the [BaseColumns](#) interface. By implementing the BaseColumns interface, your class can inherit a primary key field called \_ID that some Android classes, such as cursor adapters, expect to exist. This is not required, but can help your database work harmoniously with the Android framework.

2. Create an inner class WordList that implements BaseColumns.

```
public static abstract class WordList implements BaseColumns {  
}
```

3. Move WORD\_LIST\_TABLE, KEI\_ID, and KEY\_WORD from WordListOpenHelper into the WordList class in Contract, and make them public.
4. Go back to WordListOpenHelper and wait for Android Studio to import the constants from the Contract; or import them manually, if you are not set up for auto-imports.

## 2.3 Define URI Constants

1. Declare the URI scheme for your content provider.

Using the Contract in MinimalistContentProvider as an example, declare AUTHORITY, CONTENT\_PATH. Add CONTENT\_PATH\_URI to return all items, and ROW\_COUNT\_URI that returns the number of entries. In the AUTHORITY, use your app's name.

```
public static final int ALL_ITEMS = -2;
public static final String COUNT = "count";

public static final String AUTHORITY =
    "com.android.example.wordlistsqllwithcontentprovider.provider";

public static final String CONTENT_PATH = "words";

public static final Uri CONTENT_URI =
    Uri.parse("content://" + AUTHORITY + "/" + CONTENT_PATH);
public static final Uri ROW_COUNT_URI =
    Uri.parse("content://" + AUTHORITY + "/" + CONTENT_PATH + "/" + COUNT);
```

## 2.4 Declare the MIME types

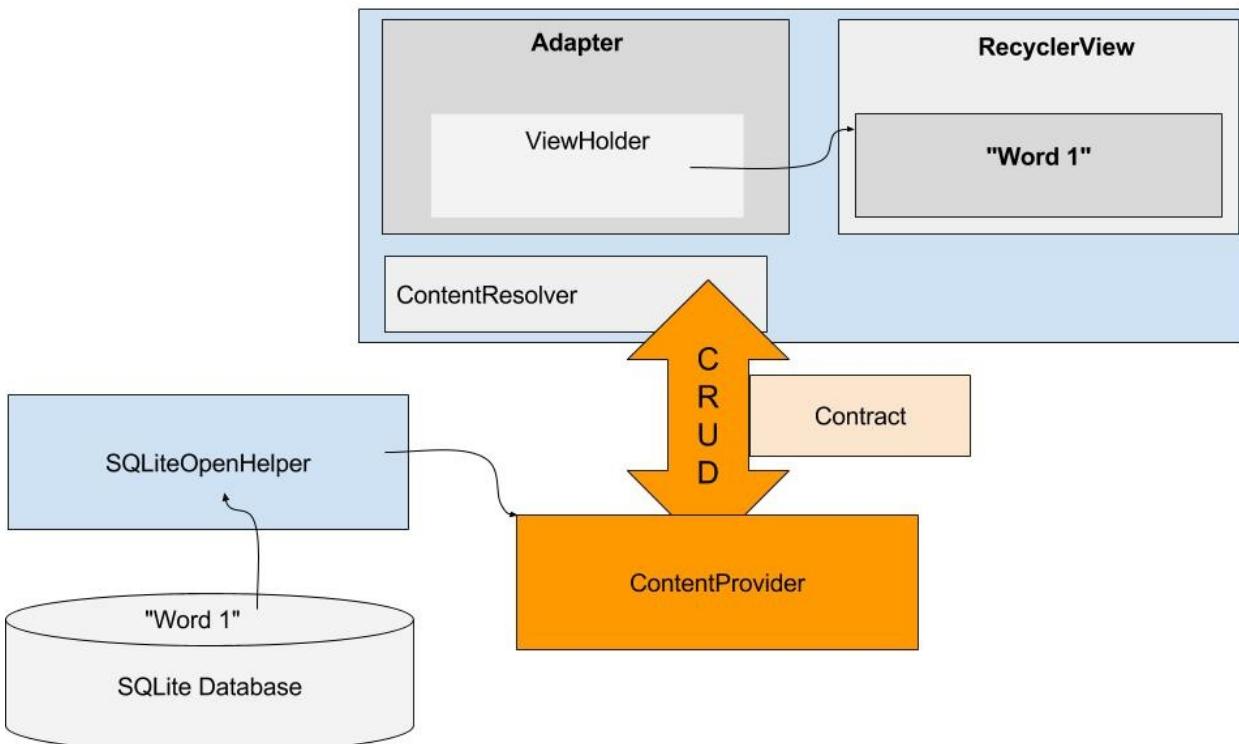
The [MIME type](#) tells an app, what type and format received data is in, so that it can process the data appropriately. Common MIME types include `text/html` for web pages, and `application/json`. Read more about [MIME types for content providers](#) in the Android documentation.

1. Declare MIME types for single and multiple record responses:

```
static final String SINGLE_RECORD_MIME_TYPE =
    "vnd.android.cursor.item/vnd.com.example.provider.words";
static final String MULTIPLE_RECORDS_MIME_TYPE =
    "vnd.android.cursor.item/vnd.com.example.provider.words\"";
```

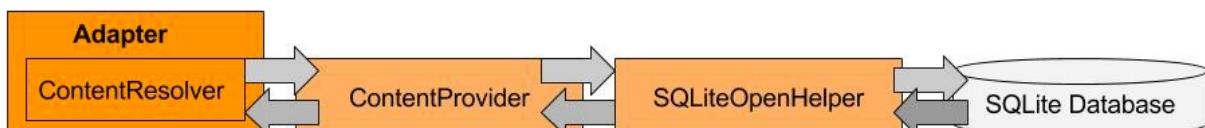
2. Run your app. It should run and look and act exactly as before you changed it.

## Task 3. Create a Content Provider



In this task you will create a content provider, implement its query method, and hook it up with the WordListAdapter and the WordListOpenHelper. Instead of querying the WordListOpenHelper, the Word List Adapter will use a content resolver to query the content provider, which in turn will query WordListOpenHelper which will query the database.

#### Life of a Query



## 3.1 Create a WordListContentProvider class

1. Create a new class that extends **ContentProvider** and call it **WordListContentProvider**.
2. In Android Studio, click on the red lightbulb, select "Implement methods", and click **OK** to implement all listed methods.
3. Specify a log TAG.
4. Declare a UriMatcher.

This content provider uses an [UriMatcher](#), a utility class that maps URIs to numbers, so you can switch on them.

```
private static UriMatcher sUriMatcher =
    new UriMatcher(UriMatcher.NO_MATCH);
```

1. Declare a WordListOpenHelper class variable, mDB.

```
private WordListOpenHelper mDB.
```

2. Declare the codes for the URI matcher as constants.

This puts the codes in one place and makes them easy to change. Use tens, so that inserting additional codes is straightforward.

```
private static final int URI_ALL_ITEMS_CODE = 10;
private static final int URI_ONE_ITEM_CODE = 20;
private static final int URI_COUNT_CODE = 30;
```

3. Change the onCreate() method to

- initialize mDB with a WordListOpenHelper,
- call the initializeUriMatching() method that you will create next,
- and return true.

```
@Override
public boolean onCreate() {
    mDB = new WordListOpenHelper(getContext());
    initializeUriMatching();
    return true;
}
```

4. Create a private void method initializeUriMatching().

5. In initializeUriMatching(), add URLs to the matcher for getting all items, one item, and the count.

Refer to the Contract and use the initializeUriMatching() method in the MinimalistContentProver app as a template.

**Solution:**

```
private void initializeUriMatching(){
    sUriMatcher.addURI(Contract.AUTHORITY, Contract.CONTENT_PATH, URI_ALL_ITEMS_CODE);
    sUriMatcher.addURI(Contract.AUTHORITY, Contract.CONTENT_PATH + "/#", URI_ONE_ITEM_CODE);
    sUriMatcher.addURI(Contract.AUTHORITY, Contract.CONTENT_PATH + "/" + Contract.COUNT,
        URI_COUNT_CODE );
}
```

## 3.2 Implement WordListContentProvider.query()

Use the MiniContentProvider as a template to implement the query() method.

1. Modify WordListContentProvider.query().

2. Switch on the codes returned by sUriMatcher.
3. For URI\_ALL\_ITEMS\_CODE, URI\_ONE\_ITEM\_CODE, URI\_COUNT\_CODE, call the corresponding in WordListOpenHelper (mDB).

**Notice** how assigning the results from mDB.query() to a cursor, generates an error, because WordListOpenHelper.query() returns a WordItem.

**Notice** how assigning the results from mDB.count() to a cursor generates an error, because WordListOpenHelper.count() returns a long.

You will fix both these errors next.

### Solution:

```
@Nullable
@Override
public Cursor query(Uri uri, String[] projection, String selection,
                     String[] selectionArgs, String sortOrder) {

    Cursor cursor = null;

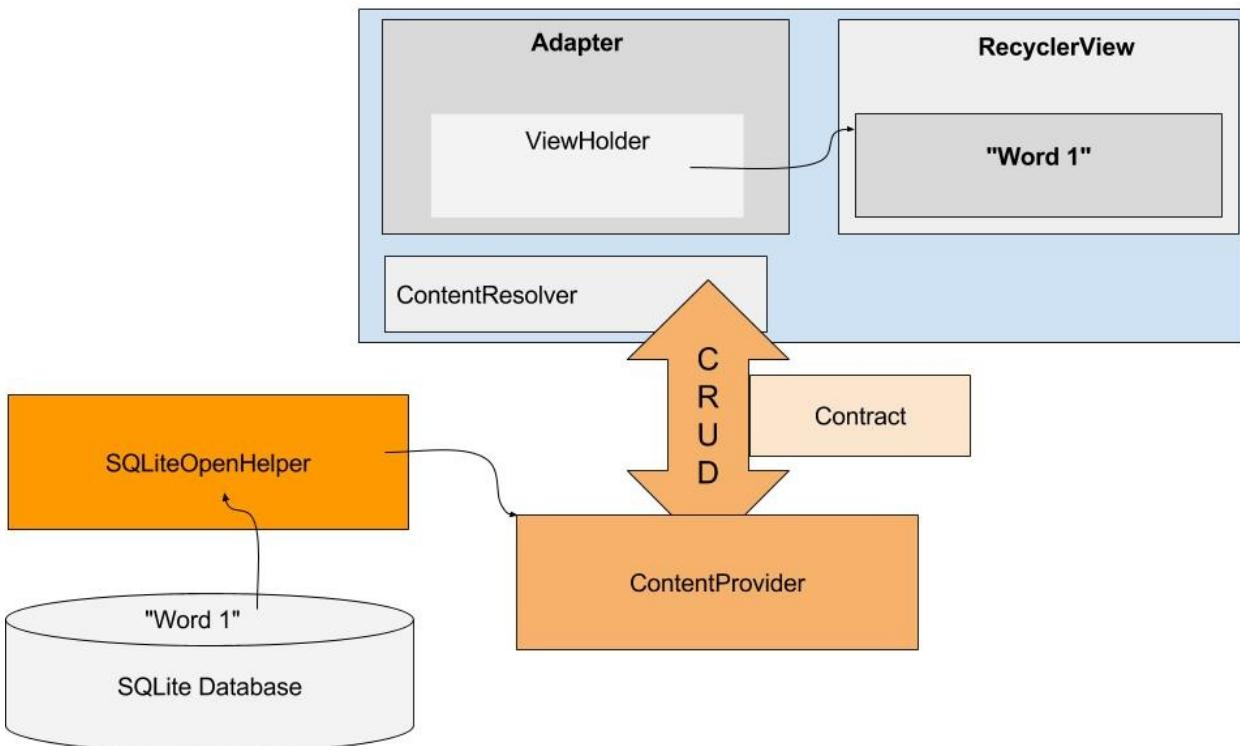
    switch (sUriMatcher.match(uri)) {
        case URI_ALL_ITEMS_CODE:
            cursor = mDB.query(ALL_ITEMS);
            break;

        case URI_ONE_ITEM_CODE:
            cursor = mDB.query(parseInt(uri.getLastPathSegment()));
            break;

        case URI_COUNT_CODE:
            cursor = mDB.count();
            break;

        case UriMatcher.NO_MATCH:
            // You should do some error handling here.
            Log.d(TAG, "NO MATCH FOR THIS URI IN SCHEME.");
            break;
        default:
            // You should do some error handling here.
            Log.d(TAG, "INVALID URI - URI NOT RECOGNIZED.");
    }
    return cursor;
}
```

## 3.3 Fix WordListOpenHelper.query() to return a Cursor



Since the content provider works with cursors, you can simplify the `WordListOpenHelper.query()` method to return a cursor.

Go ahead and do that.

This fixes the error in `WordListContentProvider.query()`.

However, this breaks `WordListAdapter.OnBindViewHolder()`, which expects a `WordItem` from `WordListOpenHelper`.

To resolve this, `WordListAdapter.onBindViewHolder()` needs to use a content resolver instead of calling the database directly, which you will do after fixing `WordListContentProvider.count()`.

**NOTE:** This kind of cascading errors and fixes is typical for working with real-life applications. If an app you are working with is well architected, you can follow the bread crumbs and fix the errors one by one.

## 3.4 Fix `WordListOpenHelper.count()` to return a Cursor

Since the content provider works with cursors, you must also change the `WordListOpenHelper.count()` method to return a cursor.

Use a [MatrixCursor](#), which is a cursor of changeable rows and columns.

1. Create a `MatrixCursor` using `Contract.CONTENT_PATH`.
2. Inside a try block, get the count and add it as a row to the cursor.
3. Return the cursor.

**Solution:**

```
public Cursor count(){
    MatrixCursor cursor = new MatrixCursor(new String[] {Contract.CONTENT_PATH});
    try {
        if (mReadableDB == null) {
            mReadableDB = getReadableDatabase();
        }
        int count = (int) DatabaseUtils.queryNumEntries(mReadableDB, WORD_LIST_TABLE);
        cursor.addRow(new Object[]{count});
    } catch (Exception e) {
        Log.d(TAG, "EXCEPTION " + e);
    }
    return cursor;
}
```

This fixes the error in WordListContentProvider.count(), but breaks WordListAdapter.getItemCount(), which expects a long from WordListOpenHelper.

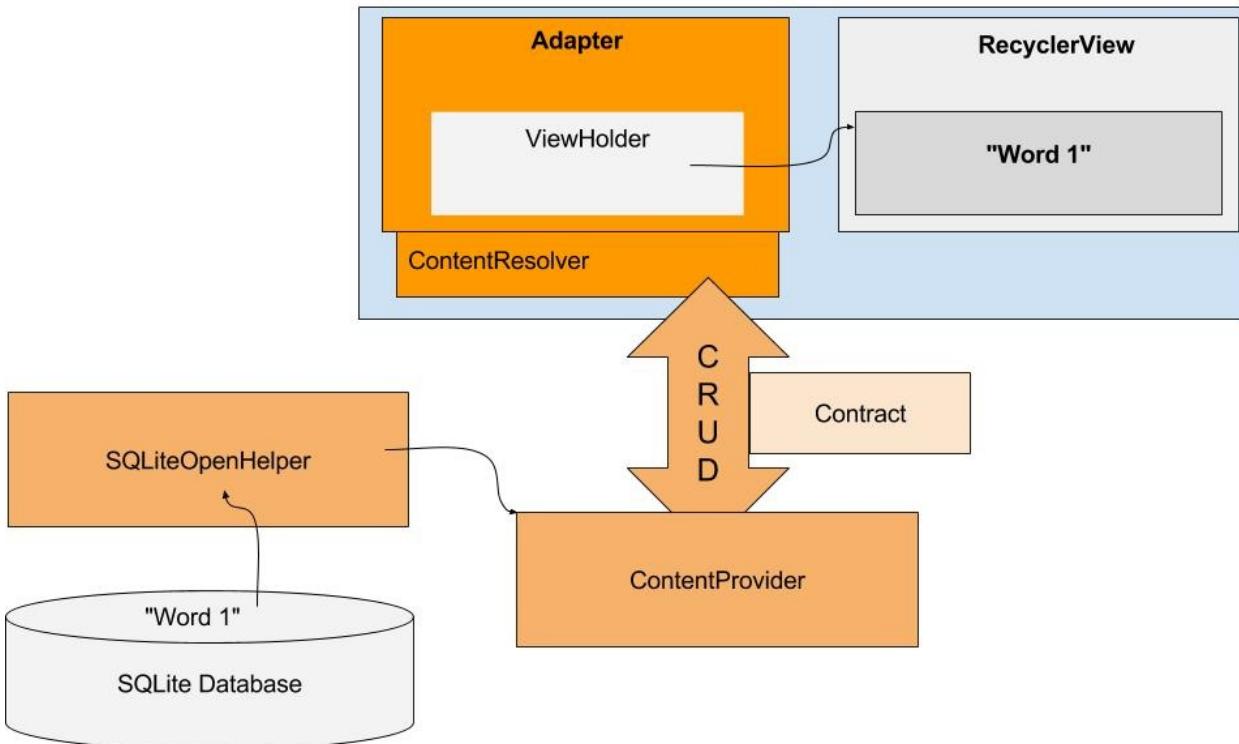
In WordListAdapter.onBindViewHolder(), instead of calling the database directly, you will have to use content resolvers, which you will do next.

### 3.5 Fix WordListAdapter.onBindViewHolder() to use a content resolver

As a developer, you have a choice to make. You can continue and finish the work on the content provider and then fix the adapter. Or you can fix the problem with the adapter to get your app into a consistent state. Which directions you choose depends on your level of experience and expertise, familiarity with the app, and personal preference.

When learning, it is better to return to a consistent state without errors as often as possible, so that you can check your progress and have some confidence that your code is correct.

So, next, you will fix `WordListAdapter.onBindViewHolder()` to use a content resolver instead of calling the `WordListOpenHelper` directly.



1. In `WordListAdapter`, delete the `mDB` variable, since you are not referencing the database anymore. This shows errors in Android Studio, that guide subsequent changes.
2. In the constructor, delete the assignment to `mDB`.
3. Refactor > Change the signature of the constructor and remove the `db` argument.
4. Add class variables for the query parameters since they will be used more than once.

The content resolver takes a query parameter, which you must build. The query is similarly structured to a SQL query, but instead of a selection statement, it uses a URI. Query parameters are very similar to SQL queries.

```

private String queryUri = Contract.CONTENT_URI.toString(); // base uri
private static final String[] projection = new String[] {Contract.CONTENT_PATH}; /





```

5. In `onBindViewHolder`, delete the first two lines of code setting current and setting the text of the holder.
6. Define a string with the uri for this query, to fetch the item at the passed in position.
7. Define an empty String variable named `word`.
8. Define an integer variable called `id` and set it to -1.

9. Create a content resolver with the specified query parameters and store the results in a Cursor called cursor. (See MainActivity of MinimalistContentProvider app for an example.)

```
String uri = queryUri + "/" + position;
String word = "";
int id = -1;

Cursor cursor = mContext.getContentResolver().query(
    Uri.parse(uri), projection,
    selectionClause, selectionArgs, sortOrder);
```

10. Instead of just getting a WordItem delivered, WordListAdapter.onBindViewHolder() has to do the extra work of extracting the word from the cursor returned by the content resolver.
- If the returned cursor contains data, extract the word and set the text of the view holder.
  - Extract the id, because you'll need it for the click listeners.
  - Close the cursor. Remember that you did not close the cursor in WordListOpenHelper.query(), because you returned it.
  - Handle the case of no data in the cursor.

```
if (cursor.moveToFirst() && cursor.getCount() >= 1){
    int indexWord =
        cursor.getColumnIndex(Contract.WordList.KEY_WORD);
    word = cursor.getString(indexWord);
    holder.wordItemView.setText(word);

    int indexId =
        cursor.getColumnIndex(Contract.WordList.KEY_ID);
    id = cursor.getInt(indexId);
} else {
    holder.wordItemView.setText("ERROR: NO WORD");
}
```

11. Fix the parameters for the click listeners for the two buttons:
- current.getId() ⇒ id
  - current.getWord() ⇒ word
12. Replace the call to mDB.delete(id) with a content resolver call to delete.

```
selectionArgs =
new String[]{Integer.toString(id)};
int deleted =
mContext.getContentResolver().delete(
    Contract.CONTENT_URI, Contract.CONTENT_PATH, selectionArgs);
```

## 3.6 Fix WordListAdapter.getCount() to use a content resolver

Fix WordListAdapter.getCount() to

- use a content resolver query to get the item count
- use the ROW\_COUNT\_URI in your query
- extract count from the cursor and return it
- return -1 otherwise
- remember to close the cursor

Use the code you just wrote for onBindViewHolder as a guideline.

**Solution:**

```
@Override
public int getItemCount() {
    int count = -1;
    Cursor cursor = mContext.getContentResolver().query(
        Contract.ROW_COUNT_URI, projection,
        selectionClause, selectionArgs, sortOrder);

    if (cursor.moveToFirst() && cursor.getCount() >= 1){
        count = cursor.getInt(0);
    }
    cursor.close();
    return count;
}
```

## 3.7 Add the content provider to the Android Manifest

1. Run your app.
2. Examine logcat for the (very common) cause of the crash.
3. Add the content provider to the Android Manifest inside the `<application>` tag.

```
<provider
    android:name=".WordListContentProvider"
    android:authorities="com.android.example.wordlistsqllwithcontentprovider.provider">
</provider>
```

4. Run your app.

Your app should run and be fully functional. If it is not, compare your code to the supplied solution code, and use the debugger and logging to find the problem.

Do not continue, if your app does not run.

## 3.8 A moment of reflection...

You have implemented a content provider and its query() method.

You followed the errors to update methods in the WordListOpenHelper and WordListAdapter classes to work with the content provider.

When you run your app, for queries, the method calls go through the content provider.

For the insert, delete, and update operations, your app is still calling WordListOpenHelper.

With the infrastructure you have built, implementing the remaining methods is a lot less work.

# Task 4. Implementing Content Provider methods

## 4.1 getType()

The getType() method is called by other apps that want to use this content provider, to discover what kind of data your app returns.

Use a switch statement to return the appropriate MIME types.

- The MIME types are listed in the contract.
- SINGLE\_RECORD\_MIME\_TYPE is for URI\_ALL\_ITEMS\_CODE
- MULTIPLE\_RECORDS\_MIME\_TYPE is for URI\_ONE\_ITEM\_CODE

**Solution:**

```
@Nullable  
@Override  
public String getType(Uri uri) {  
    switch (sUriMatcher.match(uri)) {  
        case URI_ALL_ITEMS_CODE:  
            return MULTIPLE_RECORDS_MIME_TYPE;  
        case URI_ONE_ITEM_CODE:  
            return SINGLE_RECORD_MIME_TYPE;  
        default:  
            return null;  
    }  
}
```

**Challenge:** How can you test this method, as it is not called by your app. Can you think of three different ways of testing that this method works correctly?

## 4.2 Call the content provider to insert and update words in MainActivity

To fix insert operations `MainActivity().onActivityResult` needs to call the content provider instead of the database for inserting and updating words.

1. In `MainActivity`, delete the declaration of `mDB` and its instantiation.

In `OnActivityResult()`:

Inserting:

2. If the word length is not null, create a `ContentValues` values variable and add the word to it with the key "word".
3. Replace `mDB.insert(word);` with an insert request to `a` to a content resolver.

Updating:

4. Replace `mDB.update(id, word);` with an update request to `a` to a content resolver.

### Solution snippet:

```
// Update the database
if (word.length() != 0) {
    ContentValues values = new ContentValues();
    values.put("word", word);
    int id = data.getIntExtra(WordListAdapter.EXTRA_ID, -99);

    if (id == WORD_ADD) {
        getContentResolver().insert(Contract.CONTENT_URI, values);
    } else if (id >= 0) {
        String[] selectionArgs = {Integer.toString(id)};
        getContentResolver().update(Contract.CONTENT_URI, values, Contract.WordList.KEY_ID, selectionArgs
    );
    }
    // Update the UI
    mAdapter.notifyDataSetChanged();
}
```

## 4.3 Implement insert() in the content provider

The `insert()` method in the content provider is a pass-through. So you

1. call the `OpenHelper insert()` method,

2. convert the returned long id to a content URI to the inserted item,
3. and return that URI.

Android Studio reports an error for the values parameter, which you will fix in the next steps.

#### Solution:

```
public Uri insert(Uri uri, ContentValues values) {  
    long id = mDB.insert(values);  
    return Uri.parse(CONTENT_URI + "/" + id);  
}
```

## 4.4 Fix insert() in WordListOpenHelper

Android Studio reports an error for the values parameter.

1. Open WordListOpenHelper. The insert() method is written to take a String parameter.
2. Change the parameter to be of type ContentValues.
3. Delete the declaration and assignment of values in the body of the method.

## 4.5 Implement update() in the content provider

Fix the update methods in the same way as you fixed the insert methods.

1. In WordListContentProvider, Implement update(), which is one line of code that passes the id and the word as arguments.

```
return mDB.update(parseInt(selectionArgs[0]), values.getAsString("word"));
```

2. You don't need to make any changes to update in WordListOpenHelper.

## 4.6 Implement delete() in the content provider

In WordListContentProvider, Implement the delete() method by calling the delete() method in WordListOpenHelper with the id of the word to delete.

```
return mDB.delete(parseInt(selectionArgs[0]));
```

## 4.7 Run your app

Yup. That's it. Run your app and make sure everything works.

And if it doesn't, fix it, because you'll need the code in a later practical, when you'll write an app that uses this content provider to load word list data into its user interface,.

## Coding challenges

- The wordlist is just a list of single words, which isn't terribly useful. Extend the app to display definitions, as well as a link to useful information, such as developer.android.com, stackoverflow, or wikipedia.
- Add an activity that allows users to search for words.
- Add basic tests for all the functions in WordListContentProvider.

## Conclusion

Congratulations!

You have made your way through one of the most-feared Android features. In addition, you have practiced the real-life skill of rearchitecting, refactoring, and expanding existing code to meet a new requirement.

## Additional Resources

### Developer Documentation:

- Uniform Resource Identifiers or URIs
- MIME type
- MatrixCursor and Cursors
- Content Providers

### Videos:

- Android Application Architecture
- Android Application Architecture: The Next Billion Users

## 12.3 P: Sharing content with other apps

### Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Make your content provider available to other apps
- Summary
- Resources

In this practical you will share your content provider and access it from another app.

### What you should already KNOW

For this practical you should be familiar with:

- Content providers
- Adapters

### What you will LEARN

- How to access another app's content provider.
- Setting basic permissions

### What you will DO

You will make a copy of the content provider app, remove its content provider, and get data from the original app.

You will modify WordListSQLWithContentProvider to allow read and write access to its data.

Note that this finished app will also be the starter app for the Loader practicals.

### Apps Overview

You need two apps for this practical.

- WordListSQLWithContentProvider
- Stripped copy of WordListSQLWithContentProvider.

The UI and functionality of these two apps are unchanged from previous practicals.

## Task 1. Make your content provider available to other apps

By default, apps cannot access data of other apps. They need permission to do so. In the case of user data, that permission comes from the user. In the case of a content provider, the permission comes from the content provider.

There are two kinds of permissions:

- System permissions are predefined by the system. For example, if your app wanted to read a user's calendar, it needs to request the `READ_CALENDAR` permission from user.
- Developer defined permissions. Your content provider

To make your content provider available to other apps, you need to specify the grantable permissions in the `AndroidManifest` of the provider, and declare it in the `Android Manifest` of the client.

Permissions are not covered in detail in these practicals. You can learn more in [Declaring Permissions](#), [System Permissions](#), and [Implementing Content Provider Permissions](#).

### 1.1. Modify WordListWithContentProvider to allow apps access

1. Create a copy of the `WordListSQLWithContentProvider` folder and call it `WordListClient`.
2. Open the original `WordListSQLWithContentProvider` in Android Studio.
3. Open the `AndroidManifest.xml` file.
4. Add an export statement inside the `<provider>`.

```
        android:exported="true"
```
5. Declare the required read and write permission, which is the default. Put the declaration at the top level, inside the `<manifest>` tag.

It is good practice to use your unique package name in order to keep the permission unique.

```
<permission  
    android:name="com.android.example.wordlistsqllwithcontentprovider.PERMISSION" /  
>
```

6. Run the app and leave it on the device.
7. The app with the content provider has to be installed on the device. It is not necessary for it to be running.

## 1.1. Create the Client app

1. Open WordListClient in Android Studio.
2. Rename the package (Refactor > Rename) to wordlistclient.
3. Open build.gradle(Module:app) and change the app id to wordlistclient.
4. In strings.xml, change the app name to WordListClient, to help you tell the apps apart.
5. Note that at this point, you can't install both apps, because there is a provider conflict. Both providers have the same name.
6. In the Android Manifest, remove the `<provider>` declaration.
7. In the Android Manifest, add the `<uses-permission>` for the content provider at the top level, inside the `<manifest>` tag.

```
<uses-permission android:name = "com.android.example.wordlistsqllwithcontentprovider.PERMISSION"/>
```

8. Delete the ContentProvider class.
9. Delete the WordListOpenHelper class.
10. Run WordListClient.
11. Interact with both apps. Changes made by one app are reflected in the other app.  
(Scroll the screen to trigger a screen update.)

## Coding challenges

No coding challenge.

## Summary

In this chapter, you learned how to access a content provider from another app.

## Resources

**Developer Documentation:**

- Working with System Permissions
- Implementing Content Provider Permissions

# 13.1 P: Load and display data fetched from a content provider

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. MainActivity](#)
- [Task 2: WordListAdapter](#)
- [Summary](#)
- [Resources](#)

In this practical you will learn how to load data provided by another app's content provider in the background and display it to the user, when it is ready.

Querying a [ContentProvider](#) for data you want to display takes time. If you run the query directly from an [Activity](#), it may get blocked and cause the system to issue an "Application Not Responding" message. Even if it doesn't, users will see an annoying delay in the UI. To avoid these problems, you should initiate a query on a separate thread, wait for it to finish, and then display the results.

You can do this in a straightforward way by using an object that runs a query asynchronously in the background and reconnects to your [Activity](#) when it's finished. You do this with a loader, specifically, a [CursorLoader](#). Besides doing the initial background query, a CursorLoader automatically re-runs the query when data associated with the query changes.

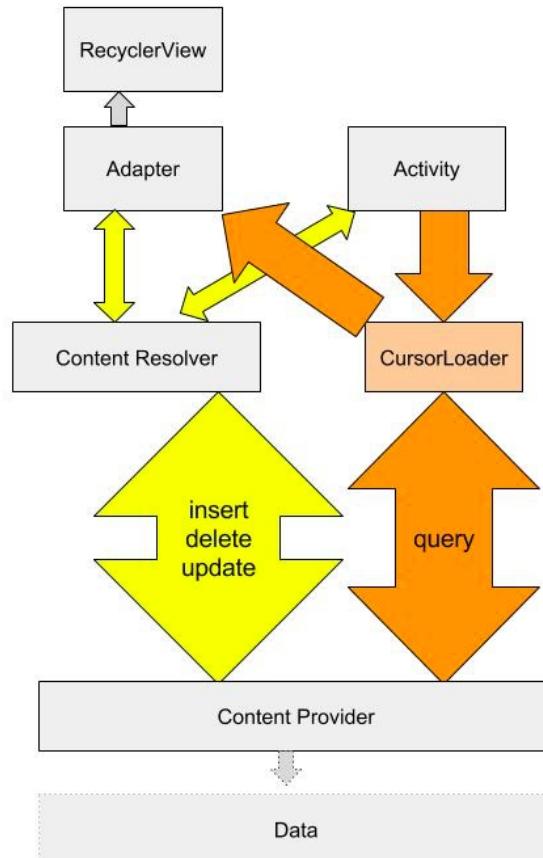
You have used an [AsyncTaskLoader](#) in a previous practical. The CursorLoader extends AsyncTaskLoader to specifically work with content providers, saving you a lot of work.

At a high level, you need the following pieces to display data from a content provider:

- An [Activity](#) or fragment.
- An instance of the [LoaderManager](#) in the Activity.
- A [CursorLoader](#) to load data backed by a [ContentProvider](#).
- An implementation for [LoaderManager.LoaderCallbacks](#), an abstract callback interface for the client to interact with the LoaderManager.
- A way of displaying the loader's data, commonly via an adapter.

The following diagram shows a simplified version of app architecture with a loader. The loader performs querying for items in the background. If the data changes, it gets a new set of data for the adapter. Using a CursorLoader, this is all taken care of automatically.

Note that it is entirely possible to build custom loaders. But since the Android system provides you with an elegant solution that saves you a lot of work, consider how you can use it as given before implementing your own solution from scratch.



## What you should already KNOW

For this practical you should be familiar with:

- Displaying data in a RecyclerView
- Adapters
- Cursors (see previous practical and concepts)
- AsyncTaskLoader
- Content Providers

## What you will LEARN

- How to access another app's content provider using a CursorLoader.

## What you will DO

In this practical you are going copy and modify WordListClient to use a CursorLoader instead of a content resolver to query the content provider.

You will make changes only to the WordListAdapter and MainActivity.

## App Overview

The user visible functionality of WordListClient is not going to change. If you had a lot of words, you could measure a difference in load time between WordListClient. Users care a lot about load times and will abandon apps that load slowly at a much higher rate.

### IMPORTANT:

- Use the final code for WordListWithContentProvider. Make sure it has the permissions added from 10.3. You can also get the code [here](#).
- Use the WordListClient that you built in 10.3. You can also get the code [here](#).

## Task 1. MainActivity: Adding a LoaderManager and LoaderCallbacks

The LoaderManager is a convenience class that manages all your loaders. You only need one loader manager per activity. For example, the loader manager takes care of registering an observer with the content provider, which receives callbacks when data in the content provider changes.

### 1.1 Add the Loader Manager

1. Open MainActivity.java
2. Extend the class signature to implement LoaderManager.LoaderCallbacks. Make sure you use the support library version.

```
public class MainActivity extends AppCompatActivity implements LoaderManager.LoaderCallbacks<Cursor>
```

3. Implement method stubs for onCreateLoader(), onLoadFinished(), and onLoaderReset().

4. In `onCreate()`, create a `LoaderManager` and register a loader with it. The first argument is a numeric tag; since you only have one loader, it doesn't matter what number you choose. You are not passing in any data, and bind the loader to the main activity.

```
getSupportLoaderManager().initLoader(0, null, this);
```

## 1.2 Implement `onCreateLoader()`

The `LoaderManager` calls this method to create the loader, if it does not already exist.

1. Create a `queryUri` and projection. The `CursorLoader` requires a URI for the query, and a context. Use the same URI that the content resolver is using to query the content provider. You can find it in the Contract.
2. Return the `CursorLoader`.

```
@Override  
public Loader<Cursor> onCreateLoader(int id, Bundle args) {  
    String queryUri = CONTENT_URI.toString();  
    String[] projection = new String[] {CONTENT_PATH};  
    return new CursorLoader(this, Uri.parse(queryUri), projection, null, null,  
}
```

## 1.3 Implement `onLoadFinished()`

1. When loading has finished, send the data to the adapter.
2. Call `setData()`, which you will implement in the next task, passing in the cursor.

```
@Override  
public void onLoadFinished(Loader<Cursor> loader, Cursor data) {  
    mAdapter.setData(data);  
}
```

## 1.4 Implement `onLoadReset()`

On reset, let the adapter know that the data has become unavailable by passing `null` to `setData()`.

```
@Override  
public void onLoaderReset(Loader<Cursor> loader) {  
    mAdapter.setData(null);  
}
```

## 1.5 Fix onActivityResult()

In `onActivityResult()`, the last line before the `else` statement, where you notify the UI to update itself, is no longer necessary, as the Loader should deliver new data to the UI when the underlying data changes.

In theory.

However, in practice, this does not work, and so you need to ask the LoadManager to restart the loader.

1. Delete: `mAdapter.notifyDataSetChanged();`
2. Insert: `getSupportLoaderManager().restartLoader(0, null, this);`

# Task 2. WordListAdapter

## 2.1. Implement setData()

1. Create a private class variable `mCursor` and initialize it to null.
2. Implement the public method `setData`. It takes a cursor parameter and returns nothing.
3. In the body, set `mCursor` to the parameter and call `notifyDataSetChanged`.

```
public void setData(Cursor cursor) {  
    mCursor = cursor;  
    notifyDataSetChanged();  
}
```

## 2.1. Fix onBindViewHolder()

1. In `onBindViewHolder`, remove the call to the content resolver and the `uri` variable.
2. Rename `cursor` to `mCursor`.
3. Fix the method to handle the case where the cursor is null.
4. Fix the method to handle the case where the RecyclerView is being built, but data is not available yet. For example, do this by using a placeholder value.
5. After inserting a new item, if the loader has not finished, position will be larger than the count, causing a null pointer exception. Handle this situation gracefully.
6. Remove the line that closes the cursor. The Loader handles this for you.

```
@Override  
public void onBindViewHolder(WordViewHolder holder, int position) {  
    String word = "";  
    int id = -1;  
  
    if (mCursor != null) {  
        int count = mCursor.getCount();  
        if (position < count){  
            mCursor.moveToPosition(position);  
            int indexWord = mCursor.getColumnIndex(Contract.WordList.KEY_WORD);  
            word = mCursor.getString(indexWord);  
            holder.wordItemView.setText(word);  
  
            int indexId = mCursor.getColumnIndex(Contract.WordList.KEY_ID);  
            id = mCursor.getInt(indexId);  
        } else {holder.wordItemView.setText("Waiting...");}  
    } else {  
        holder.wordItemView.setText("ERROR: NO WORD");  
    }  
    [... rest of method unchanged...]
```

## Coding challenges

No coding challenge.

## Conclusion

In this chapter,

## Resources

### Developer Documentation:

- [Loaders](#)
- [Running a query with a CursorLoader](#)
- [CursorLoader](#).

# Appendix Utilities

## Compare Custom Objects

Whenever your data model calls for objects to be sorted, it becomes necessary to define how these objects can be compared to each other. Do they have some kind of member variable that represents their rank? There are many reasons that you would need to compare your objects, and the [Comparable](#) interface allows you to do just that. The Comparable interface requires that you implement a single method: `compareTo(<T> another)` where `<T>` is the parameterized type you implemented Comparable with, and the type of object you are comparing to (i.e if you want to compare your Foobar instance to other Foobar instances, you would implement `Comparable<Foobar>` and your `compareTo` method would take Foobar as a parameter). The `compare` method should do the following:

- return a negative integer if the object is less than the parameter.
- return a positive integer if the object is greater than the parameter.
- return zero if the objects are equal.

This allows for the class to be sorted in a natural order.

## Copy and rename a project

For some of the lessons, you will need to make a copy of a project before making new changes. You may also want to copy a project to use some of its code in a new project. In either case you can copy the project, and then rename and refactor the new project's components to use the new project's name.

### 1. Copy the project

1. On your computer's file system (not in Android Studio), make a copy of the directory containing the existing project, which we'll call **ExistingProject** (substitute the name of your existing project). The copied directory is automatically named **ExistingProject Copy**.
2. Rename the **ExistingProject Copy** directory to **NewProject** (substitute the new name you want to use for the new project).

### 2. Rename and refactor the project components

The old name of the project, **ExistingProject**, still appears throughout the packages and files in the new copy of your project. Use these steps in Android Studio to change the file and the package references in your app to the new name.

1. Start Android Studio, and click **Open an existing Android Studio project**. Navigate to the **NewProject** directory, select it, and click **OK**.
2. Select **Build > Clean Project** to remove the auto-generated files.
3. Click the **1:Project** side-tab to see your files in the Project view.
4. Expand **app > java**, select the **com.example.android.existingproject** folder, and choose **Refactor > Rename**.
5. Click **Rename Package**.
6. Change **existingproject** to **newproject**.
7. Check the **Search in comments and strings** and **Search for text occurrences** options, and then click **Refactor**. The Find Refactoring Preview pane appears, showing code to be refactored.
8. Click **Do Refactor**.
9. Expand **res > values** and double-click the **strings.xml** file.
10. Change the **name="app\_name"** string to **New Project**.

### 3. Update the build.gradle and AndroidManifest.xml files

Each app you create must have a unique application ID, as defined in the app's build.gradle file. Even though the above steps should have changed the build.gradle file, you should check it to make sure, and also sync the project with the gradle file:

1. Expand **Gradle Scripts** and double-click **build.gradle (Module: app)**.
2. Under **defaultConfig**, check to make sure that the value of the **applicationID** key has been changed to "**com.example.android.newproject**". If it has not changed, change it now.
3. Click **Sync Now** in the top right corner of the Android Studio window.

**Tip:** You can also choose **Tools > Android > Sync Project with Gradle File** to sync your gradle files.

In addition, some apps include the app name in readable form (such as "New Project" rather than newproject) as a label in the AndroidManifest.xml file. Follow these steps to check for and, if necessary, change the label:

1. Expand **app > manifests** and double-click **AndroidManifest.xml**.
2. Find the statement below, and change the label if necessary to the string resource for the new app name:

```
android:label="@string/app_name"
```

# Delete a project

All the files for an Android project are contained in the project's folder on the computer's file system. To delete a project you can just delete the folder.

However, Android Studio also keeps a list of recent projects that you have opened, which is different from the project folders on the file system. You can delete a project from the list of recent projects in Android Studio, but the project files will still remain on the computer's filesystem. Conversely if you delete a project folder on the filesystem and then try to open it from the project list from within AS, you'll get a "can't find that folder" error.

So to delete a project altogether from everywhere, you need to:

1. Delete the folder from the filesystem by moving it to the trash or using `rm -rf` in the shell.
2. On the initial Android Studio screen, click the name of the project and press delete. -  
OR-

Select File > Open Recent > Manage Projects, click the name of the project and press delete.

# Extract Resources

## 1. Extract Strings

In order for your app to be localizable into multiple languages, it is best practice to keep all of your string resources in the same place: your `strings.xml` file.

### Create string resources:

There are several ways to create string resources:

- Add them manually in the `strings.xml` file using the following syntax:

```
<string name="string_name">String Value</string>
```

- Wherever the string will be used, i.e. the `text` attribute of a `TextView`, type in the desired name for a string resource in the following format: `@string/string_name`. It will be highlighted in red since the resource does not yet exist. Make sure your cursor is in the highlighted text and press **Alt + Enter** and select **Create string value resource**. Enter your desired string and press **OK**. That's it! The string gets automatically added to your `strings.xml` file.
- You can also select any existing, hard-coded string in either XML or Java and press **Alt**

+ Enter, and select **Extract string resource**.

## Access string resources:

You will need to access your strings in two ways: in XML and in Java.

In XML, you should access the string resource using the following syntax:

```
@string/string_name .
```

In Java, access the string resource using the following syntax:

```
getString(R.string.string_name) .
```

## 2. Extract Dimensions

Hard-coded dimensions should also be moved to their own location: the dimens.xml file.

This allows for you to specify different dimensions using [resource qualifiers](#). The process is identical to extracting strings, except the destination is the dimens.xml file rather than the strings.xml file.

## 3. Extract Styles

If you have several elements that share attributes, it may be convenient to define these attributes in the style.xml file. To learn more about the appropriate use of Styles, see the [Styles and Themes](#) lesson. To extract existing attributes into a style that you can then apply to other views, do the following:

1. Place your cursor in the view for which you want to extract the attributes.
2. Right click and select **Refactor > Extract > Style**.
3. Name the style, and select the desired attributes. If the **Launch 'Use Style WHere Possible' refactoring after the sylte is extracted** is checked, Android Studio will search the rest of the file for the selected attributes, and apply the style to views where the attributes match.
4. Click **OK**. That's it!

## Save State of Custom Objects

In Android, you will frequently create Custom objects to represent your particular Data Model. In order to preserve the state of these objects, you must be able to pass them into the savedInstanceState bundle. In order to do so, your custom class must implement the [Parcelable](#) interface. This allows for primitive types (int, string, byte, etc) to be saved in the savedInstanceState callback. Do the following:

1. After setting up the data in your custom class (only the primitive data types will be saved), add the Parcelable implementation to your class declaration:
2. The declaration will be underlined in red, since you have to implement the interface methods. With your cursor on the underlined text, press **Alt + Enter** and select **Implement methods**.
3. Choose both `describeContents()` and `writeToParcel(Parcel dest, int flags)`. Click **OK**.
4. The class name will still be underlined, indicating that the interface is not fully implemented yet. Select the class name, and again press **Alt + Enter** and choose **Add Parcelable implementation**. Android studio will automatically add the required code. Note the variables for which you want to preserve the state (primitive types) are written to the Parcel in the `writeToParcel` method.
5. You can now save the state of these objects using the `savedInstanceState` bundles methods: `putParcelable`, `putParcelableArray`, and `putParcelableArrayList` and the respective getters.