



Android Developer Fundamentals Course

Learn to develop Android Applications

Practical Workbook

Developed by Google Developer Training Team

December 2016



Table of Contents

Introduction	1.1
Unit 1. Get started	1.2
Lesson 1: Build your first app	1.2.1
1.1: Install Android Studio, Hello World, Logging	1.2.1.1
1.2A: Make Your First Interactive UI	1.2.1.2
1.2B: Using Layouts	1.2.1.3
1.3: Working with TextView Elements	1.2.1.4
1.4: Learning Resources	1.2.1.5
Lesson 2: Activities	1.2.2
2.1: Create and Start Activities	1.2.2.1
2.2: Activity Lifecycle and State	1.2.2.2
2.3: Activities and Implicit Intents	1.2.2.3
Lesson 3: Testing, debugging, and using support libraries	1.2.3
3.1: Using the Debugger	1.2.3.1
3.2: Testing your App	1.2.3.2
3.3: Using Support Libraries	1.2.3.3
Unit 2. User experience	1.3
Lesson 4: User interaction	1.3.1
4.1: Use Keyboards, Input Controls, Alerts, and Pickers	1.3.1.1
4.2: Use an Options Menu and Radio Buttons	1.3.1.2
4.3: Using the App Bar and Tabs for Navigation	1.3.1.3
4.4: Create a Recycler View	1.3.1.4
Lesson 5: Delightful user experience	1.3.2
5.1: Drawables, Styles, and Themes	1.3.2.1
5.2: Material Design: Lists, Cards and Colors	1.3.2.2
5.3: Supporting Landscape, Multiple Screen Sizes and Localization	1.3.2.3
Lesson 6: Testing your UI	1.3.3
6.1: Use Espresso to test your UI	1.3.3.1
Unit 3. Working in the background	1.4
Lesson 7: Background Tasks	1.4.1

7.1: Create an AsyncTask	1.4.1.1
7.2: Connect to the Internet with AsyncTask and AsyncTaskLoader	1.4.1.2
7.3: Broadcast Receivers	1.4.1.3
Lesson 8: Triggering, scheduling and optimizing background tasks	1.4.2
8.1: Notifications	1.4.2.1
8.2: Alarm Manager	1.4.2.2
8.3: Job Scheduler	1.4.2.3
Unit 4. All about data	1.5
Lesson 9: Preferences and Settings	1.5.1
9.1: Shared Preferences	1.5.1.1
9.2: Adding Settings to an App	1.5.1.2
Lesson 10: Storing data using SQLite	1.5.2
10.2A: SQLite Database	1.5.2.1
10.2B: Searching a SQLite Database	1.5.2.2
Lesson 11: Sharing data with content providers	1.5.3
11.1A: Implement a Minimalist Content Provider	1.5.3.1
11.1B: Add a content provider to your database	1.5.3.2
11.1C: Sharing content with other apps	1.5.3.3
Lesson 12: Loading data using loaders	1.5.4
12.1: Load and display data fetched from a content provider	1.5.4.1
Appendix Utilities	1.6

Android Developer Fundamentals Course – Practicals

Learn to develop Android applications

Practicals Workbook

Developed by Google Developer Training Team

Last updated: December 2016



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

1.1: 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](#)
- [Related concepts](#)
- [Learn more](#)

Welcome to the practical exercises. You will learn to:

- 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 purposes.

What you should already KNOW

For this practical you should be able to:

- Understand the general software development process for object-oriented applications using an IDE (Integrated Development Environment).
- Demonstrate that you have at least 1-3 years of experience in object-oriented

programming, with some of it focused on the Java programming language. (These practicals will not explain object-oriented programming or the Java language.)

What you will NEED

For these practicals, 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

You will learn to:

- Install and use the Android IDE.
- Understand the development process for building Android apps.
- Create an Android project from a basic app template.

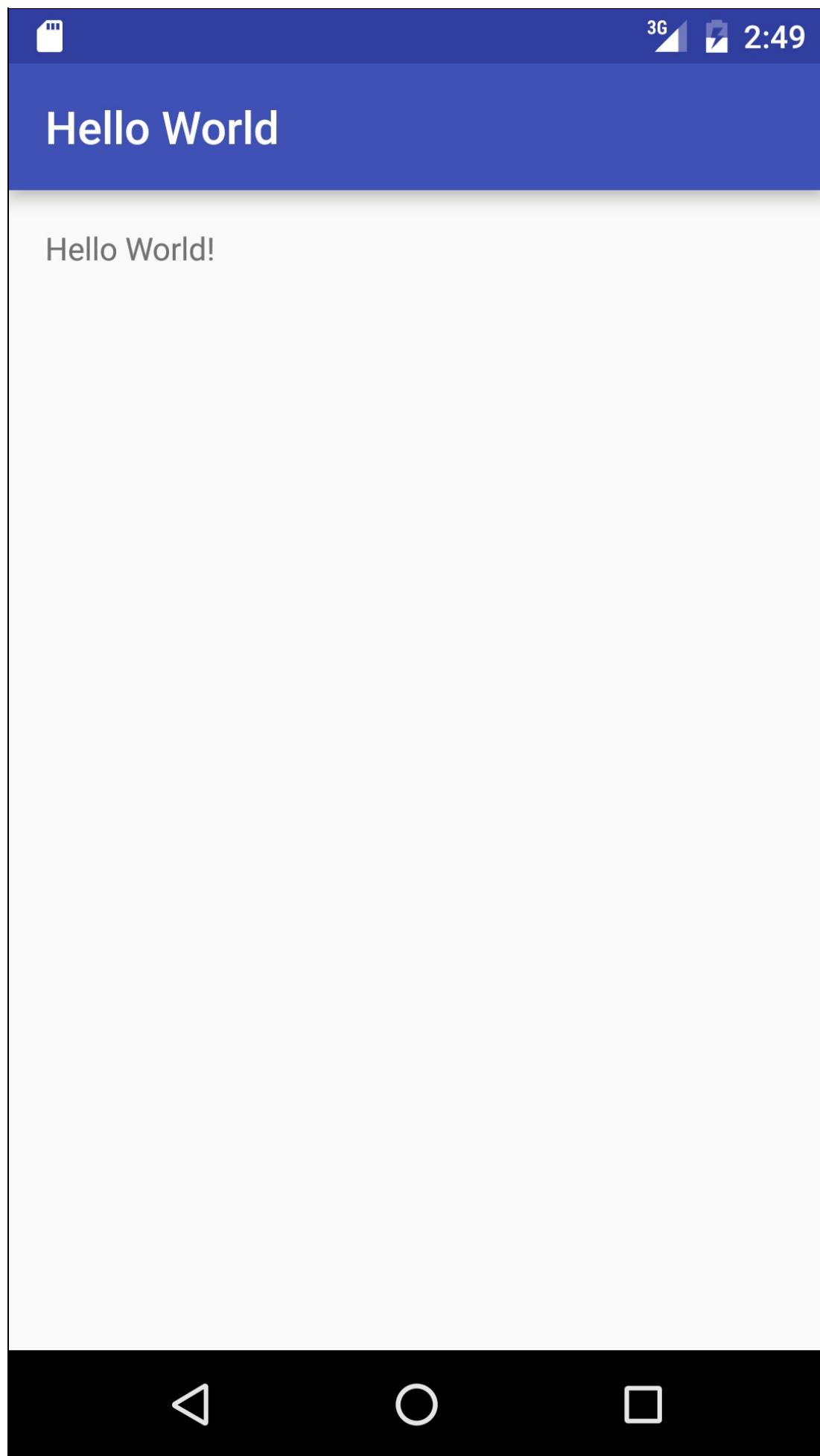
What you will DO

- 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, from a template, a new Android project for the 'Hello World' app. This simple app displays the string "Hello World" on the screen of the Android virtual or physical device.

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 advanced 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.

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 in the sections below.

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 is below 7 or if Java is not installed, you need to install the latest version of the Java SE development kit before installing Android Studio.

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 in order to proceed. Then download the version appropriate for the computer you are developing on.

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 enter 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 version 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/

Mac:

1. Open Terminal.
2. Confirm you have JDK by typing "which java".
3. Check that you have the needed version of Java, by typing "java -version".
4. Set JAVA_HOME using this command in Terminal: `export JAVA_HOME=`which java``
5. enter `echo $JAVA_HOME` to confirm the path.

Linux:

See: https://docs.oracle.com/cd/E19182-01/820-7851/inst_cli_jdk_javahome_t/

Important: Don't install Android Studio until after the Java JDK is installed. 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 correctly installed the JDK by going to the directory where you installed it. To identify where the JDK 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, and some of the steps may seem redundant.
3. When the download completes, 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 implement the "Hello World" app to verify that Android studio is correctly installed and learn the basics of developing with Android Studio.

2.1 Create the "Hello World" app

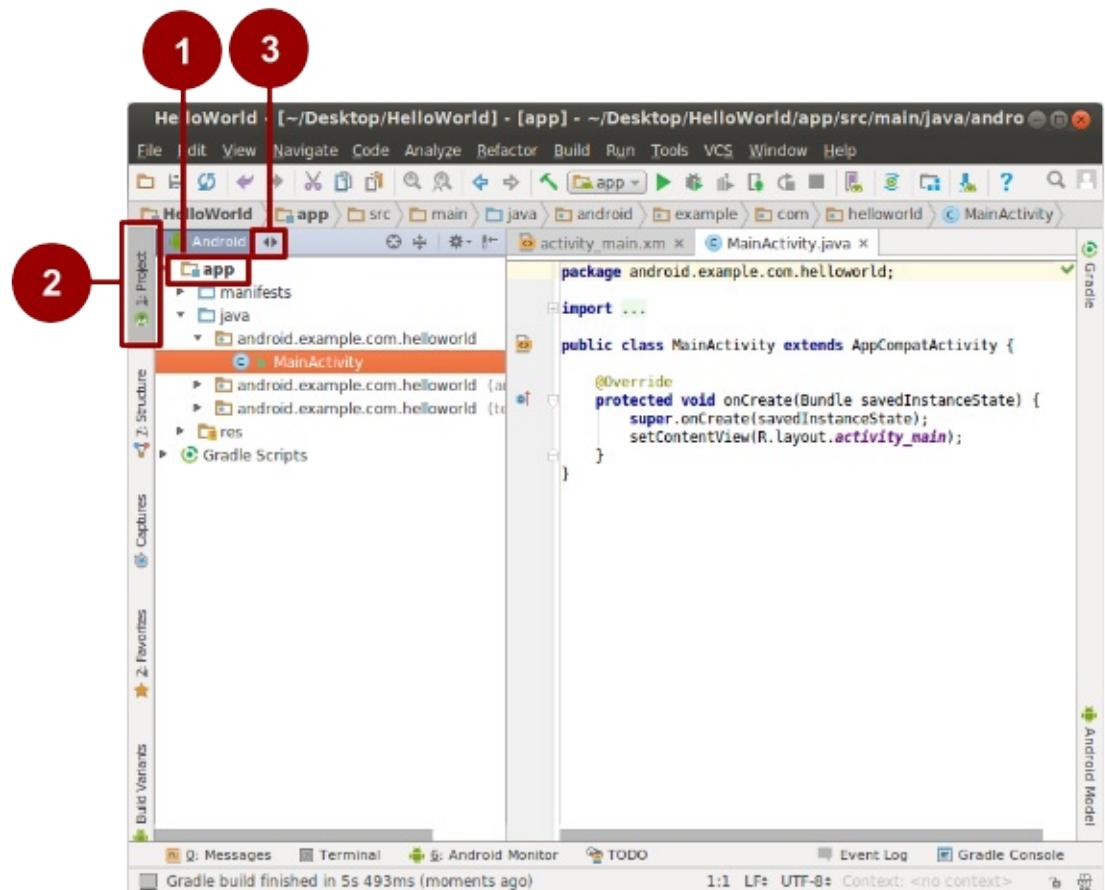
1. Launch Android Studio if it is not already opened.
2. In the main **Welcome to Android Studio** window, click "Start a new Android Studio project".
3. In the **New Project** window, give your application an **Application Name**, such as "Hello World".
4. Verify the Project location, or choose a different 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 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.

The Android Studio window should look similar to the following diagram:



You can look at the hierarchy of the files for 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.

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 of your 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 of 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 the domain name you have specified)**: 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 test 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 launcher 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-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 like strings, dimensions, and colors in your XML and Java 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 chosen theme, and you can add your own colors or change them based on your app's requirements.
 - **dimens.xml**. Store the 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 styles for your app and theme go here. Styles help give your app a consistent look for all UI elements.

3.2 The Gradle build system

Android Studio uses [Gradle](#) as its build system. As you progress through these practicals, you will learn more about gradle and what you need to build and run your apps.

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 that behaves exactly like a physical version of that device , but it resides on your computer .

Why: With virtual devices, you can test your apps on different devices (tablets, phones) with different API levels to make sure it looks good and works for most users. You do not need to 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 of 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 launcher 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.

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 it. If necessary, click the link to start the download, and click **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 go ahead and close it.)

Task 5. Run your app on an emulator

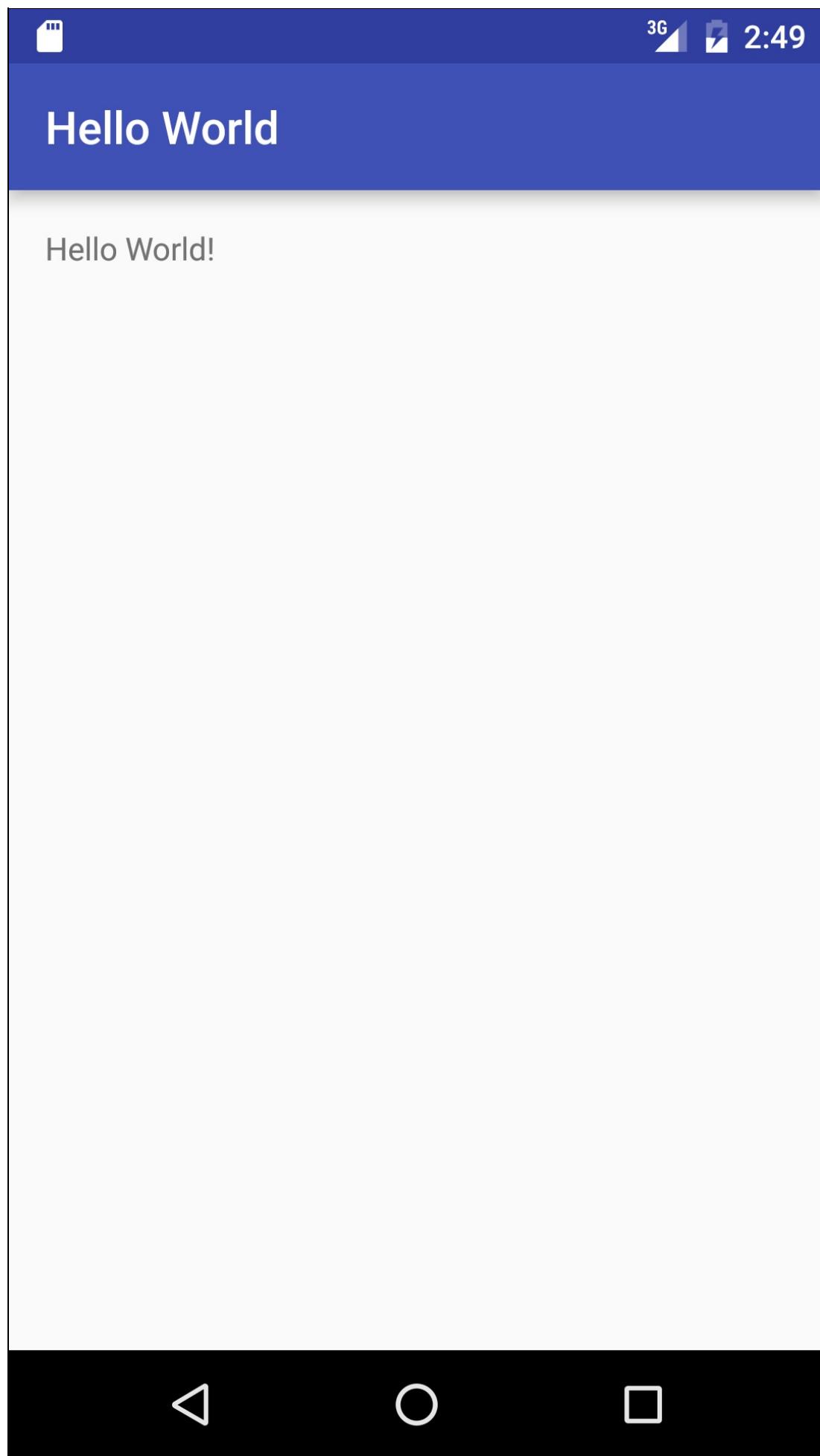
In this task, you will finally run your Hello World app.

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

Coding challenge

Note: All coding challenges are optional, and are not requirements for subsequent practicals.

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.

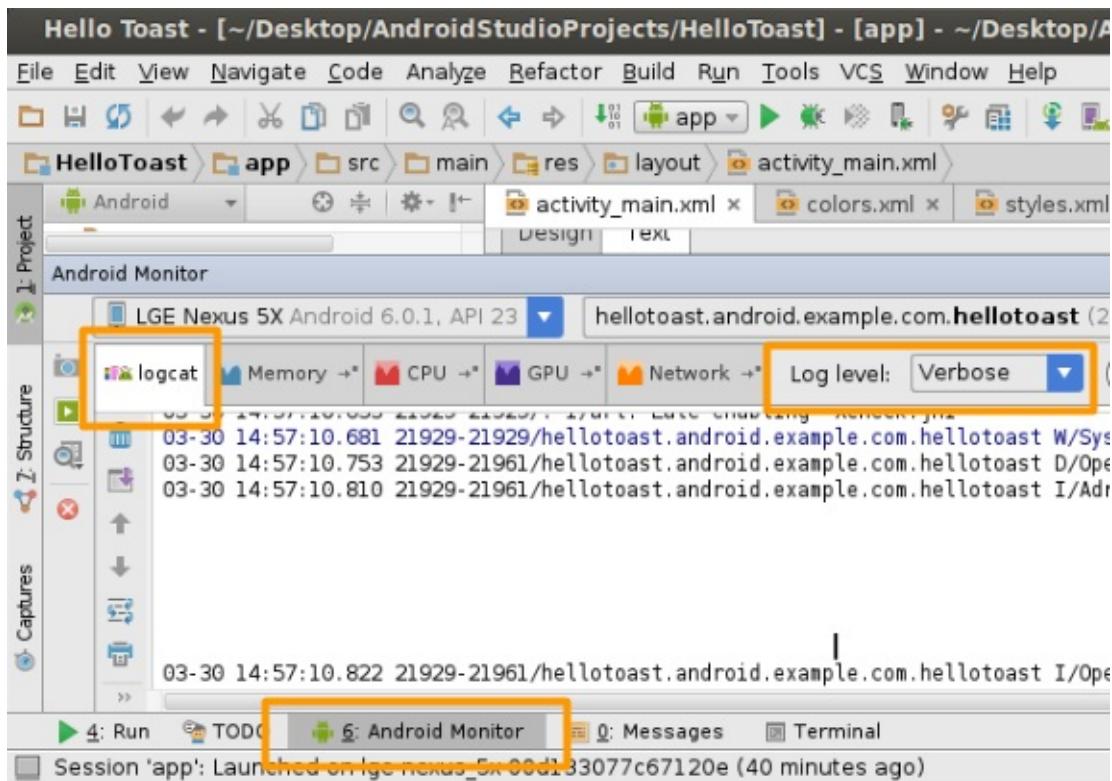
The **Android Monitor** displays information about your app.

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**.



Log 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:

- Log – The [Log class](#). API for sending log messages.
- 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.
- "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). Select all check 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:

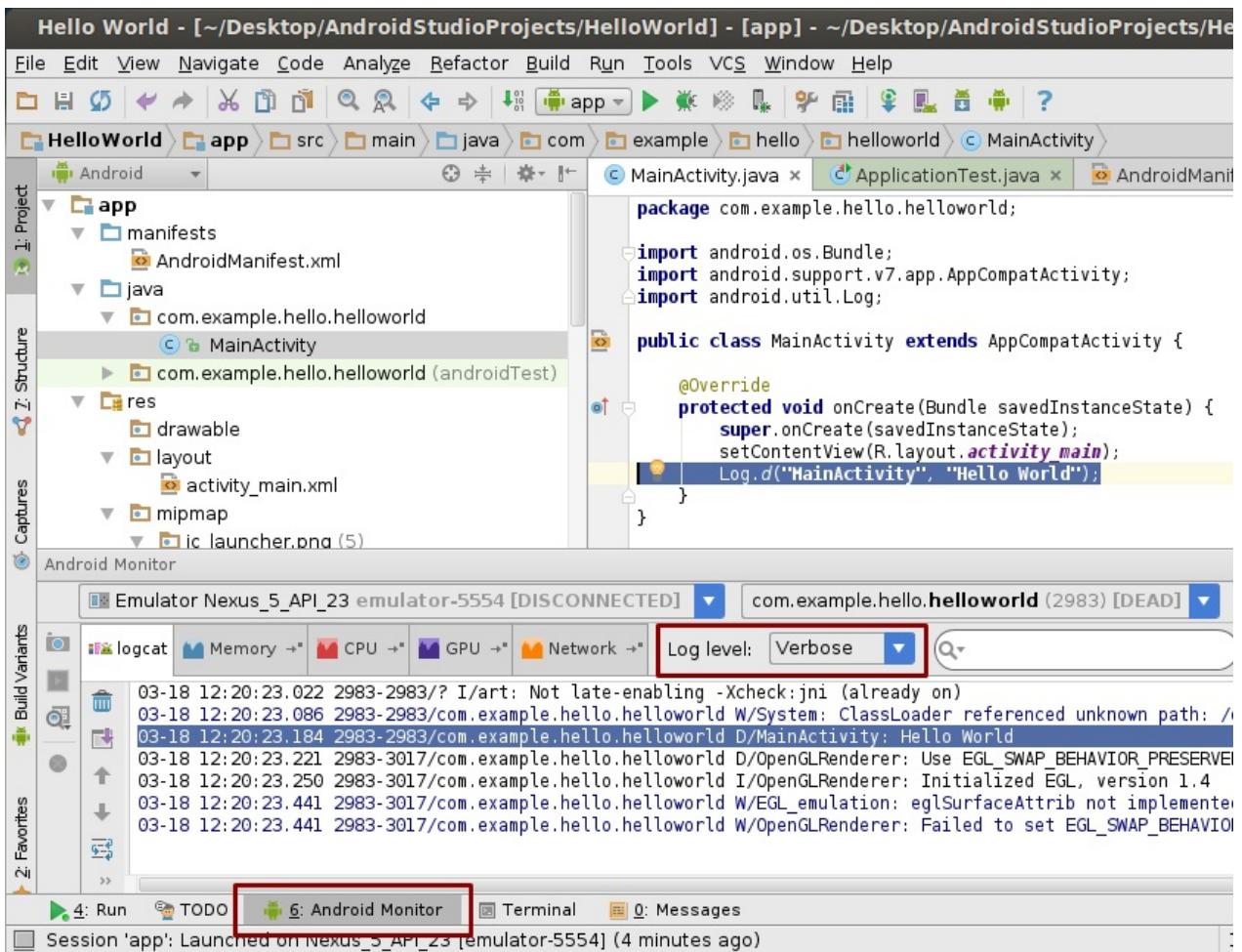
```
package com.example.hello.helloworld;

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");
    }
}
```

Output Log Message

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



Coding challenge

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

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 will see that the code actually refers to a string
        resource. Ctrl-click @string/app_name to see where the resource is
        specified. This will be covered in a later practical . -->
    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>
```

Coding challenge

Note: All coding challenges are optional.

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 learn 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'
}
```

- 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 the 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 Target** window opens with the list of available emulators and connected devices.
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](#).
- Check the latest documentation, programming forums, or get help from your instructors.

Coding challenge

Note: All coding challenges are optional.

Challenge: 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.

Summary

In this chapter, you learned to:

- Install Android Studio
 - Obtain a basic understanding of the development workflow once you have launched in Android Studio.
 - Have basic comprehension of the structure of an Android app in the build environment.
 - Have a basic understanding of the Android Manifest, and what it is used for.
 - Add log statements to the code that give you a basic tool for debugging.
- Deploy the Hello World app on the Android emulator and [optionally] on a mobile

device.

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Create Your First Android App](#)

Learn more

- [Android Studio download page](#)
- [How do I install Java?](#)
- [Android Studio documentation](#)
- [Supporting Multiple Screens](#)
- [Gradle Wikipedia page](#)
- [Reading and Writing Logs](#)

1.2A: 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
- Related concept
- Learn more

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. You specify the views in XML layout files. For example, views can be components that:

- display text ([TextView](#) class)
- allow you to edit text ([EditText](#) class)
- represent clickable 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 can explore the view hierarchy of your app in the Layout Editor's Component Tree pane.

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 the Hello World app 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 \(Model-View-Presenter\)](#) pattern.

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

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

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

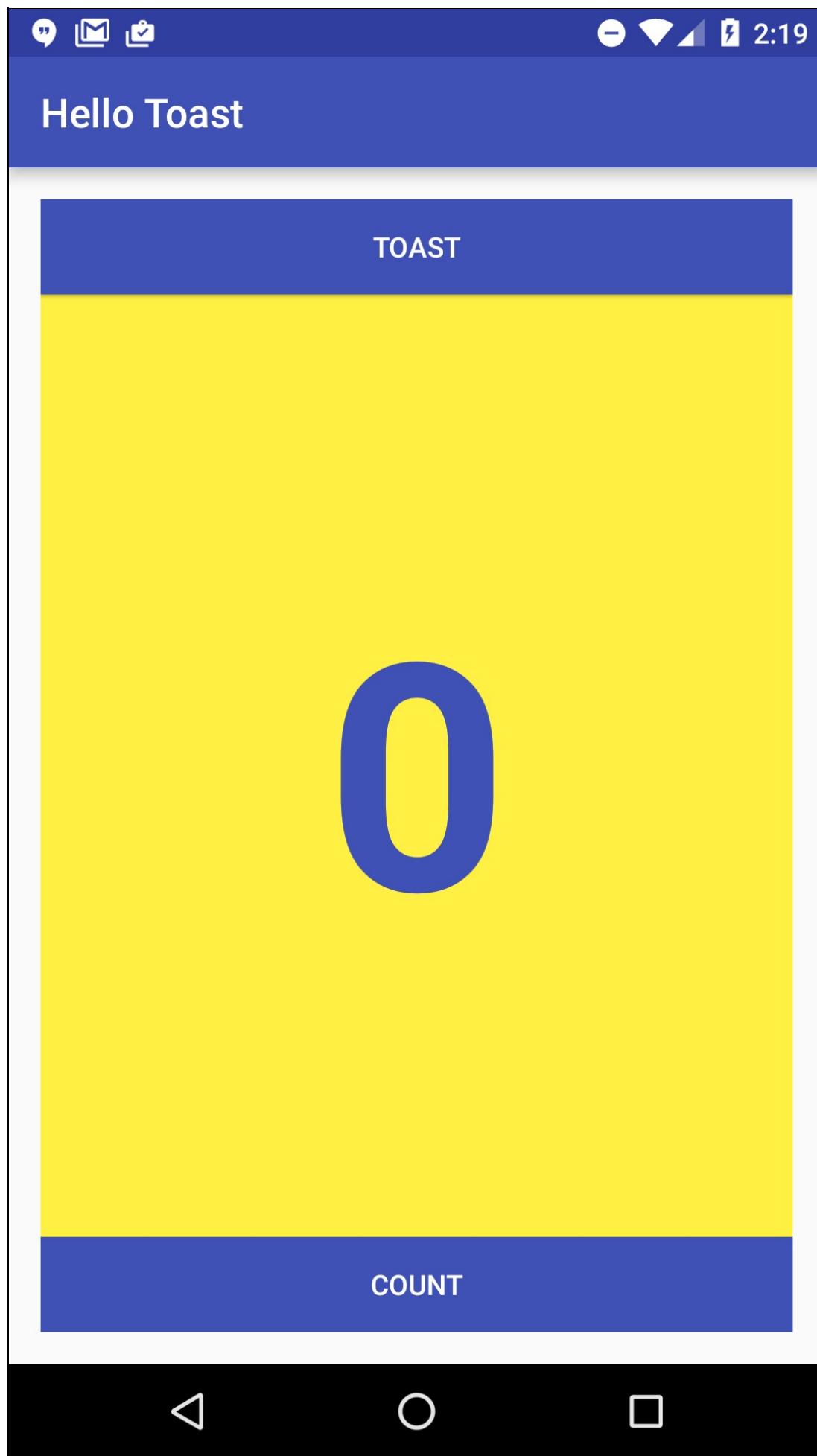
In this practical, you will:

- 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 design and implement 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 box	Checked

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

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

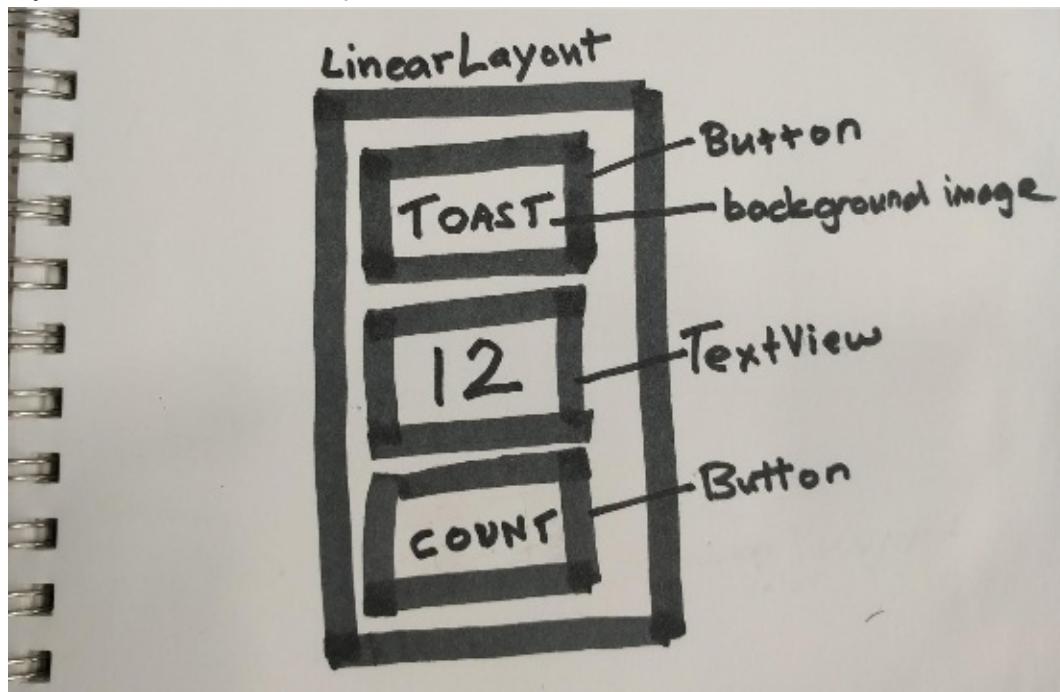
In this task, you will create and configure a user interface for the "Hello Toast" app by arranging view UI components on the screen.

Why: Every app should start with the user experience, even if the initial implementation is very basic.

[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

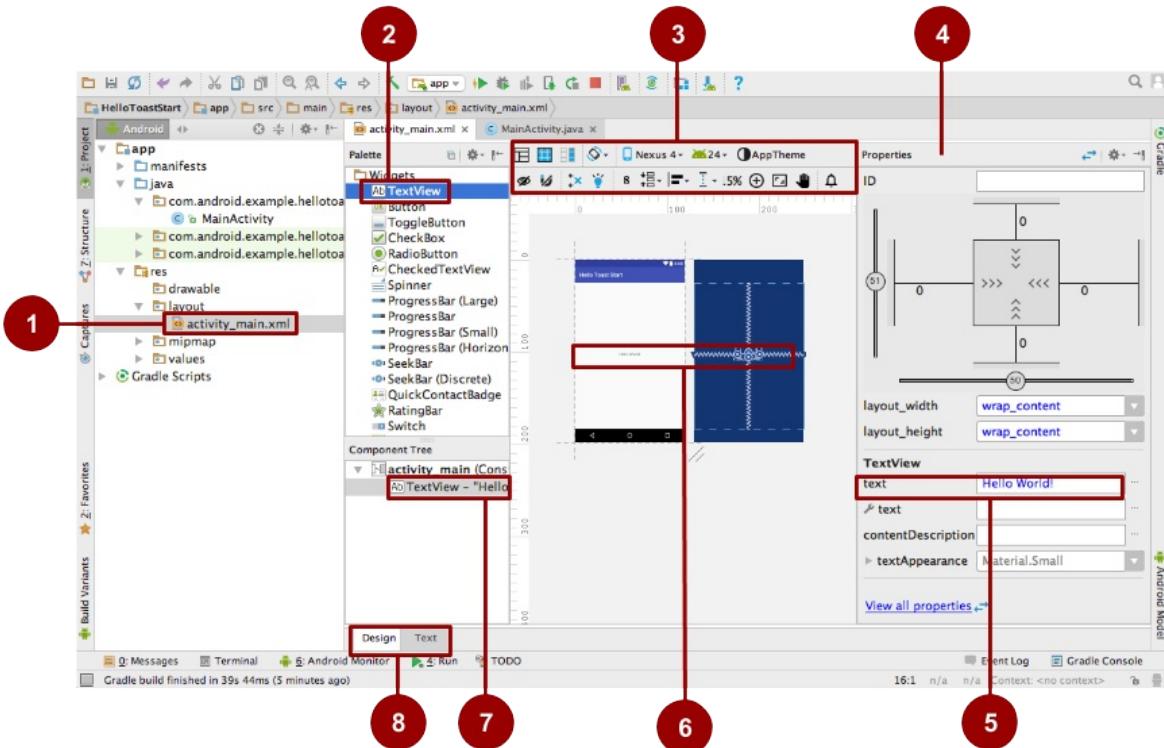
Use the Layout Editor to create the layout of the user interface elements, and to preview your app using different devices and app themes, resolutions, and orientations.

Refer to the screenshot below to match

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

The 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 a 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) developers 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 this is completed, undo the changes (for UI changes, use **Edit > Undo** or the keyboard shortcut for the 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. Additional help can be found 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, [LinearLayout](#) 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 manually.
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...**

It may say "no lines changed: code is already properly formatted".

7. Run the 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. 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 TextView, and another Button element, in that order, one below the other into the virtual device layout.

Adjust the UI Elements

1. 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 view needs 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

1. 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!



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.

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 parent view.
- 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
layout_width	match_parent (to fill the screen)
layout_height	match_parent (to fill the screen)
orientation	vertical

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
```

- Extract and name the remaining strings from the views as follows:

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

- 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

Similar to strings, it is a best practice to extract view dimensions from the main layout 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:

- Look at the dimens.xml resource file. There should be values for the default screen margins defined. For the dimensions of views, it is better not to use hard-coded values, because that prevents views from adjusting to the screen size.
- If necessary, change the layout_width of all elements inside the LinearLayout to "match_parent".

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.

- If necessary, change the layout_height of all elements inside the LinearLayout to "wrap_content".

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 screen density and user's font size preference. It is recommended you use this unit when specifying font sizes, so they will be adjusted for both the screen density and the user's preference.

```
    android:textSize="160sp"
```

2. Extract the text size of the `TextView` as a dimension resource named `count_text_size`, as follows:
 - i. Click the **Text** tab to show the XML code, if you haven't already done so.
 - ii. Place the cursor on "`160sp`".
 - iii. Press **Alt-Enter (Option-Enter on the Mac)**.
 - iv. Click **Extract dimension resource**.
 - v. Set the **Resource name** to `count_text_size`, and click **OK**. (If you make a mistake, you can undo the change with `Ctrl-Z`).
 - vi. 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`.
3. Change the `textStyle` of the `show_count` `TextView` to bold.

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

4. Change the text color in the `show_count` text view to the primary color of the theme. Look at the `colors.xml` resource file to see how they are defined.

The `colorPrimary` is one of the predefined theme base colors and is used for the app bar. For example, in a production app, you could 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 the buttons to the primary color of the theme.

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

6. Change the color of the text in both buttons to white. White is one of the colors provided as an Android Platform Resource. See [Accessing Resources](#).

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

7. Add a background color to the TextView.

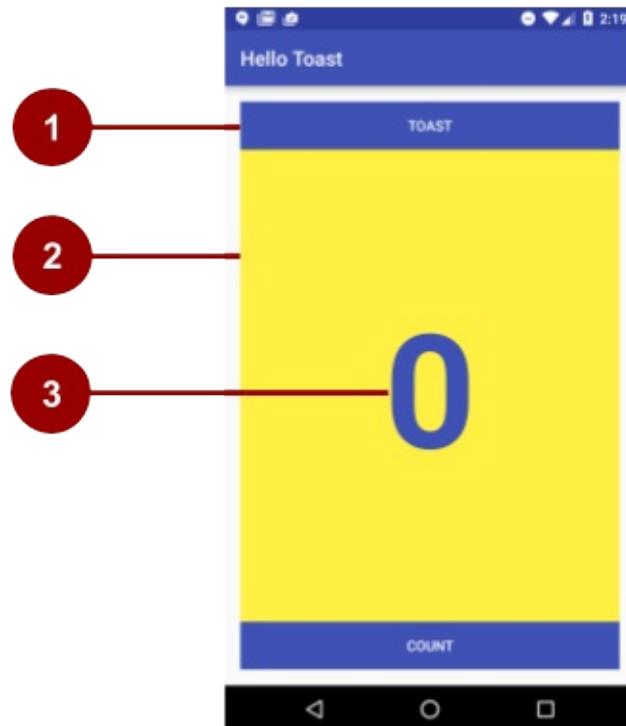
```
    android:background="#FFFF00"
```

8. In the Layout Editor (Text tab), place your mouse cursor over this color and press **Alt-Enter** (**Option-Enter** on the Mac).
9. Select **Choose color**, which brings up the color picker, pick a color you like, or go with the current yellow, then click **Choose**.
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.

```
<color name="myBackgroundColor">#FFF043</color>
```

3.5 Gravity and weight

Specifying gravity and weight properties gives you additional control over arranging views



and content in linear layouts.

1. The `android:layout_gravity` attribute specifies how a view is aligned within its *parent View*. Because the views match their parent in width, it is not necessary to set this explicitly. You can center a view that is narrow horizontally in its parent:

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

2. The `android:layout_weight` attribute indicates how much of the extra space in the LinearLayout will be allocated to the views that have this parameter set. If only one view has this attribute, it gets all the extra screen estate. For multiple views, the space is pro-rated. For example, if the buttons have a weight of 1 and the text view 2, totalling 4, the buttons get $\frac{1}{4}$ of the space each, and the textview half.
3. The `android:gravity` attribute specifies the alignment of the *content of a View within the View itself*. The counter is centered in its view with: `android:gravity="center"`

Do the following:

1. Center the text in a the `show_count` TextView horizontally and vertically:

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

2. Make the `show_count` TextView adjust to the size of the screen:

```
    android:layout_weight="2"
```

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>
</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="count_text_size">160sp</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="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/button_label_toast"
        android:background="@color/colorPrimary"
        android:textColor="@android:color/white" />

    <TextView
        android:id="@+id/show_count"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        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"
        android:layout_weight="2" />

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

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later chapters.

- 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, gravity,

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](#).

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 user clicks an on-screen button.
- Associate this method to the view, so this method is called 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 the `makeText()` factory method 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 common pattern that you can reuse the code you are going to write.

1. Get the context of the application.

Displaying a `Toast` message 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 current activity. `Context context`

```
= getApplicationContext();
```

When you are already in the context of the activity whose context you need, you can also use `this` as the shortcut to the context.

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

- o `Toast.LENGTH_LONG`
- o `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.

- o The context is the application context we got earlier.
- o The message is the string you want to display
- o The duration is one of the predefined constants `Toast.LENGTH_LONG` or `Toast.LENGTH_SHORT`.

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

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

- 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.

- o `R.` identifies the parameter as a resource.
- o `string` references the name of the XML file where the resources is defined.
- o `toast_message` is the name of the resource.

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

5. Display the toast.

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

6. 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) {
    // Create a toast show it.
    Toast toast = Toast.makeText(this, R.string.toast_message, Toast.LENGTH_LONG);
    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 private member variable `mCount` to track the count and start it at 0.
2. In `MainActivity.java`, add a private member variable `mShowCount` to get the reference of the `show_count` `TextView`.
3. In the `countUp()` method, increase the value of the `count` variable each time the button is clicked.
4. 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 type you expect, in this case `(TextView)`.

In order to get this resource only once, use a member variable and set it in `onCreate()`.

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

5. Set the text in the text view to the value of the `count` variable.

```

if (mShowCount != null)
    mShowCount.setText(Integer.toString(mCount));

```

6. 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 {  
    private int mCount = 0;  
    private TextView mShowCount;
```

in onCreate():

```
mShowCount = (TextView) findViewById(R.id.show_count);  
  
countUp Method:  
    public void countUp(View view) {  
        mCount++;  
        if (mShowCount != null)  
            mShowCount.setText(Integer.toString(mCount));  
    }
```

Resources:

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

Solution code

Android Studio project: [HelloToast](#)

Coding challenge

Note: All coding challenges are optional and are not a prerequisite for later chapters.

Challenge: Even a simple app like Hello Toast can be the foundation of many scoring or product ordering apps. Write one app that 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.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Layouts, Views, and Resources](#)

Learn more

- 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](#)

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](#)
- [Architectural Patterns \(overview\)](#)

1.2B: Using Layouts

Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App Overview](#)
- [Task 1: Change the layout to RelativeLayout](#)
- [Task 2: Change the layout to ConstraintLayout](#)
- [Task 3: Create layout variants](#)
- [Coding Challenge](#)
- [Summary](#)
- [Related concepts](#)
- [Learn more](#)

When you start an Android Studio project, the template you choose provides a basic layout with views. As you learned in a previous practical, you can line views up quickly and easily in a layout using [LinearLayout](#), which is a view group that aligns child views within it horizontally or vertically.

This practical explores two other layout view groups:

- [RelativeLayout](#): A group of child views in which each view is positioned and aligned relative to other views within the view group. Positions of the child views are described in relation to each other or to the parent view group.
- [ConstraintLayout](#): A layout similar to RelativeLayout but more flexible. It groups child views using anchor points (a connection to another view), edges, and guidelines to control how views are positioned relative to other elements in the layout.
ConstraintLayout was designed to make it easy to drag and drop views in the layout editor of Android Studio

What you should already KNOW

From the previous practicals, you should be able to:

- Create a Hello World app with Android Studio.
- Run an app on the emulator or a device.
- Implement a TextView in a layout for an app.
- Create and using string resources.

- Convert layout dimensions to resources.

What you will LEARN

You will learn to:

- Use the layout editor in Android Studio
- Position views within a RelativeLayout
- Position views within a ConstraintLayout
- Create variants of the layout for landscape orientation and larger displays

What you will DO

In this practical you will:

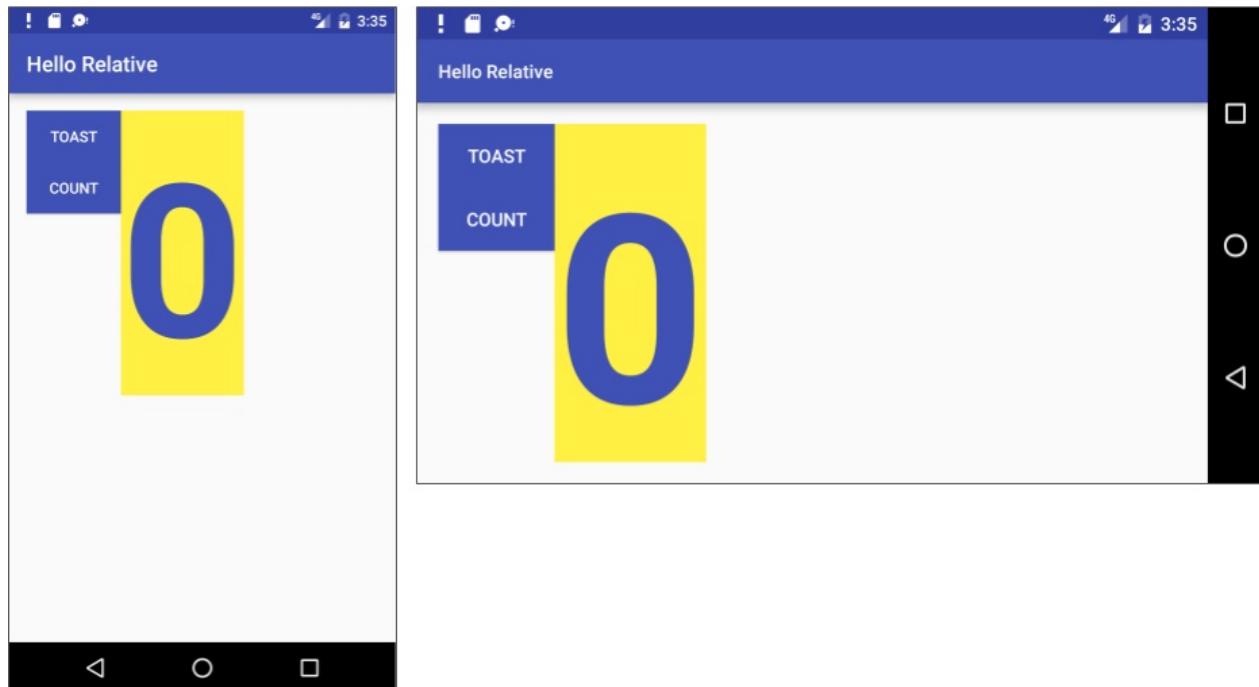
- Experiment with using RelativeLayout and ConstraintLayout.
- Copy and refactor the Hello Toast app to create the Hello Relative app.
- Change the root view group in the main layout to be RelativeLayout.
- Rearrange the views in the main layout to be relative to each other.
- Copy and refactor the Hello Relative app to create Hello Constraint.
- Change the root view group in the main layout to be ConstraintLayout.
- Modify the layout to add constraints to the views.
- Modify the views for layout variants for landscape orientation and larger displays.

App Overview

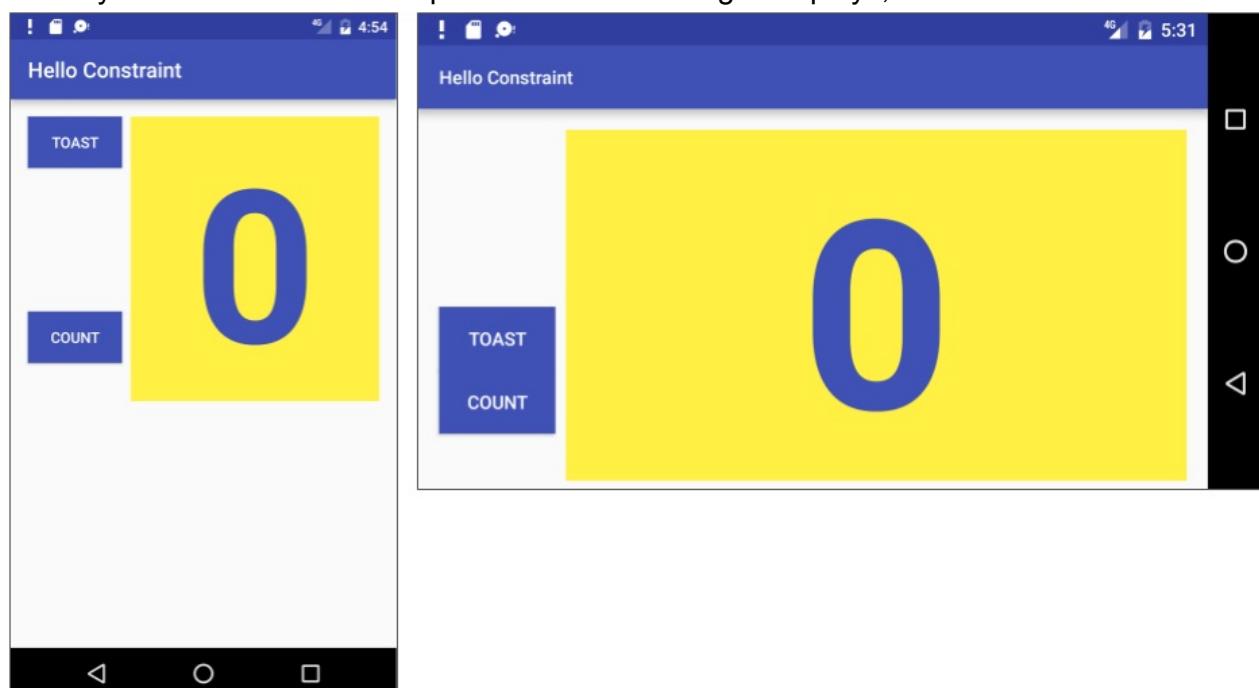
The Hello Toast app in a previous practical uses a LinearLayout to arrange the views in the activity layout, as shown in the figure below.



In order to practice using the layout editor, you will copy the Hello Toast app and call the new copy Hello Relative, in order to experiment with a RelativeLayout. You will use the layout editor to arrange the views in a different UI layout as shown below.



Finally, you will make another copy of the app and call it Hello Constraint, and replace LinearLayout with ConstraintLayout. ConstraintLayout offers more visual aids and positioning features in the layout editor. You will create an entirely different UI layout, and also layout variants for landscape orientation and larger displays, as shown below.



Android Studio project: [HelloToast](#)

Task 1: Change the layout to RelativeLayout

A [RelativeLayout](#) is a view grouping in which each view is positioned and aligned relative to other views within the group. In this task, you will investigate using RelativeLayout.

1.1 Copy and refactor the Hello Toast app

1. Copy the **HelloToast** project folder, rename it to **HelloRelative**, 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 **Hello Relative** (with a space) as the app's name.

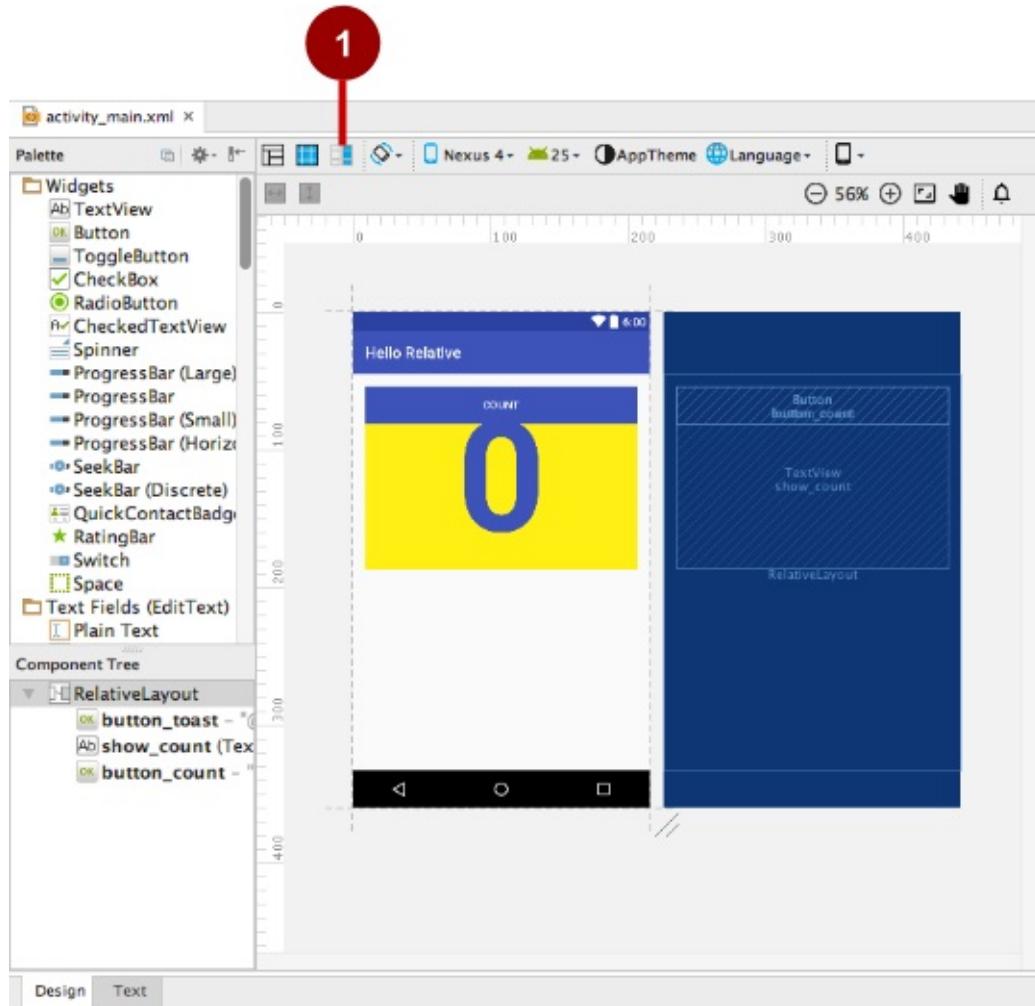
1.2 Change LinearLayout to RelativeLayout

1. Open the **activity_main.xml** layout file, and click the **Text** tab at the bottom of the editing pane to see the XML code.
2. Change the `<LinearLayout` at the top to `<RelativeLayout` so that the statement looks like this:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
```
3. Scroll down to make sure that the ending tag `</LinearLayout>` has also changed to `</RelativeLayout>`; if it hasn't, change it manually.

1.3 Rearrange views with the Design tab

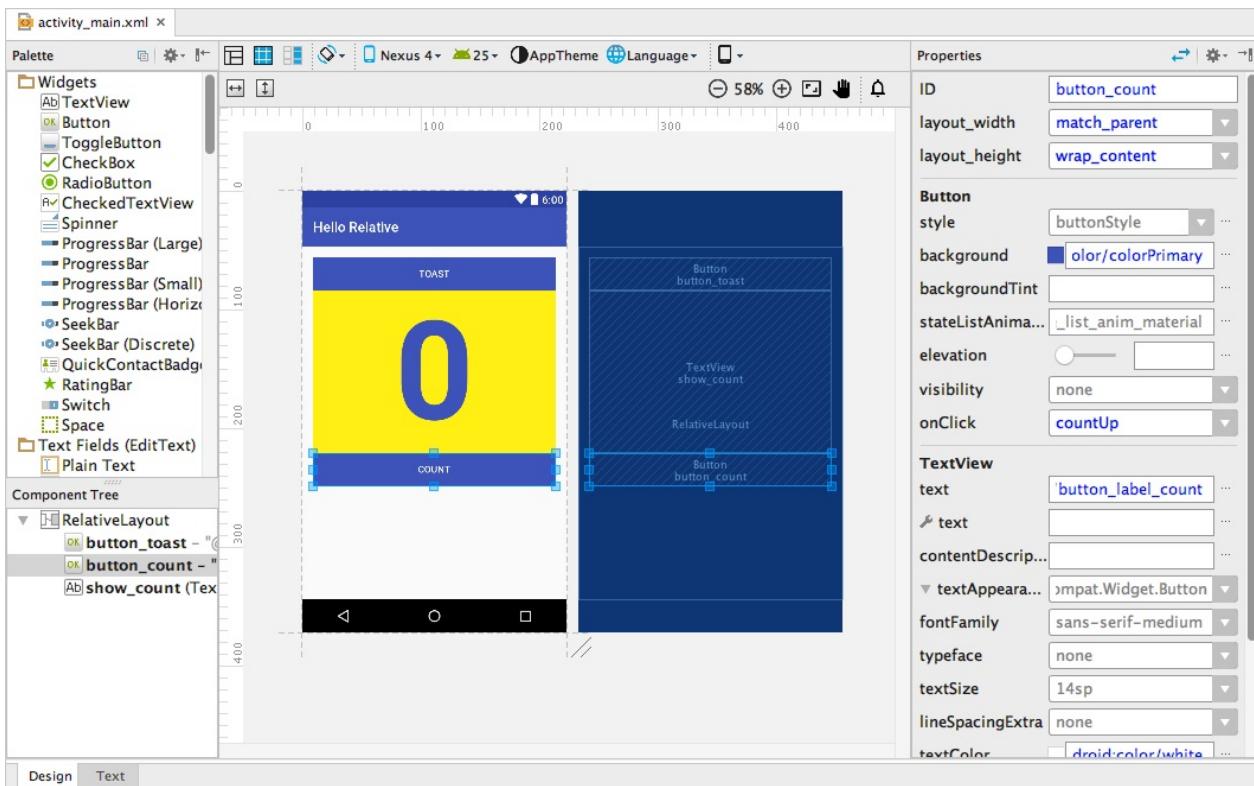
1. Click the **Design** tab at the bottom of the editing pane.
2. The editing pane should now look like the figure below, with the layout design and its blueprint. If you see only a layout design, or only a blueprint, click the **Show Design + Blueprint** button (#1 in the figure below).



3. With the change to `RelativeLayout`, the layout editor also changed some of the view attributes. For example:

- The `button_count` view for the `COUNT` button is overlaying the `button_toast` view for the `TOAST` button, which is why you can't see the `TOAST` button. However, in the blueprint, you can see that the two buttons are occupying the same space.
- The top part of the `show_count` view (showing `0`) is also overlaying the `COUNT` and `TOAST` buttons.

4. Drag the `button_count` view (for the `COUNT` button) to an area below the `show_count` view (showing `0`), and then drag it up to the bottom of the `show_count` view until it snaps into place as shown below. Also drag the `show_count` view so that the top of the view snaps to the bottom of the `button_toast` view for the `TOAST` button.



Tip: When selecting a view in the layout, its properties appear in the Properties pane on the right side of the editing pane. These properties correspond to XML attributes in the XML code for the layout, which you will examine in the next step.

1.4 Examine the XML code in the Text tab

Follow these steps to see how the app looks:

1. Run the app. The app works the same way as before. The only difference is that the layout uses a `RelativeLayout` to position the elements, rather than a `LinearLayout`. In the next task, you will change the layout of the UI.
2. Change the device or emulator orientation to landscape. Note that the `button_count` view disappears because the screen layout does not accommodate the landscape orientation. You will fix this problem in a subsequent task in this practical.
3. Click the **Text** tab at the bottom of the editing pane.
4. Examine the changes to the XML code in the editing pane as a result of changing `LinearLayout` to `RelativeLayout`. Start by examining the second Button (`button_count`):

```
<Button  
    android:id="@+id/button_count"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
    android:background="@color/colorPrimary"  
    android:onClick="countUp"  
    android:text="@string/button_label_count"  
    android:textColor="@android:color/white"  
    android:layout_below="@+id/show_count"  
    android:layout_centerHorizontal="true" />
```

Two new XML attributes were automatically added by the layout editor after you moved the `Button` (`button_count`) in the layout:

```
    android:layout_below="@+id/show_count"  
    android:layout_centerHorizontal="true"
```

The `android:layout_below` attribute places the `button_count` view directly below the `show_count` view. This attribute is one of several attributes for positioning views within a `RelativeLayout` — you place views in relation to the XML code for `show_count` view, which you also moved in the layout editor, is now in a position below the two buttons in the Text view. This is due to the change from `LinearLayout` to `RelativeLayout`. The `show_count` view also now includes the following attributes, as a result of moving the view in the layout editor:

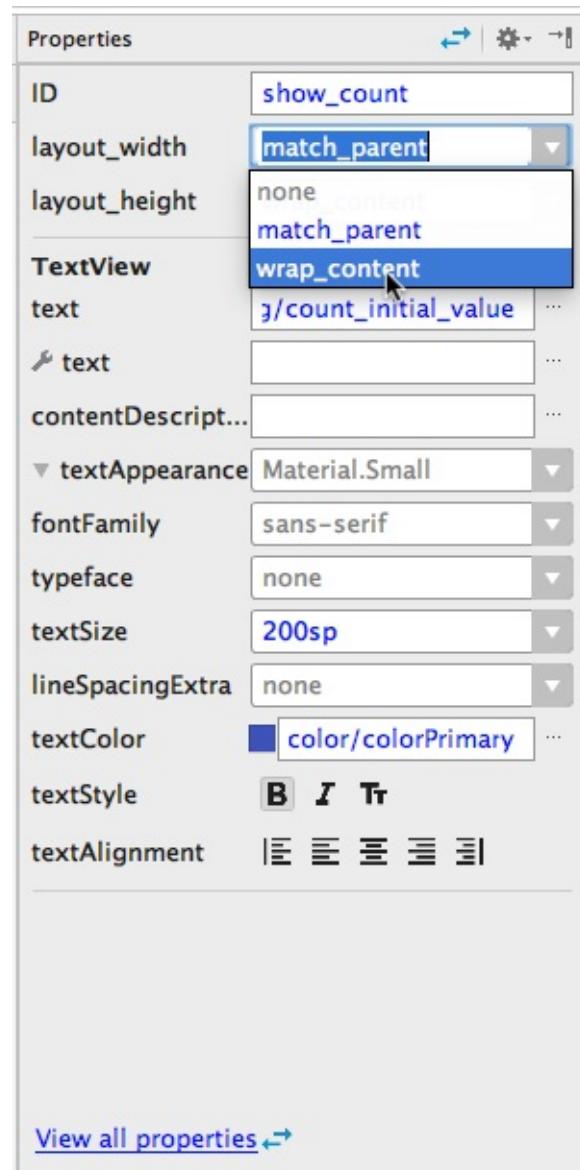
```
    android:layout_below="@+id/button_toast"  
    android:layout_alignParentLeft="true"  
    android:layout_alignParentStart="true"
```

The `android:layout_alignParentLeft` aligns the view to the left side of the `RelativeLayout` parent view group. While this attribute by itself is enough to align the view to the left side, you may want the view to align to the right side *if* the app is running on a device that is using a right-to-left language. Thus, the `android:layout_alignParentStart` attribute makes the "start" edge of this view match the start edge of the parent. The *start* is the left edge of the screen if the preference is left-to-right, or it is the right edge of the screen if the preference is right-to-left.

1.5 Rearrange elements in the RelativeLayout

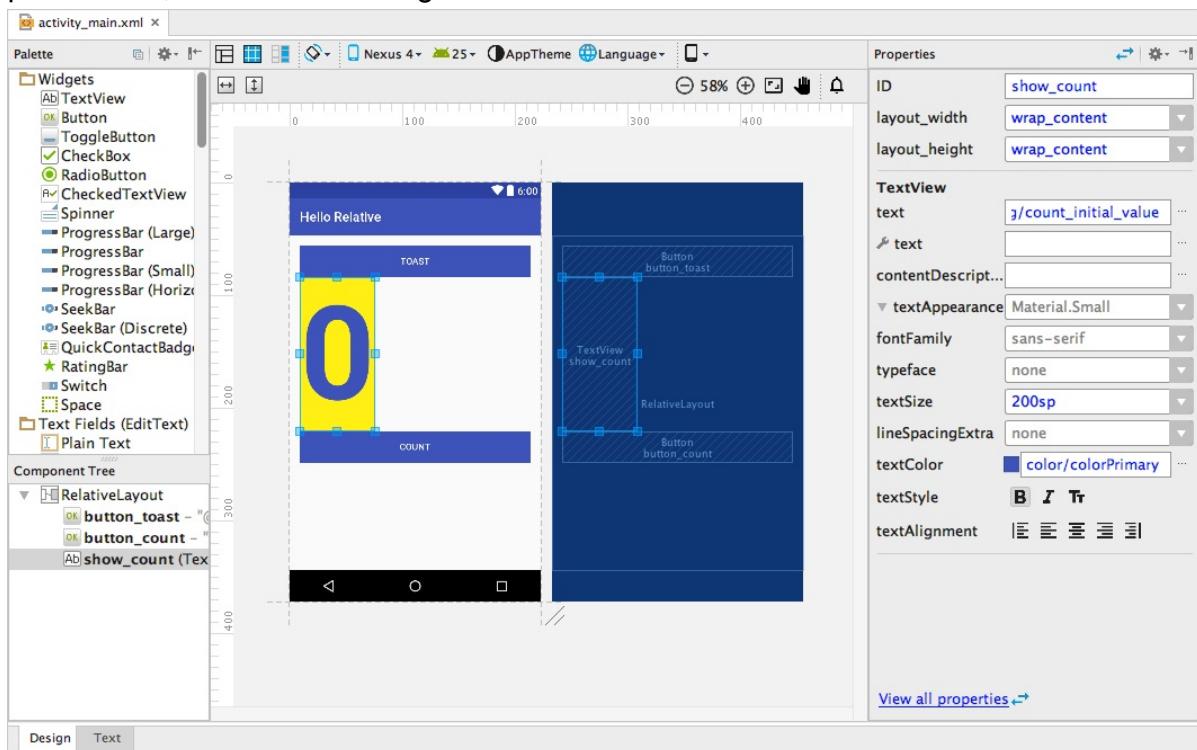
1. To experiment more with `RelativeLayout`, select the `activity_main.xml` layout again for editing (if it's not already selected), and click the **Design** tab at the bottom of the editing pane.

2. Select the `show_count` view in the layout or the Component Tree, and change its `layout_width` in the Properties pane on the right side of the window to `wrap_content`

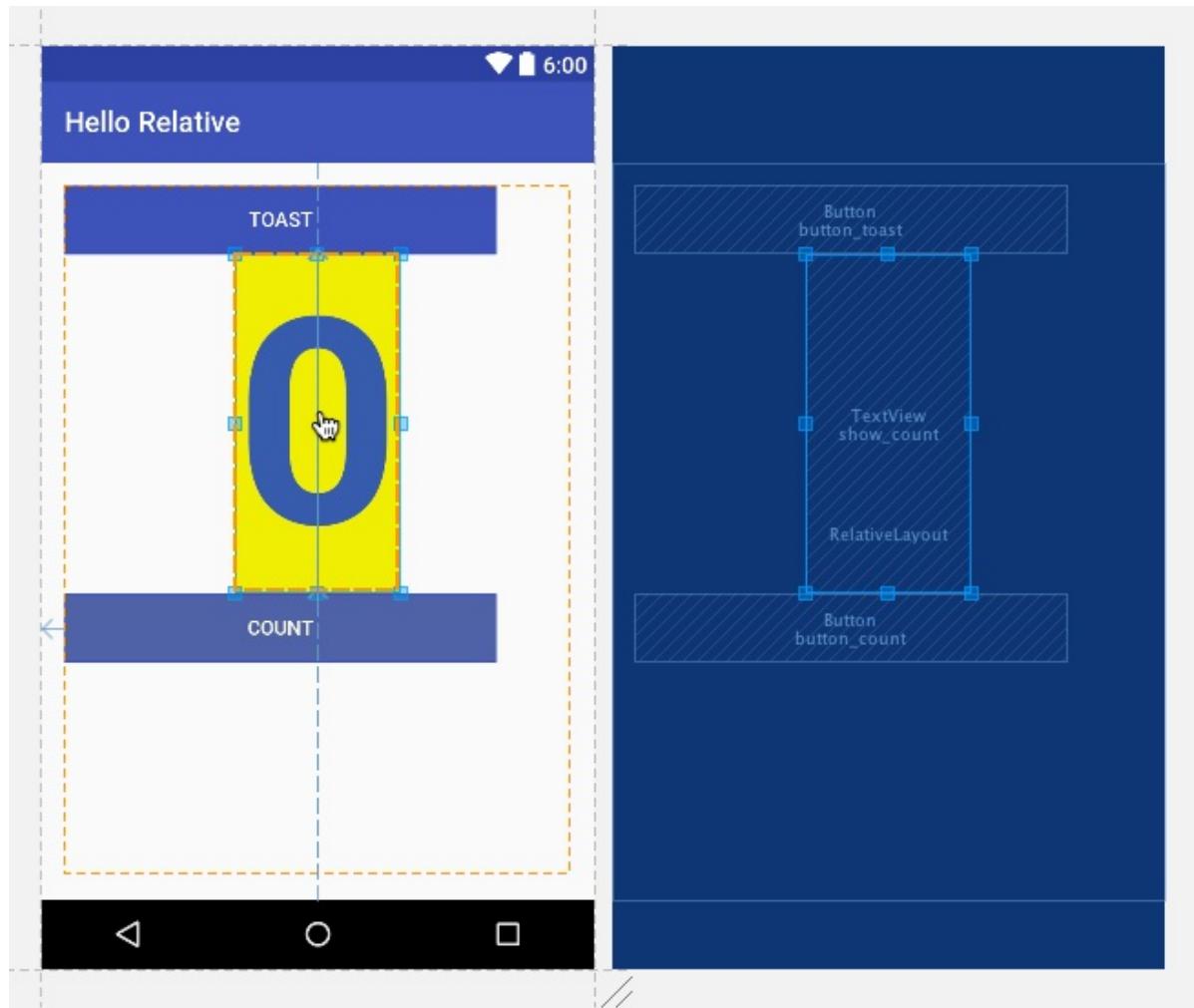


as shown in the figure below.

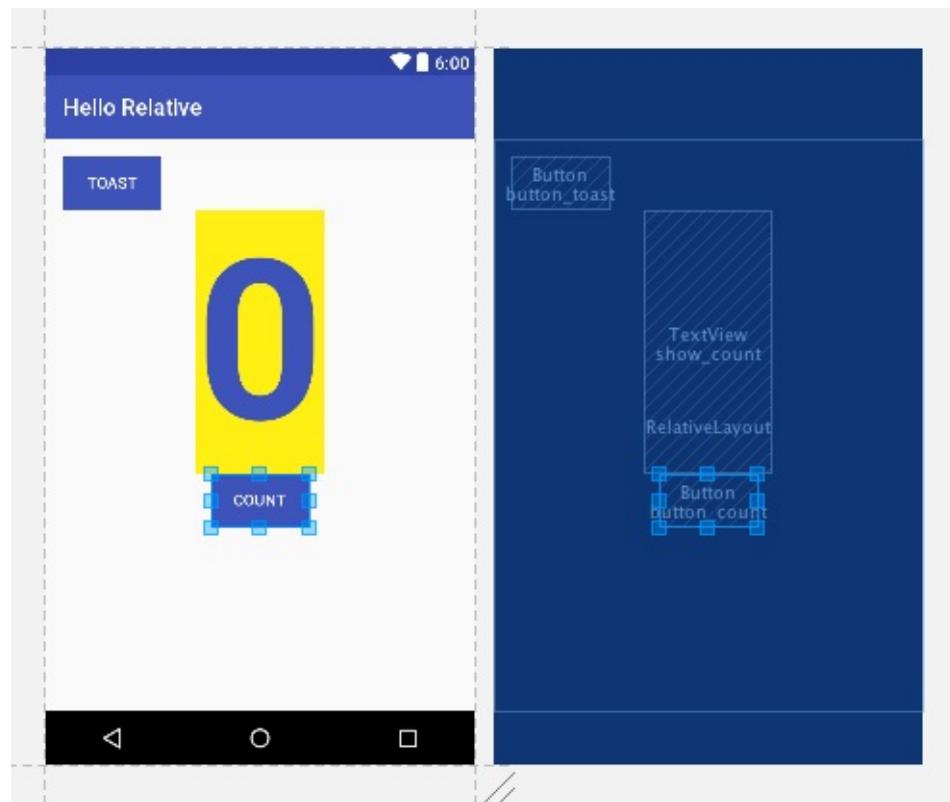
The layout editor displays a thinner `show_count` view aligned to the left side of the parent view, as shown in the figure below.



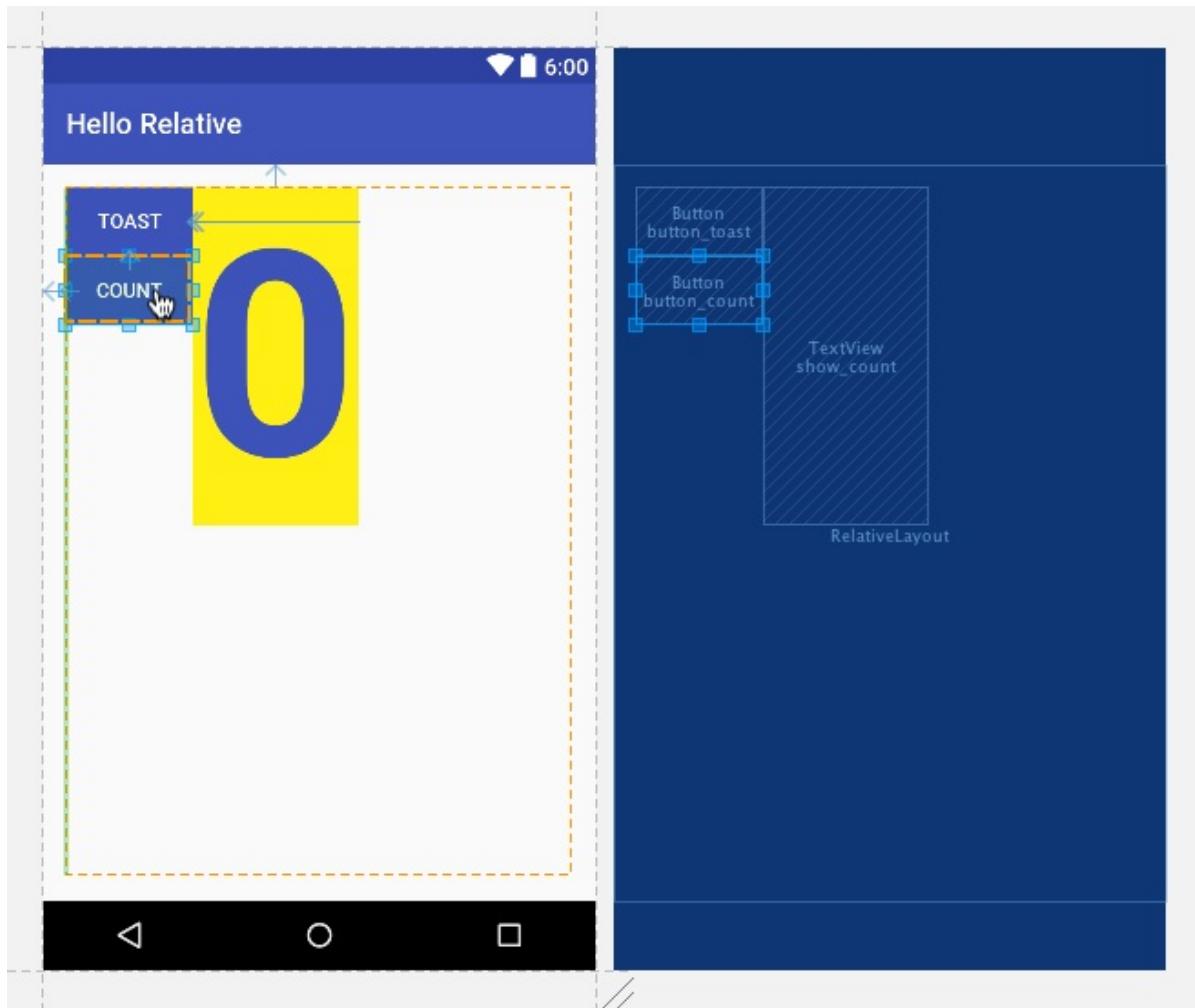
3. Drag the `show_count` view horizontally to the center of the layout. As you drag the view, a center guide appears — the view's center should snap into place with the guide as shown below.



4. Select the `button_toast` view and change its `layout_width` to **wrap_content** in the Properties pane, and then change the `layout_width` of the `button_count` view to **wrap_content**. The layout should now look like the figure below.



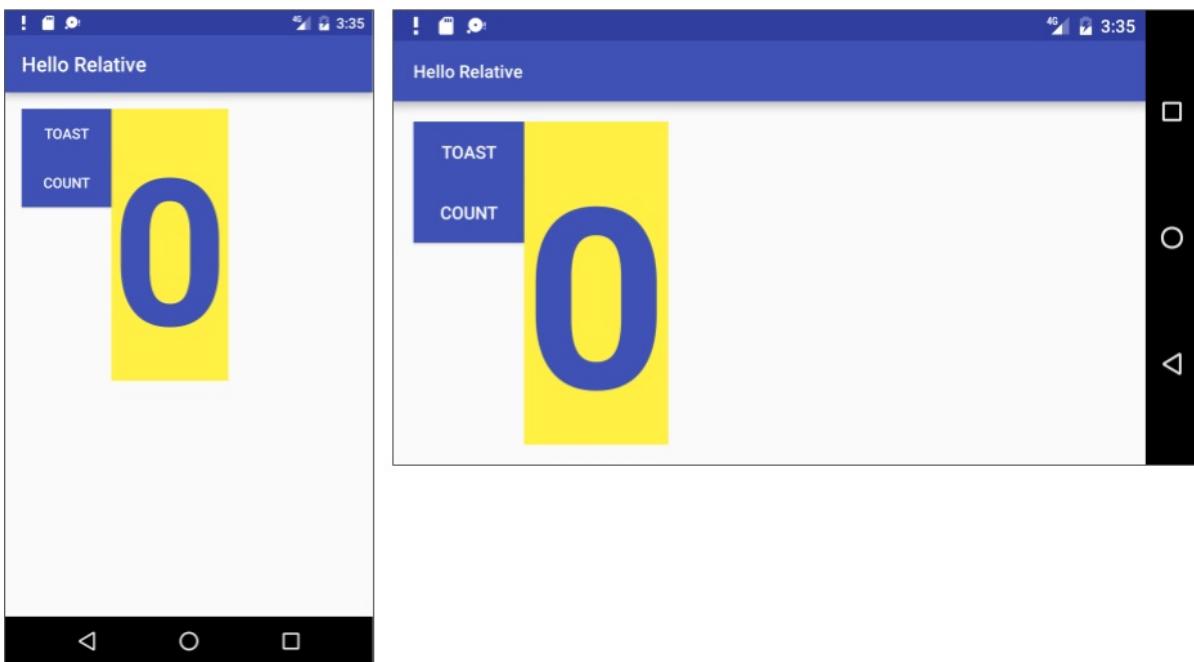
5. Drag the `button_count` view up to just below the `button_toast` view so that it snaps to the bottom of the `button_toast` view, and drag the `show_count` view up next to the right edge of the `button_toast` view so that it snaps to the right edge of the button. The layout should now look like the figure below:



6. Click the **Text** tab at the bottom of the editing pane, and examine the changes to the XML code as a result of moving the views in the layout:
 - The `show_count` view now uses the following attributes to position it to the *right* of and the *end* of the `button_toast` view:


```
android:layout_toRightOf="@+id/button_toast"
        android:layout_toEndOf="@+id/button_toast"
```
 - The `button_count` view now uses the following attributes to position it *below* the `button_toast` view:


```
android:layout_below="@+id/button_toast"
```
7. Run the app. The app works the same way as before (since we didn't change any Java code). However, the layout is different, as shown in the figure below. Change the device or emulator orientation to landscape to see that the new layout works for both orientations.



Tip: To learn more about how to position views in a RelativeLayout, see "[Positioning Views](#)" in the "Relative Layout" topic of the API Guide.

Solution code: Android Studio project: [HelloRelative](#)

Task 2: Change the layout to ConstraintLayout

ConstraintLayout is a view group available in the Constraint Layout library, which is included with Android Studio 2.2 and higher. The constraint-based layout lets a developer build complex layouts without having to nest view groups, which can improve the performance of the app. It is built into the layout editor, so that the constraining tools are accessible from the **Design** tab without having to edit the XML by hand.

In this task you will copy and refactor the Hello Toast app to create the Hello Constraint app. You will then change the root `LinearLayout` view group in the main layout to be `ConstraintLayout`. After changing the root view group, you will rearrange the views in the main layout to have constraints that govern their appearance.

2.1 Copy and refactor the Hello Toast app

1. Copy the **HelloToast** project folder, rename it to **HelloConstraint**, 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 **Hello Constraint** (with a space) as the app's name.

2.2 Add ConstraintLayout to your project

Check to be sure that ConstraintLayout is available in your project:

1. In Android Studio, choose **Tools > Android > SDK Manager**.
2. In the left pane, click **Android SDK**.
3. In the right pane, click the **SDK Tools** tab at the top of the pane.
4. Expand **Support Repository** and see if ConstraintLayout for Android and Solver for ConstraintLayout are already checked.
 - If "Installed" appears in the Status column, you're all set. Click **Cancel**.
 - If "Not installed" appears, or "Update" appears:
 - i. Click the checkbox next to ConstraintLayout for Android and Solver for ConstraintLayout. A download icon should appear next to each checkbox.
 - ii. Click one of the following:
 - **Apply** to start installing the components and remain in SDK Manager to make other changes.
 - **OK** to install the components.
 - iii. After installing the components (and making other changes if needed), click **Finish** to finish using the SDK Manager.

2.3 Convert a layout to ConstraintLayout

Android Studio has a built-in converter to help you convert a layout to `ConstraintLayout`.

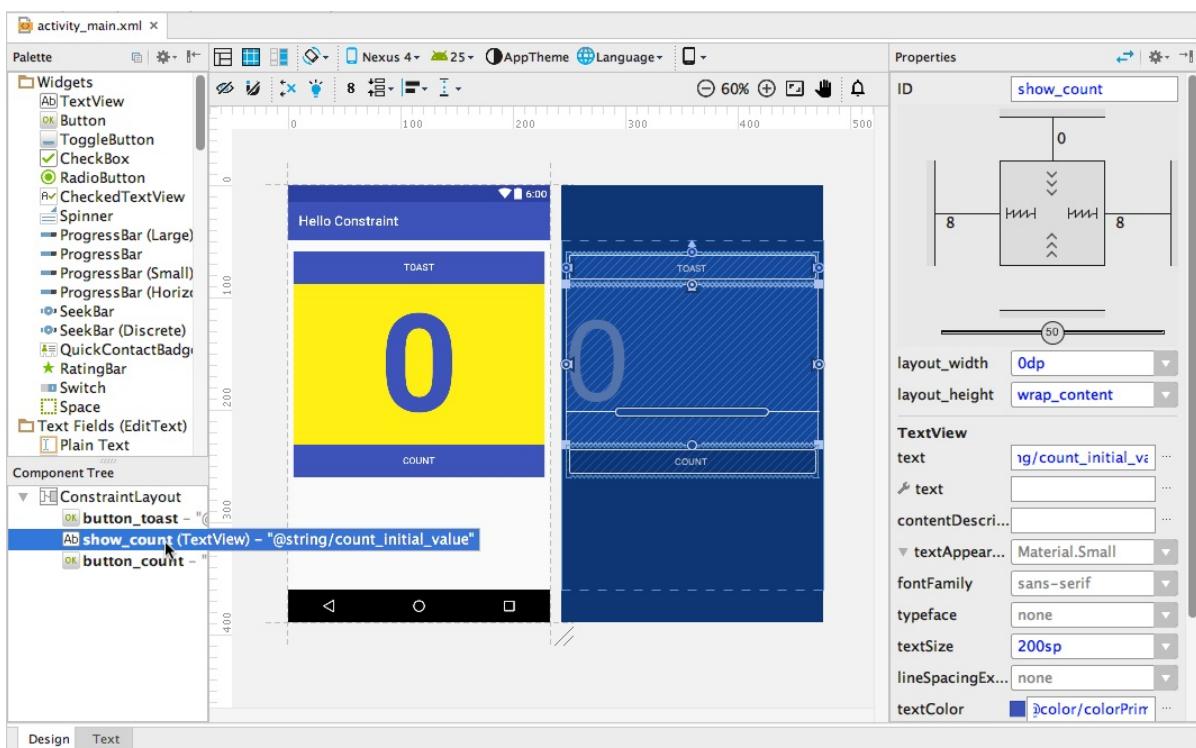
Follow these steps:

1. Open the layout file (**activity_main.xml**) in Android Studio and click the **Design** tab at the bottom of the editor window.
2. In the Component Tree window, right-click **LinearLayout** and then choose **Convert layout to ConstraintLayout** from the context menu.
3. The converter displays an alert with two checkboxes already checked. Don't uncheck them—make sure both options remain *checked*:
 - i. **Flatten Layout Hierarchy**: This option removes all other nested layouts in the hierarchy. The result is a single, flat layout, which may be more efficient for these purposes.
 - ii. **Don't flatten layouts referenced from other files**: If a particular layout defines an `android:id` attribute that is referenced in your Java code, you may not want to flatten that layout because your code may no longer work. However, in HelloConstraint, you don't have an `android:id` attribute for a layout, only for views.

4. In the Add Project Dependency alert, click **OK** to add the constraint-layout library. Android Studio automatically copies the appropriate dependency to your project's **build.gradle (Module: app)** file and syncs the change for you. The layout editor reappears with `ConstraintLayout` as the root view group.

Note: If the layout editor has a problem with the change, you see a Rendering Problems warning. Click **build** in the message `Tip: Try to build the project.` This will re-sync your project's **build.gradle (Module: app)** file with the new dependency.

5. The layout editor's Component Tree pane now shows `ConstraintLayout` as the root view group for the layout with the other views beneath it, as shown in the figure below. Click the **show_count** view in the Component Tree pane. The `show_count` view is also selected in the blueprint, and its properties appear in the Properties pane on the right side.



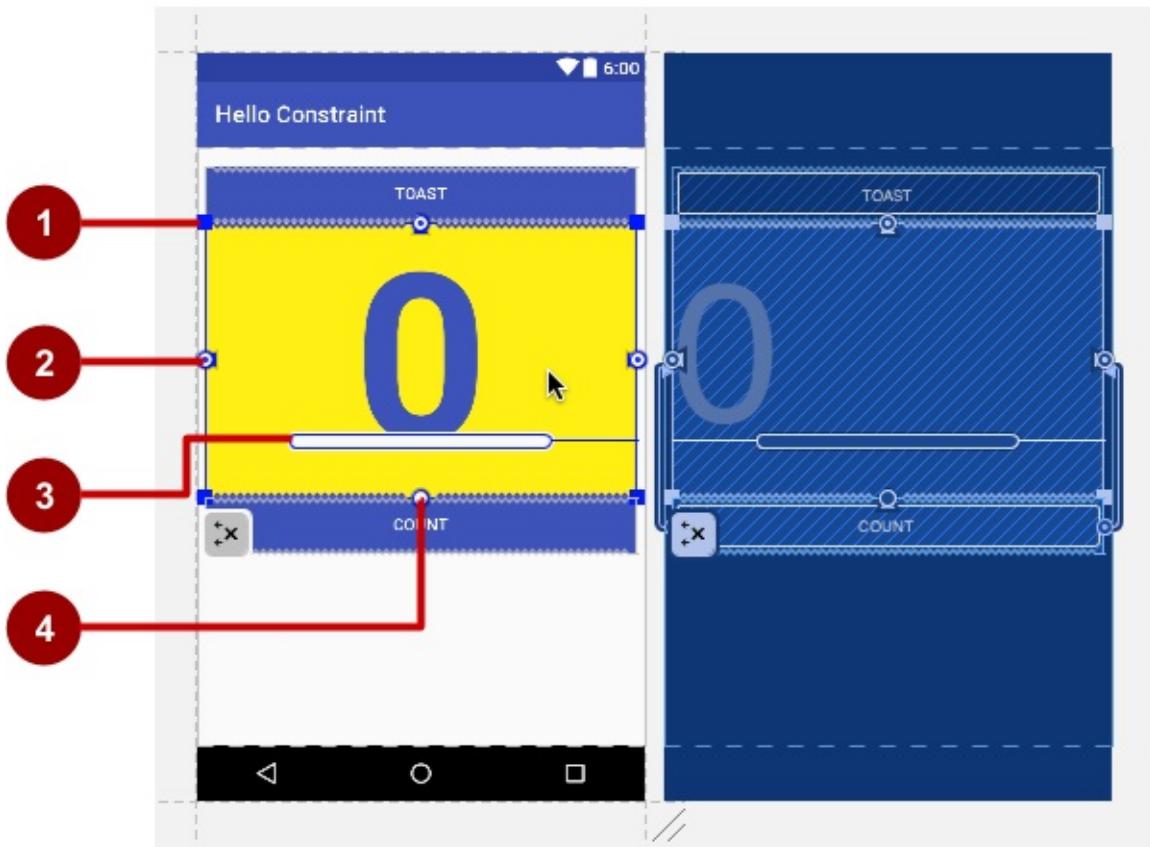
2.4 Explore the layout editor

The layout editor offers more features in the **Design** tab when you use a `ConstraintLayout`, including more visual layout tools and a second row of icons for more tools.

The visual layout and blueprint offer handles for defining constraints. A *constraint* is a connection or alignment to another view, to the parent layout, or to an invisible guideline. Follow these steps to explore the constraints that Android Studio created when you converted the `LinearLayout` to `ConstraintLayout`:

1. Click the `show_count` view in the Component Tree pane.
2. Hover the cursor over the `show_count` view in the layout, as shown in the figure below.

Each constraint appears as a line extending from a circular handle. Each view has a circular constraint handle in the middle of each side. After selecting a view in the Component Tree pane or clicking on it in the layout, the view also shows resizing handles on each corner.



In the above figure:

1. **Resizing handle.**
2. **Constraint line and handle.** In the figure, the constraint aligns the left side of the `show_count` view to the left side of the `button_toast` button.
3. **Baseline handle.** The baseline handle aligns the text baseline of a view to the text baseline of another view.
4. **Constraint handle** without a constraint line.

The layout editor also offers a row of buttons that let you configure the appearance of the layout:



In the figure above:

1. **Design, Blueprint, and Both:** Click the **Design** icon (first icon) to display a color preview of your layout. Click the **Blueprint** icon (middle icon) to show only outlines for each view. You can see *both* views side by side by clicking the third icon.

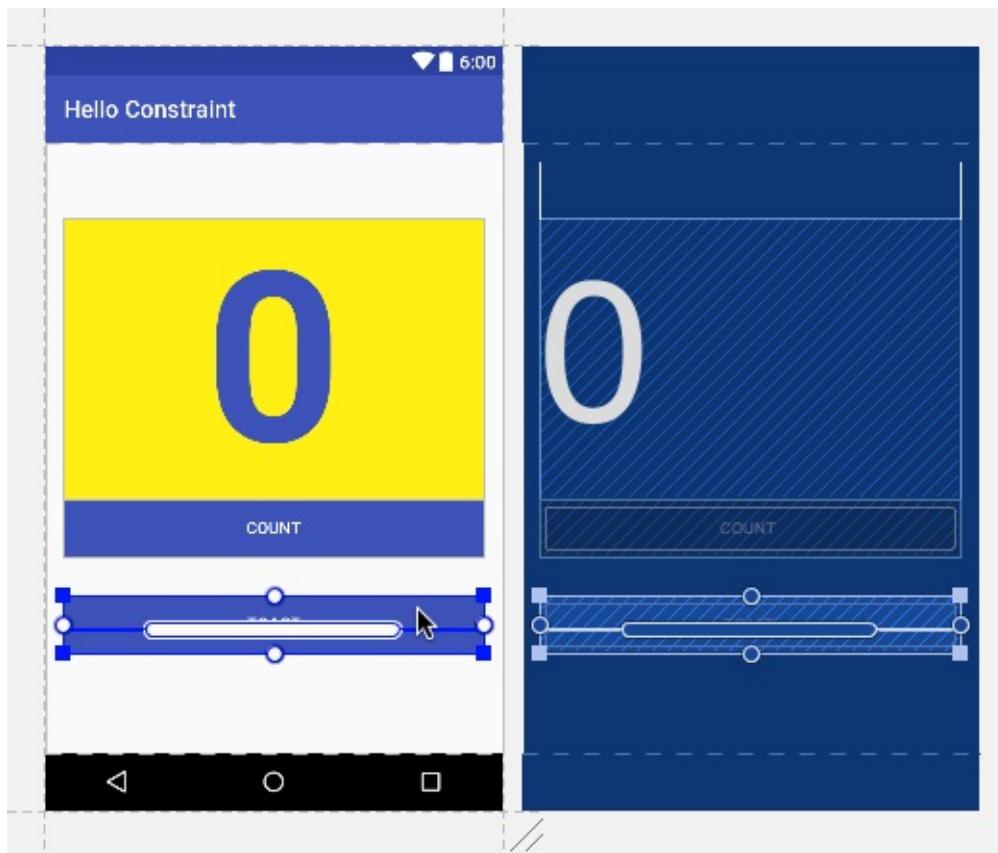
2. **Screen orientation:** Click to rotate the device between landscape and portrait.
3. **Device type and size:** Select the device type (phone/tablet, Android TV, or Android Wear) and screen configuration (size and density).
4. **API version:** Select the version of Android on which to preview the layout.
5. **App theme:** Select which UI theme to apply to the preview.
6. **Language:** Select the language to show for your UI strings. This list displays only the languages available in the string resources.
7. **Layout Variants:** Switch to one of the alternative layouts for this file, or create a new one.

Tip: To learn more about using the layout editor, see [Build a UI with Layout Editor](#). To learn more about how to build a layout with ConstraintLayout, see [Build a Responsive UI with ConstraintLayout](#).

2.5 Clear constraints

Android Studio automatically infers the constraints for layout elements when you convert a layout to use `ConstraintLayout`. However, the guesses may not be what you want. Follow these steps to clear the constraints in order to freely position the elements in the layout:

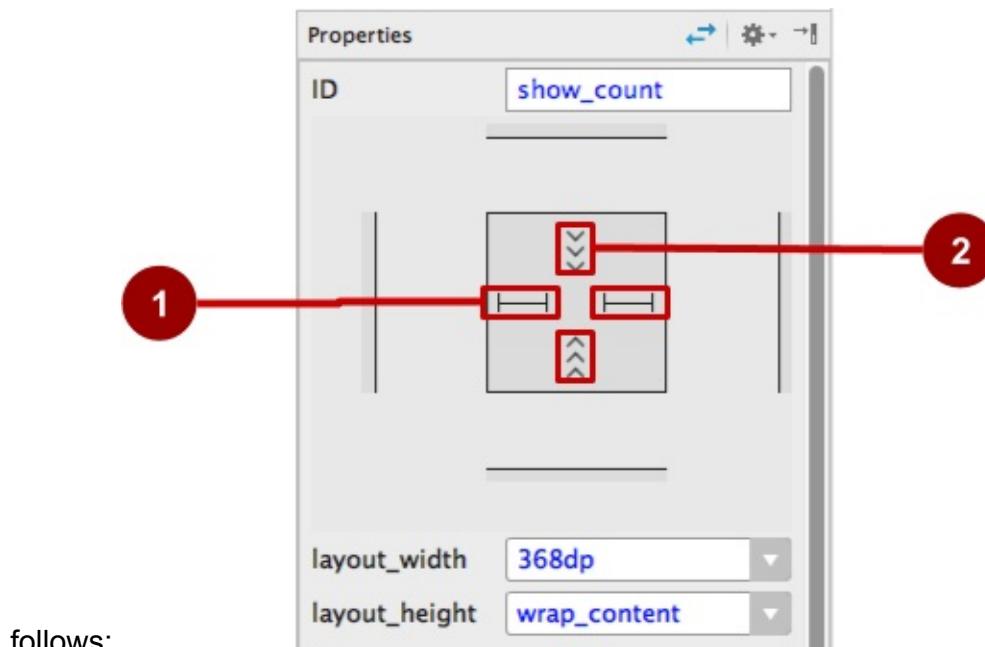
1. Right-click (or Control-click) **ConstraintLayout** in the Component Tree pane, and choose **Clear All Constraints**.
- Tip:** You can also delete a single constraint line by hovering the cursor over the constraint handle until a red circle appears, and then clicking the handle. The **Clear All Constraints** command is faster for removing all constraints.
2. With constraints removed, you can now move the views on the layout freely. Drag the `button_toast` view down to any position below the `button_count` view, so that the yellow `show_count` view is at the top, as shown in the figure below.



2.6 Resize a view

The layout editor offers resizing handles on all four corners of a view to resize the view quickly. You can drag the handles on each corner of the view to resize it, but doing so hard-codes the width and height dimensions, which you should avoid for most views because hard-coded view dimensions cannot adapt to different content and screen sizes.

Instead, use the Properties pane on the right side of the layout editor to select a sizing mode that doesn't use hard-coded dimensions. The Properties pane includes a square sizing panel at the top. The symbols inside the square represent the height and width settings as



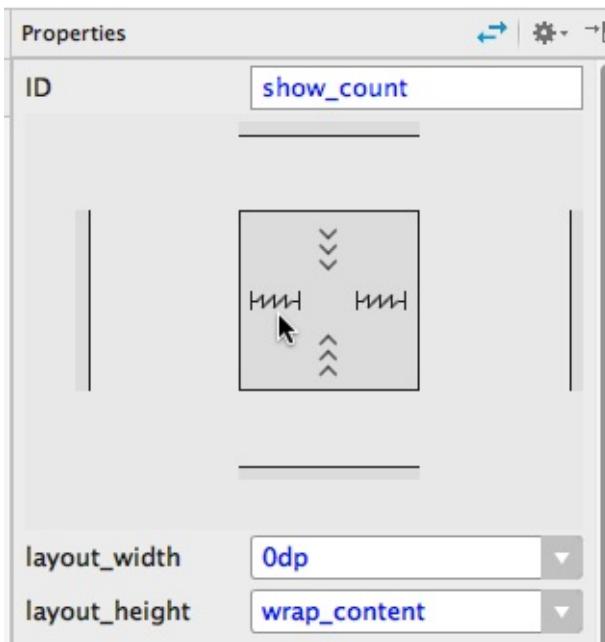
follows:

In the above figure:

1. **Horizontal view size control.** The horizontal size control, which appears in two segments on the left and right sides of the square, specifies the `layout_width`. The straight lines indicate that the dimension is fixed and set in the `layout_width` property below the square.
2. **Vertical view size control.** The vertical size control, which appears in two segments on the top and bottom sides of the square, specifies the `layout_height` property. The angles indicate that this size control is set to `wrap_content`, which means the view will expand exactly as needed to fit its contents.

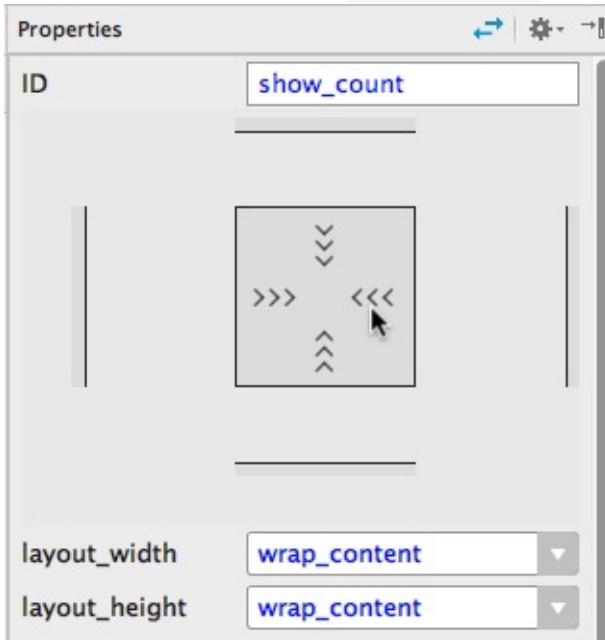
Follow these steps to resize the `show_count` view:

1. Click the `show_count` view in the Component Tree pane.
2. Click the horizontal view size control in the Properties pane. The straight lines change to spring coils, as shown in the figure below, which represents "any size". The `layout_width` property is set to zero because there is no set dimension, but the view can expand as much as possible to meet constraints and margin settings.

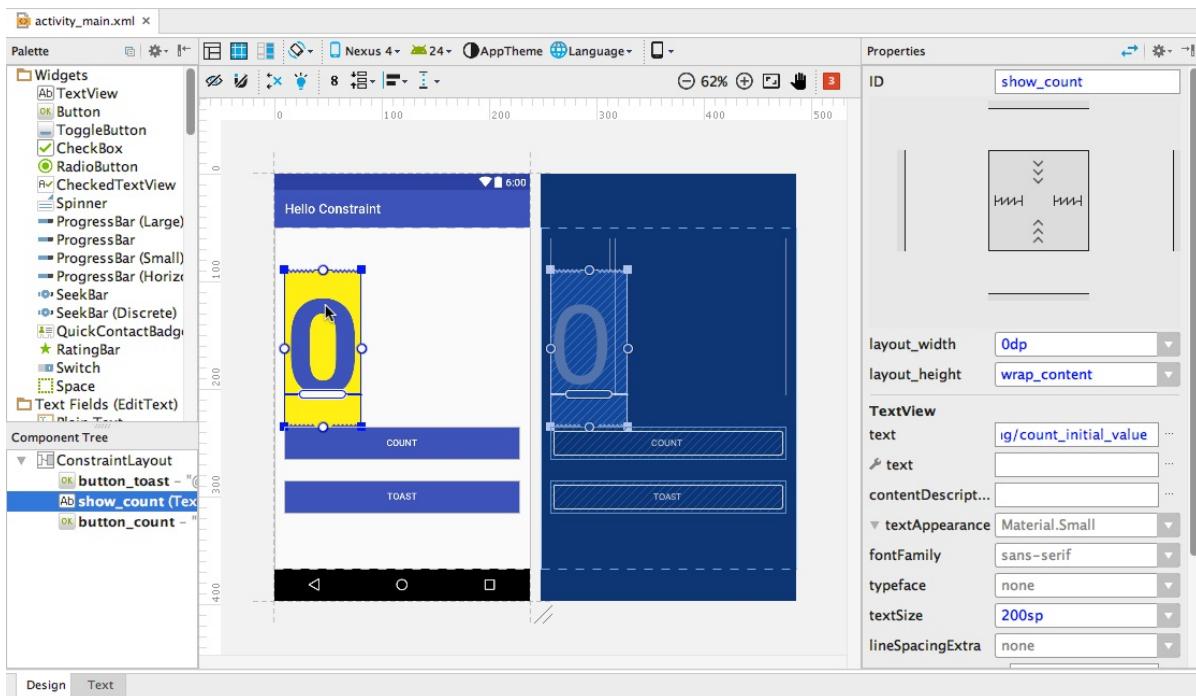


You will use this setting to anchor the size of the view to constraints, but first, continue to experiment with settings.

- Click the horizontal view size control again (either left or right side), just to see what other choices you have. The spring coils change to angles, as shown in the figure below, indicating that the `layout_width` is set to `wrap_content`.



- Click the horizontal view size control again, and it toggles back to the straight lines, indicating a fixed dimension. Click it again so that the lines change to spring coils, as shown in the figure below, which represents "any size".



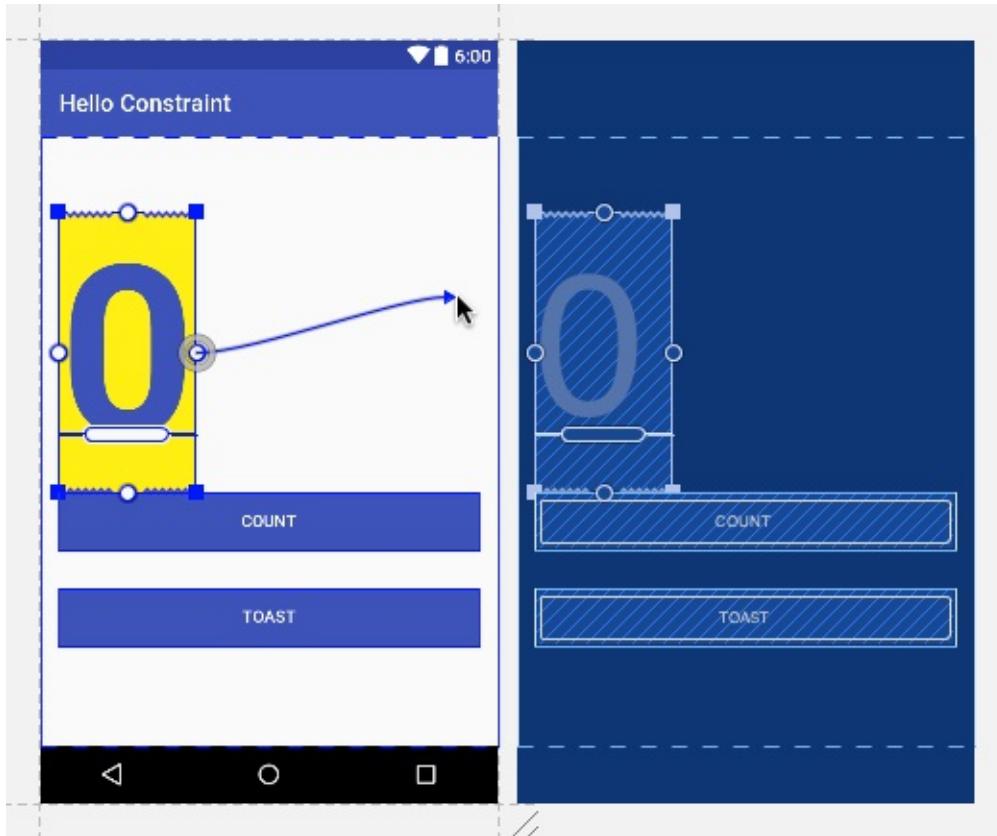
2.7 Add constraints to views

You will add a constraint to the `show_count` view so that it stretches to the right edge of the layout, and another constraint so that the view is positioned just below the top edge of the layout. Since the view was set to "any size" in the previous step, the view will expand as needed to match the constraints.

You will also move the two buttons into position on the left side of the `show_count` view, constrain the `button_toast` button to the top and left edges of the layout, and constrain the `button_count` button so that its text baseline matches the text baseline of the `show_count` view.

1. To create a right-side constraint for the `show_count` view, click the view in the layout, and then hover over the view to see its constraint handles. Click-and-hold the constraint handle on the right side of the view, and drag the constraint line that appears to the right edge of the layout, as shown in the figure below.

1. To create a right-side constraint for the `show_count` view, click the view in the layout, and then hover over the view to see its constraint handles. Click-and-hold the constraint handle on the right side of the view, and drag the constraint line that appears to the right edge of the layout, as shown in the figure below.



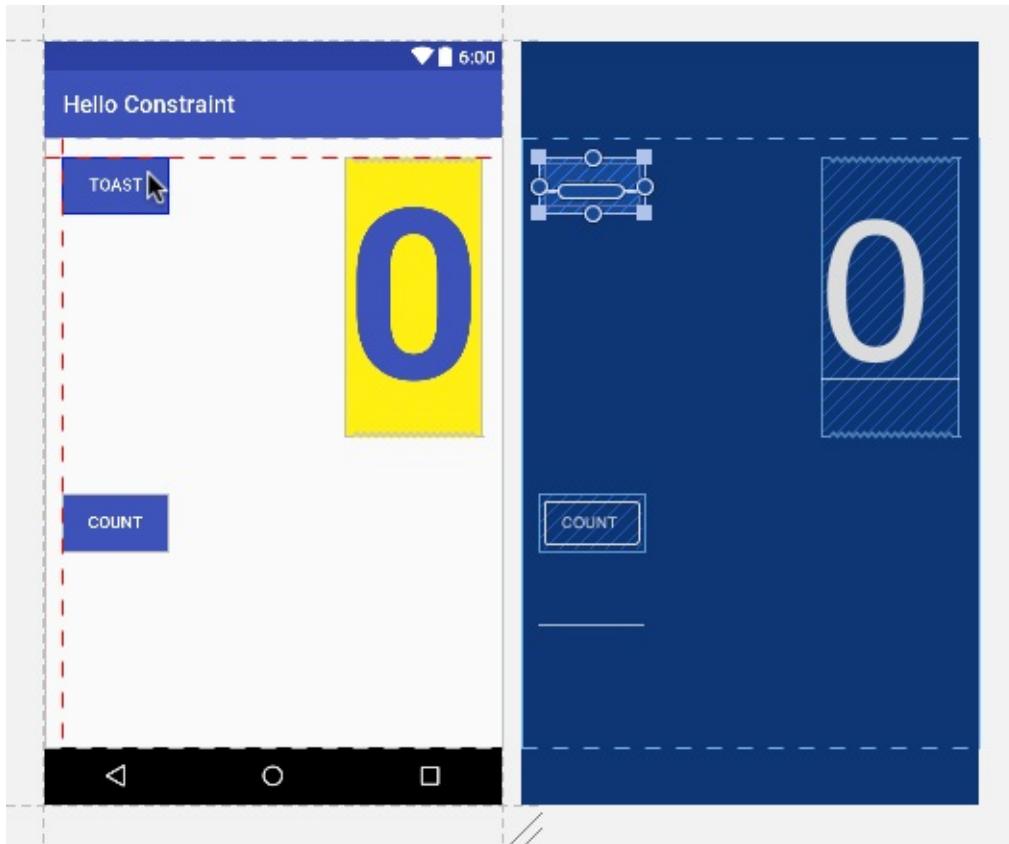
As you release from the click-and-hold, the constraint is made, and the `show_count` view jumps to the right edge of the layout.

This constrains the view to the top edge. After dragging the constraint, the `show_count` view jumps to the top right edge of the layout, because it is anchored to both the top and right edges.

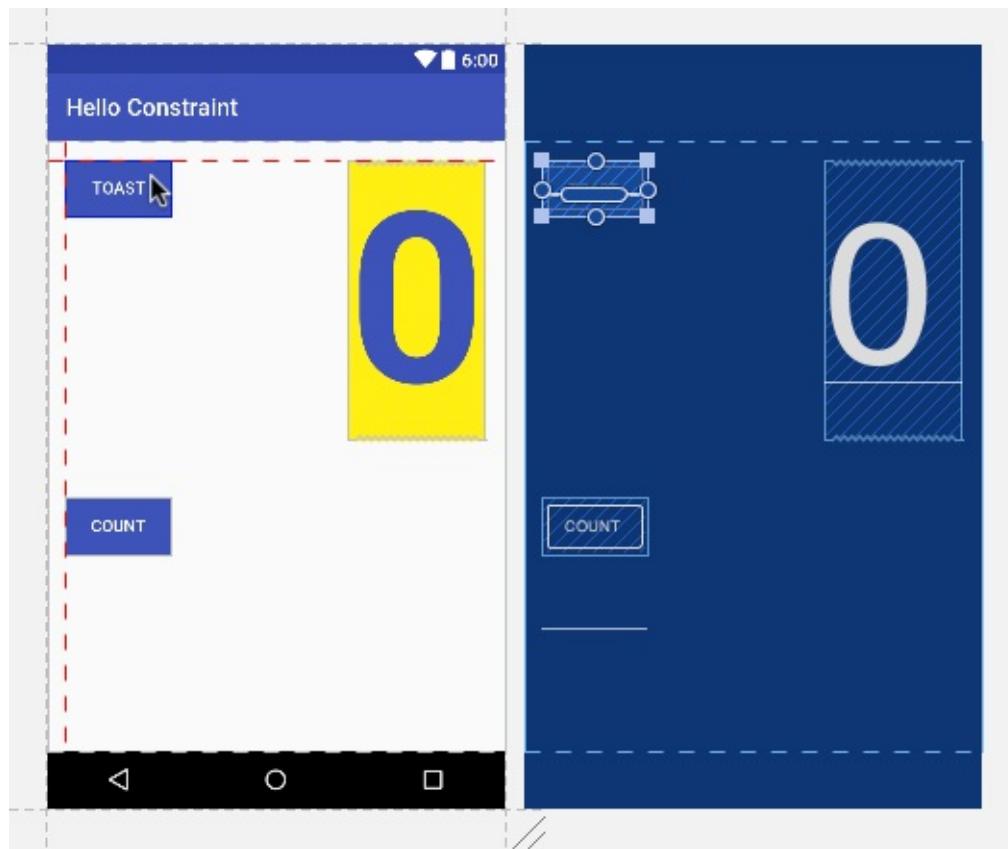
3. Click the `button_toast` view, and use the Properties panel as shown previously to resize the view to `wrap_content` for both the `layout_width` and `layout_height`. Also resize the `button_count` view to `wrap_content` for both the `layout_width` and `layout_height`.

You use `wrap_content` for the buttons so that if the button text is localized into a different language, the button will appear wider or thinner to accommodate the word in the different language.

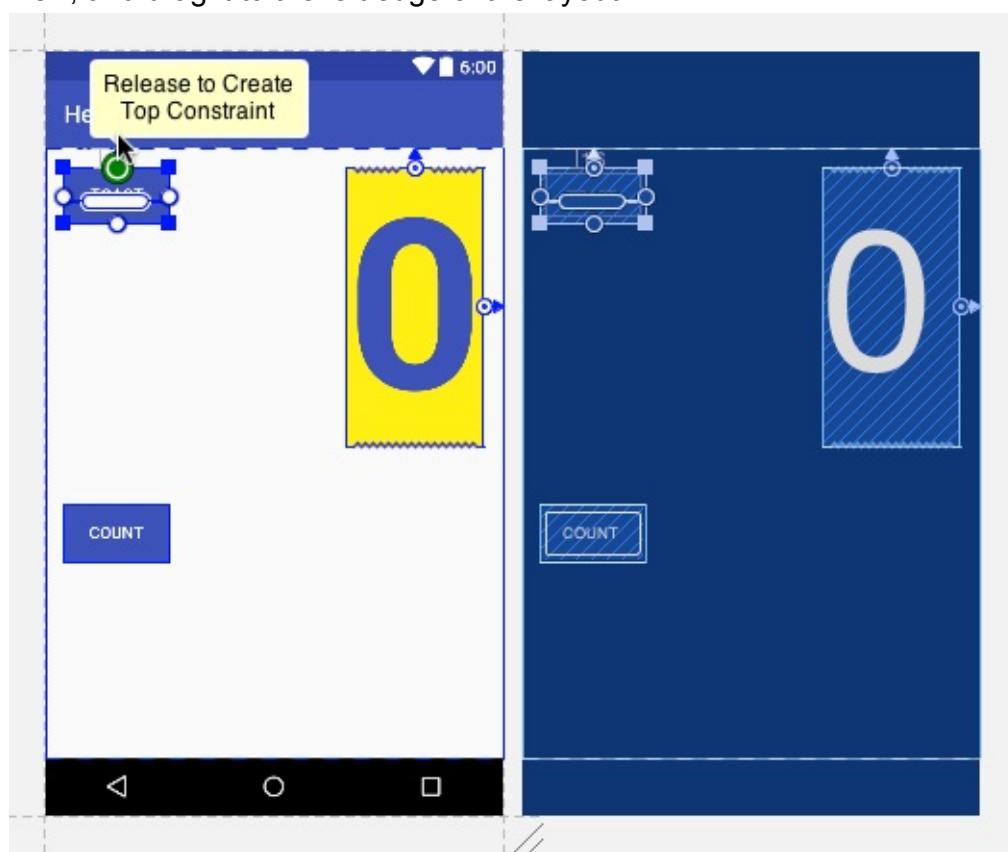
4. Drag the `button_toast` view into position on the left side of the `show_count` view as shown in the figure below. Guides appear so that you can snap the view into position against the top and left margins.



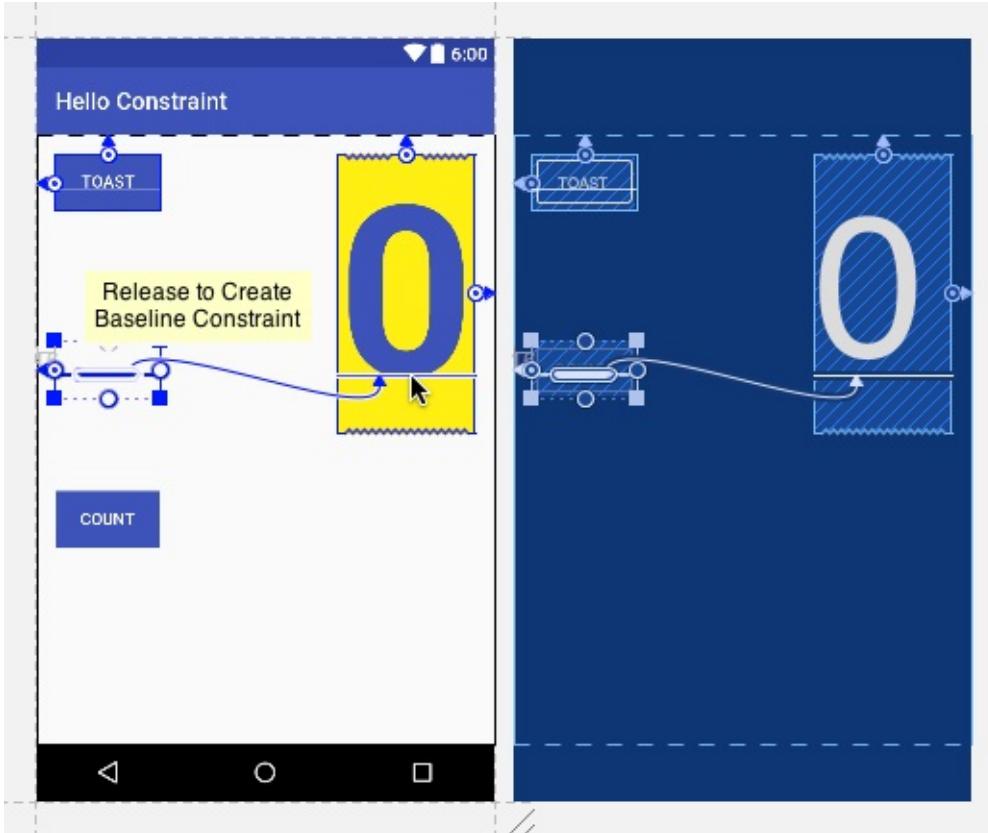
5. Select the `button_toast` view in the layout, click the constraint handle that appears on the top of the view, and drag it to the top edge of the layout under the app bar as shown in the figure below. Then click the constraint handle that appears on the left side of the view, and drag it to the left edge of the layout.



5. Select the `button_toast` view in the layout, click the constraint handle that appears on the top of the view, and drag it to the top edge of the layout under the app bar as shown in the figure below. Then click the constraint handle that appears on the left side of the view, and drag it to the left edge of the layout.



7. To create a baseline constraint between the `button_count` view's text baseline and the `show_count` view's text baseline, select the `button_count` view, and then hover over the view's baseline handle for two seconds until the handle blinks white. Then click and drag the constraint line that appears to the baseline of the `show_count` view, as shown in the figure below.

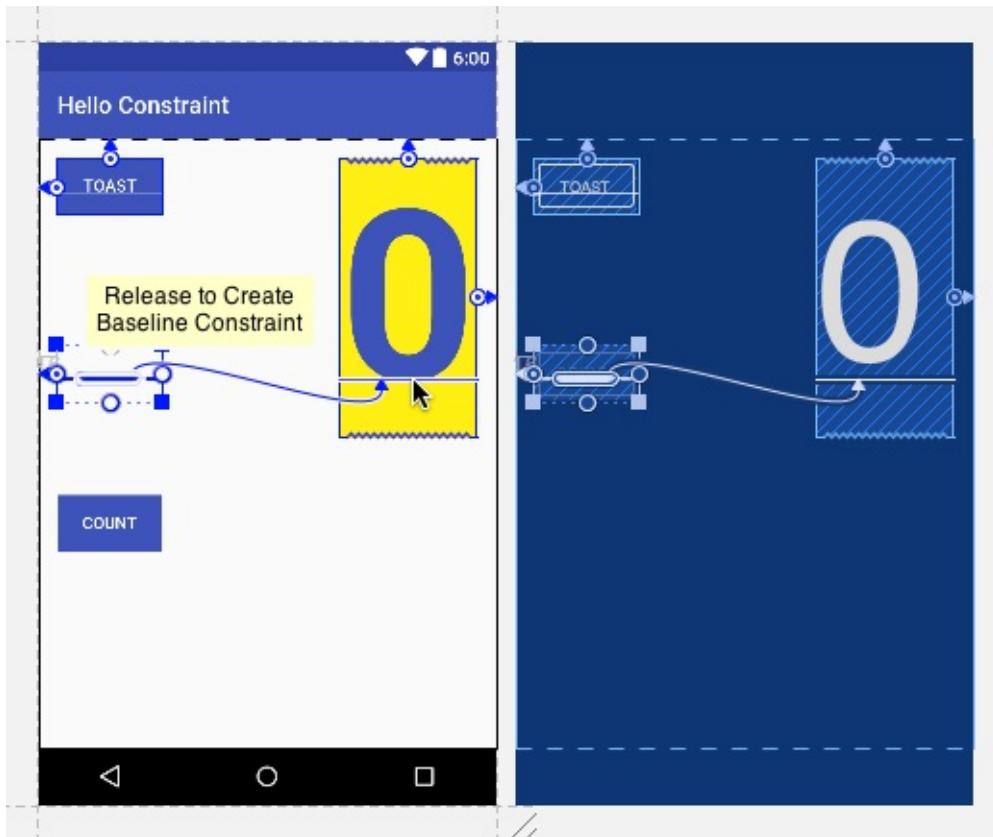


You now have a layout in which each view is set to non-specific dimensions and constrained to the layout. One button's text is aligned to a TextView's baseline, so that if you move the TextView, the button moves with it.

Tip: If a view doesn't have at least *two* constraints, it appears at the top of the layout.

8. Although the `show_count` view already has two constraints, you can add another constraint to it. Drag a constraint line from the constraint handle on the left side of the view to the right side of the `button_count` view, as shown in the figures below.

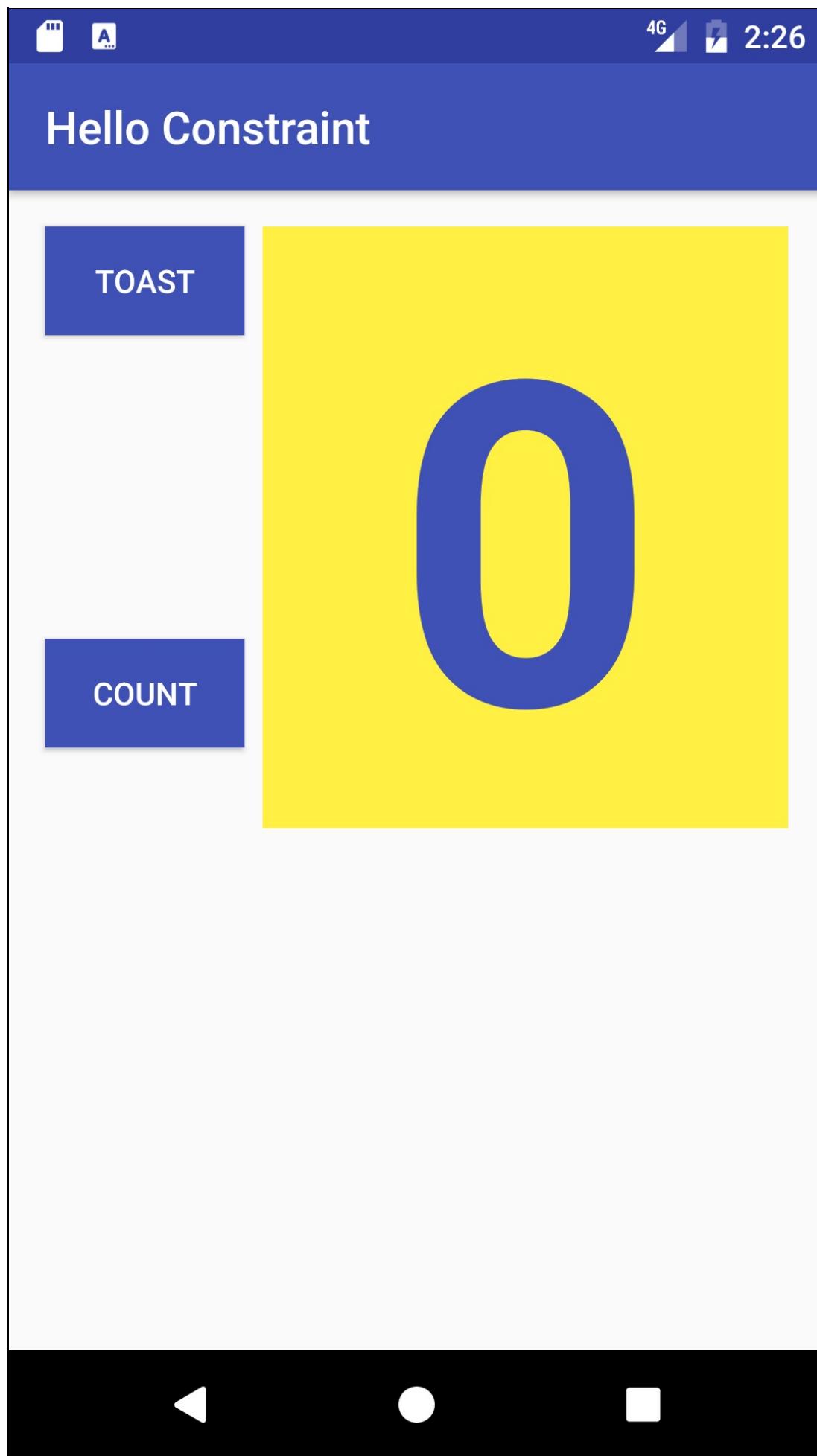
drag the constraint line that appears to the baseline of the `show_count` view, as shown in the figure below.

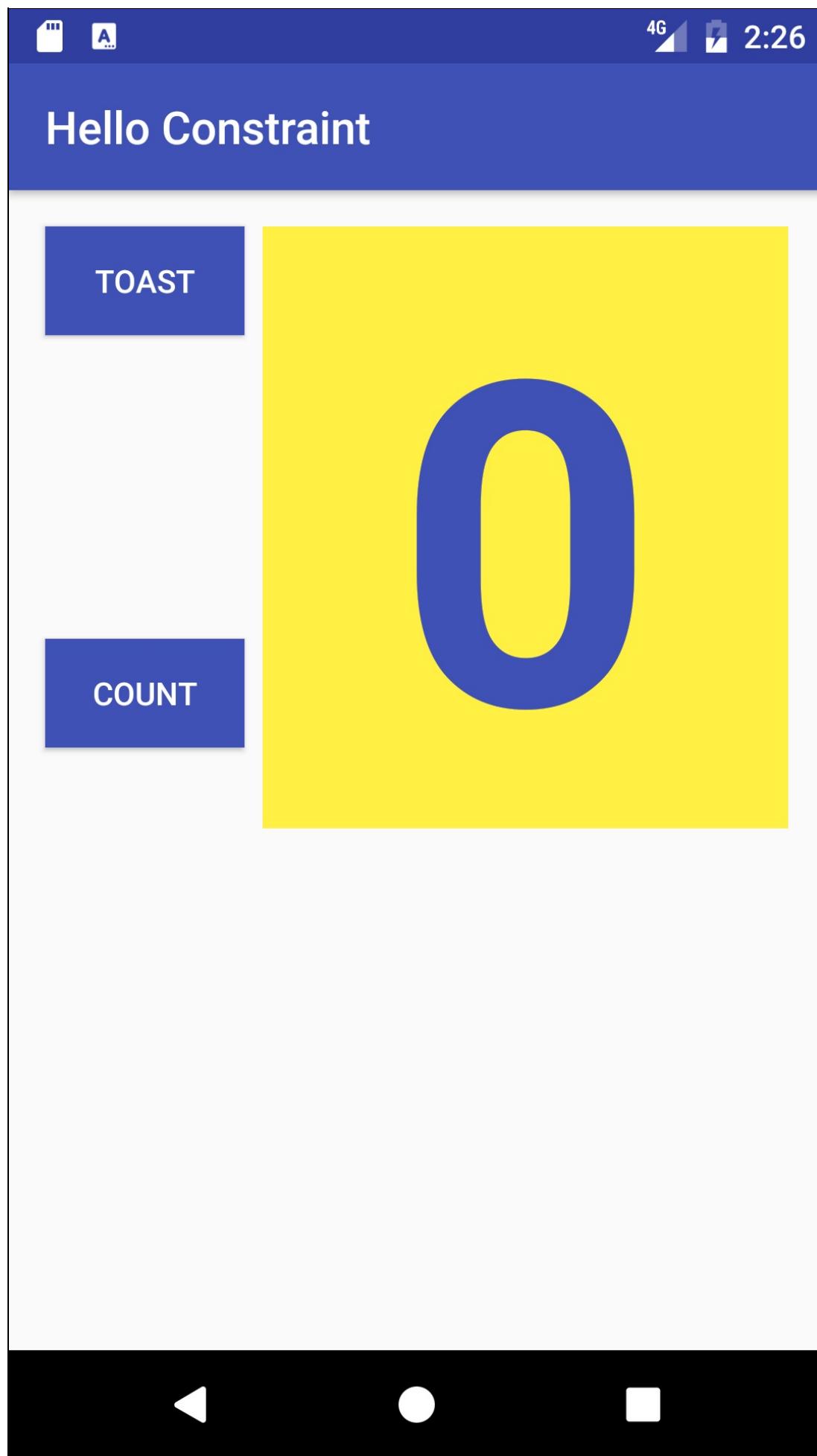


You now have a layout in which each view is set to non-specific dimensions and constrained to the layout. One button's text is aligned to a TextView's baseline, so that if you move the TextView, the button moves with it.

Tip: If a view doesn't have at least *two* constraints, it appears at the top of the layout.

8. Although the `show_count` view already has two constraints, you can add another constraint to it. Drag a constraint line from the constraint handle on the left side of the view to the right side of the `button_count` view, as shown in the figures below.





view that aligns its top edge to the parent view:

```
<Button  
    android:id="@+id/button_toast"  
    ...  
    app:layout_constraintTop_toTopOf="parent"  
    ...
```

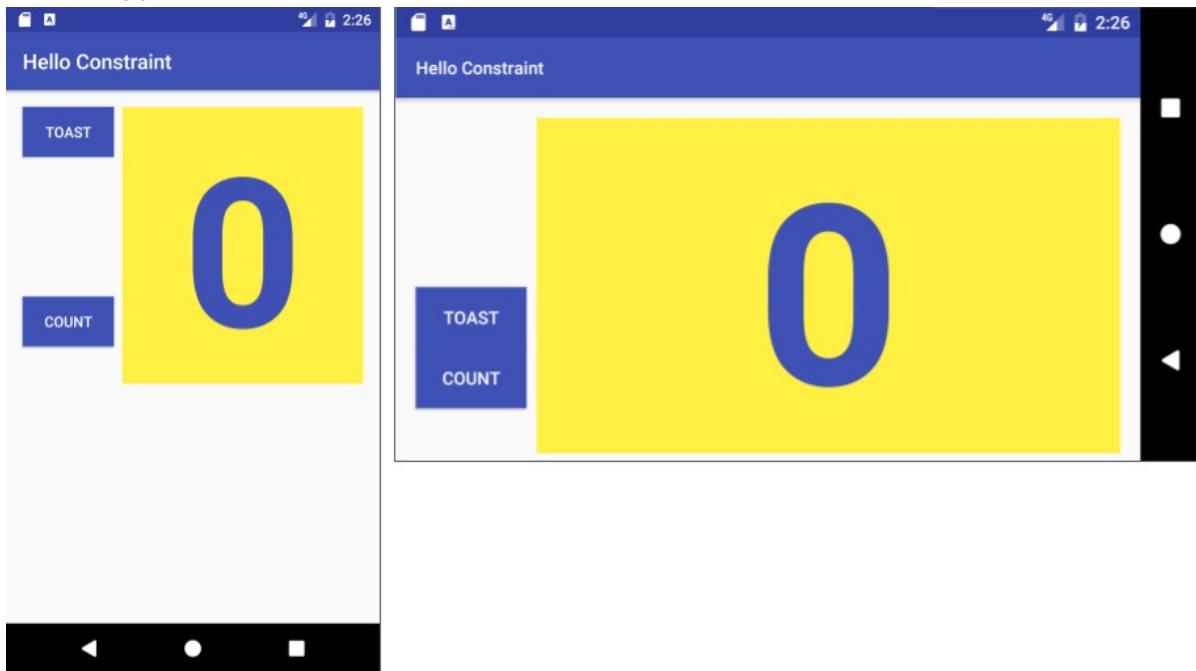
6. Change this constraint so that the `button_toast` view's bottom edge is aligned to the top edge of the `button_count` view.

Hint: If the constraint to align the top of a view to another view is

`app:layout_constraintTop_toTopOf`, what do you think the constraint is to align the bottom of a view to the top of another view? Answer:

```
app:layout_constraintBottom_toTopOf="@+id/button_count"
```

7. Run the app, and switch to landscape mode to see the different layout. The layouts should appear as shown below.



Solution code

Android Studio project: [HelloToast](#)

Android Studio project: [HelloRelative](#)

Android Studio project: [HelloConstraint](#)

view that aligns its top edge to the parent view:

```
<Button  
    android:id="@+id/button_toast"  
    ...  
    app:layout_constraintTop_toTopOf="parent"  
    ...
```

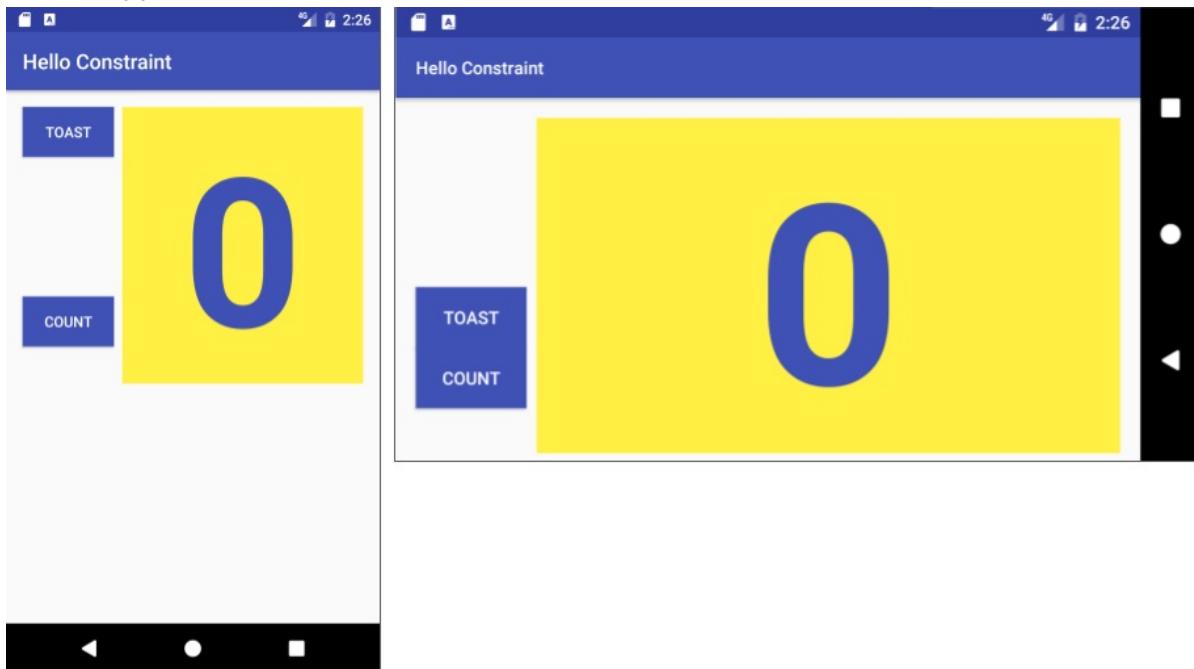
6. Change this constraint so that the `button_toast` view's bottom edge is aligned to the top edge of the `button_count` view.

Hint: If the constraint to align the top of a view to another view is

`app:layout_constraintTop_toTopOf`, what do you think the constraint is to align the bottom of a view to the top of another view? Answer:

```
app:layout_constraintBottom_toTopOf="@+id/button_count"
```

7. Run the app, and switch to landscape mode to see the different layout. The layouts should appear as shown below.



Solution code

Android Studio project: [HelloToast](#)

Android Studio project: [HelloRelative](#)

Android Studio project: [HelloConstraint](#)

- [Layouts, Views, and Resources](#)

Learn more

Developer Documentation:

- [View](#)
- [Relative Layout](#)
- [Build a UI with Layout Editor](#)
- [Build a Responsive UI with ConstraintLayout](#)

Other:

- Codelabs: [Using ConstraintLayout to design your views](#)

1.3: 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](#)
- [Related concept](#)
- [Learn more](#)

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 in which the user can scroll its contents vertically.

If you have more information than fits on the device's display, you can create a *scrolling view* so 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 device display. You can also use a scrolling view to enable users to enter multiple lines of text, or to combine UI 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](#), and developers should place only *one* view as a child within it, where the child view contains the entire contents to scroll. This child view may itself be a view group (such as a layout manager like [LinearLayout](#)) with a complex hierarchy of objects. 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 the 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, consider using a [RecyclerView](#), which is described in a separate practical.

What you should already KNOW

From previous practicals, you should be able to:

- Create a Hello World app with Android Studio.
- Run an app on an emulator or a device.
- Implement a TextView in a layout for an app.
- Create and use string resources.
- Convert layout dimensions to resources.

What you will LEARN

You will learn to:

- Use XML code to add multiple TextView elements.
- Use XML code to define a scrolling view.
- Display free-form text with some HTML formatting tags.
- Style the TextView background color and text color.
- Include a web link in the text.

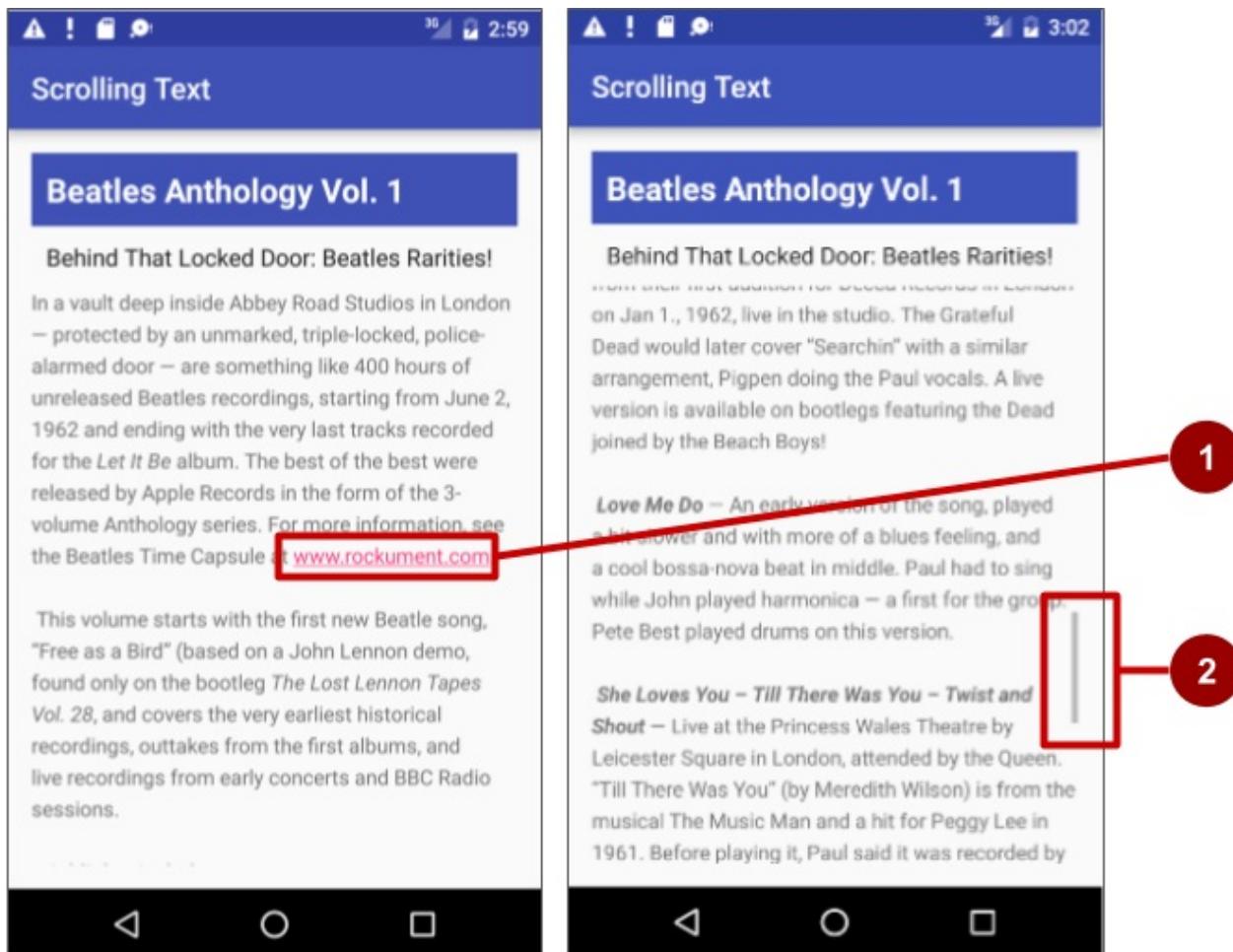
What you will DO

In this practical, 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.

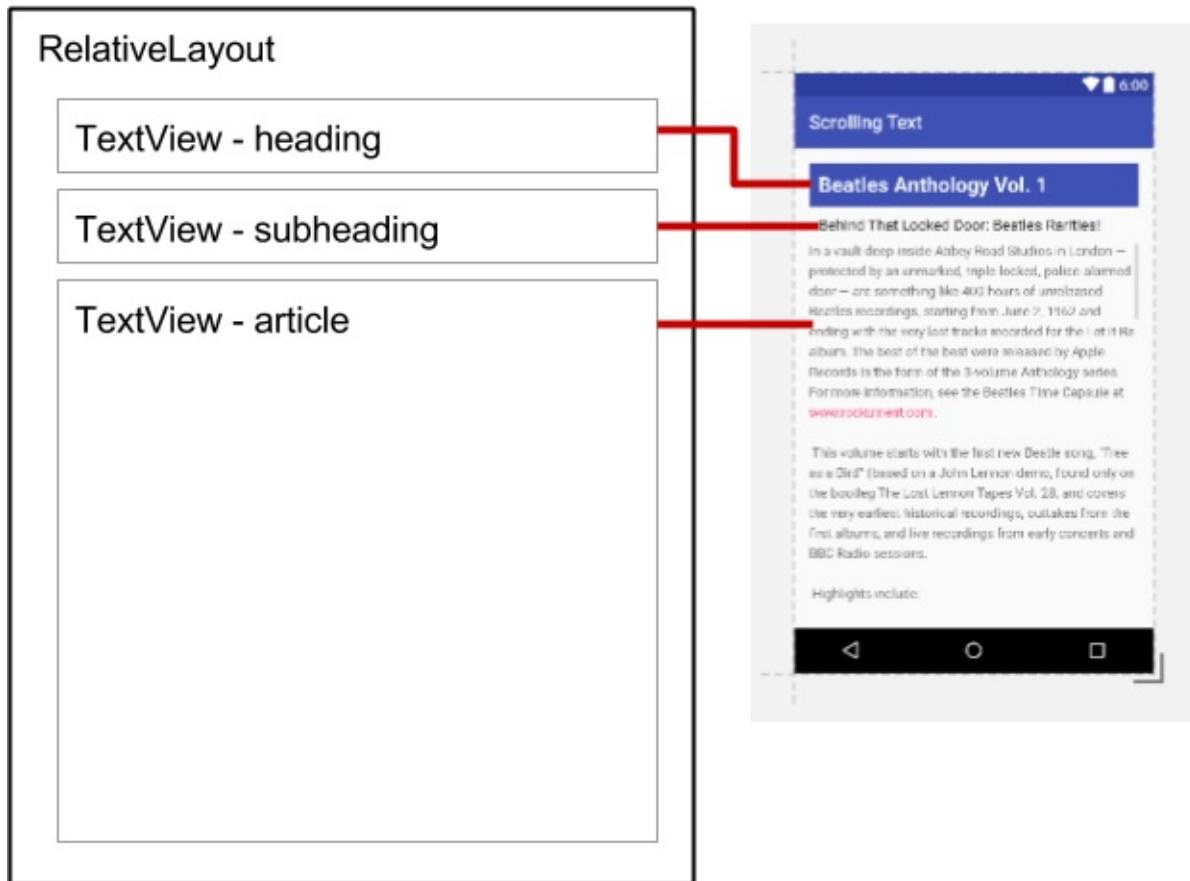


In the above figure, the following appear:

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 to the layout for an article title and subtitle, and change the existing "Hello World" TextView element to show a lengthy article. The figure below is a diagram of the layout.



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 by the Empty Layout template is a child View within the RelativeLayout view group. For more information about using a RelativeLayout, see the [Relative Layout API Guide](#).

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>android:id</code>	<code>"@+id/article_heading"</code>
<code>layout_width</code>	<code>"match_parent"</code>
<code>layout_height</code>	<code>"wrap_content"</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>

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

4. 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`.

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

5. 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`.

6. 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>android:id</code>	<code>"@+id/article_subheading"</code>
<code>layout_width</code>	<code>"match_parent"</code>
<code>layout_height</code>	<code>"wrap_content"</code>
<code>android:layout_below</code>	<code>"@+id/article_heading"</code>
<code>android:padding</code>	<code>"@dimen/padding_regular"</code>
<code>android:textAppearance</code>	<code>"@android:style/TextAppearance"</code>
<code>android:text</code>	<code>"Article Subtitle"</code>

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`.

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

TextView Attribute	Value
<code>android:id</code>	<code>"@+id/article"</code>
<code>android:lineSpacingExtra</code>	<code>"5sp"</code>
<code>android:layout_below</code>	<code>"@+id/article_subheading"</code>
<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**.
10. Reformat and align the code by choosing **Code > Reformat Code**. It is a good practice to reformat and align your code so that it is easier for you and others to understand.

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 [ScrollingText app](#), or use your own generic 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):

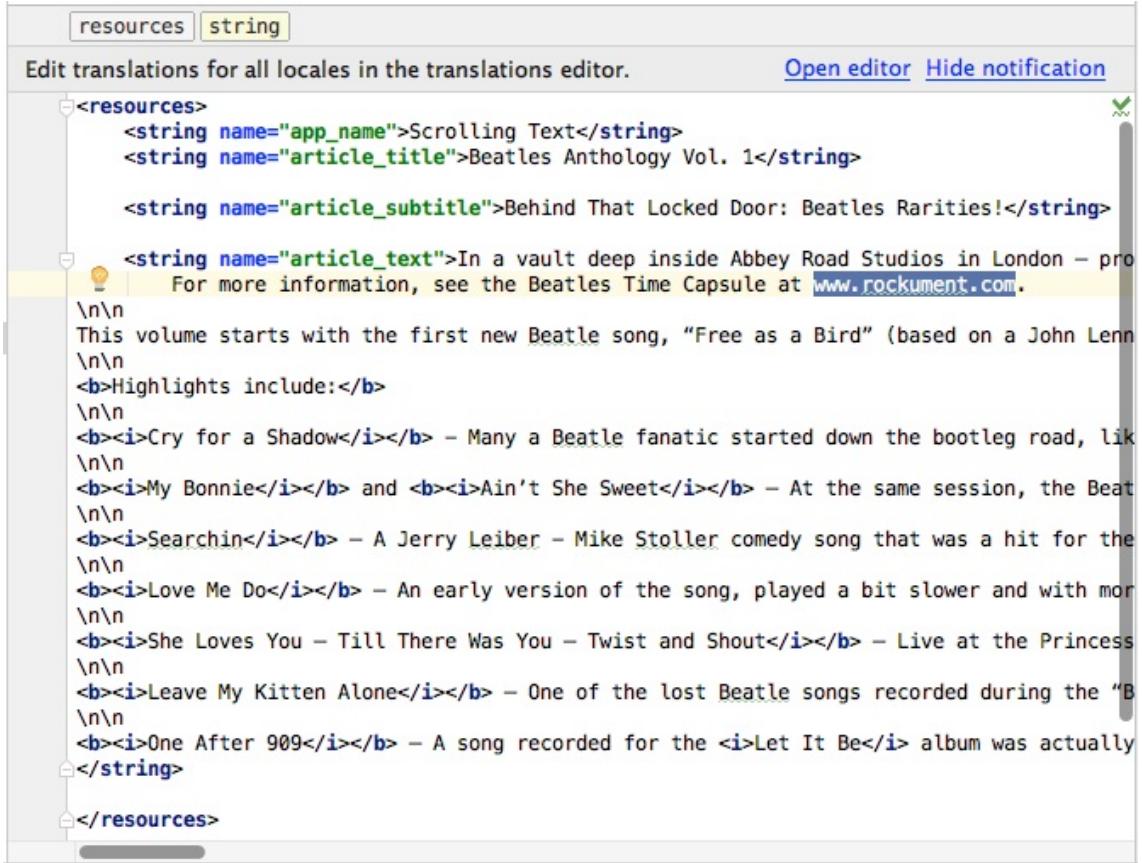
- i. As you enter or paste text in the **strings.xml** file, the text lines don't wrap around to the next line—they extend beyond the right margin. This is the correct behavior—each new line of text starting at the left margin represents an entire paragraph.
- ii. Enter `\n` to represent the end of a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

- iii. 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.
- iv. Enter the HTML and `` tags around words that should be in bold.
- v. 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.
- vi. You can combine bold and italics by combining the tags, as in ... words...`</i>`. Other HTML tags are ignored.
- vii. Enclose The entire text within `<string name="article_text"> </string>` in the strings.xml file.

- viii. Include a web link to test, such as www.google.com (the example below uses 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.



The screenshot shows the 'string' tab selected in the 'resources' editor. The XML code defines several string resources:

```

<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 – pro
        \n\n
        This volume starts with the first new Beatle song, "Free as a Bird" (based on a John Lenn
        \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, lik
        \n\n
        <b><i>My Bonnie</i></b> and <b><i>Ain't She Sweet</i></b> – At the same session, the Beat
        \n\n
        <b><i>Searchin</i></b> – A Jerry Leiber – Mike Stoller comedy song that was a hit for the
        \n\n
        <b><i>Love Me Do</i></b> – An early version of the song, played a bit slower and with mor
        \n\n
        <b><i>She Loves You – Till There Was You – Twist and Shout</i></b> – Live at the Princess
        \n\n
        <b><i>Leave My Kitten Alone</i></b> – One of the lost Beatle songs recorded during the "B
        \n\n
        <b><i>One After 909</i></b> – A song recorded for the <i>Let It Be</i> album was actually
    </string>

    </resources>

```

A yellow highlight covers the line containing the URL www.rockument.com. The status bar at the top right shows a green checkmark icon.

4. Run the app.

The article appears, and you can even scroll it, but the scrolling is not smooth and there is no scroll bar because you haven't yet included a ScrollView (which you will do in the next task). Note also that tapping a web link does not currently do anything. You will also fix that in the next task.

Solution code

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

    <TextView
        android:id="@+id/article"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        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:id="@+id/article"  
    ...  
    android:autoLink="web"  
    ... />
```

2.2 Add a `ScrollView` to the layout

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

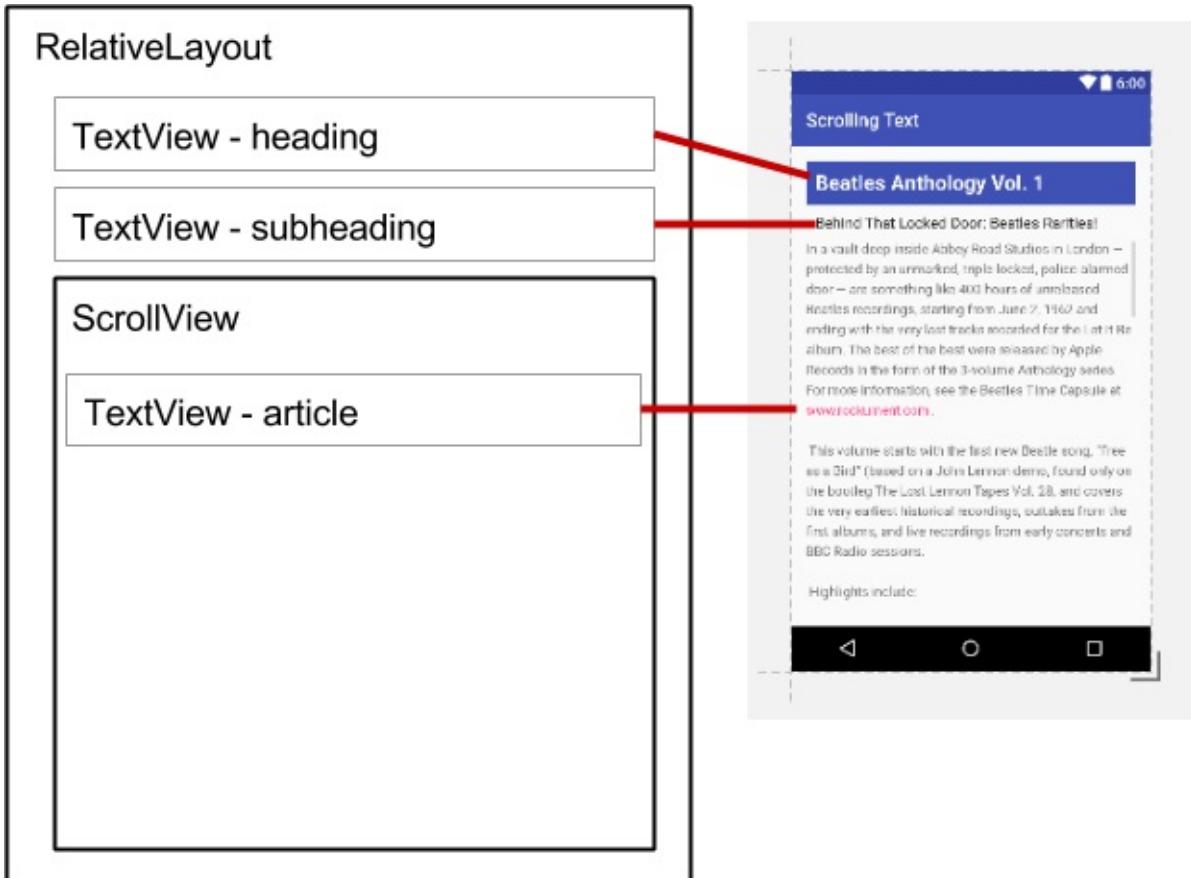
1. Add a `ScrollView` between the `article_subheading` `TextView` and the `article` `TextView`. As you enter `<ScrollView`, Android Studio automatically adds `</ScrollView>` at the end, and presents the `android:layout_width` and `android:layout_height` attributes with suggestions. Choose **wrap_content**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

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

2. Move the ending `</ScrollView>` code *after* the `article` `TextView` so that the `article` `TextView` attributes are inside the `ScrollView` XML element.
3. 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:



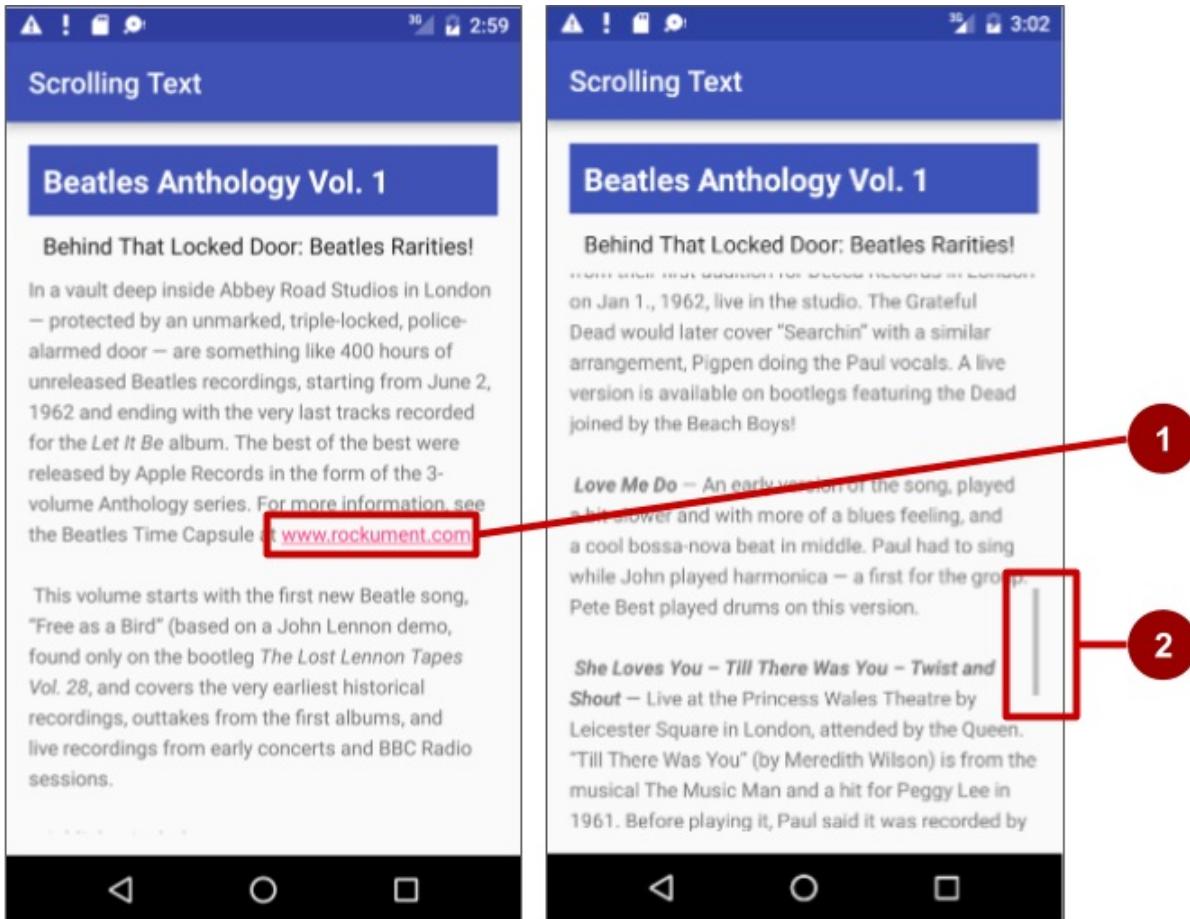
4. Choose **Code > Reformat Code** to reformat the XML code so that the `article` `TextView` now appears indented inside the `<ScrollView>` code.
5. Run the app.

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) into a web link.

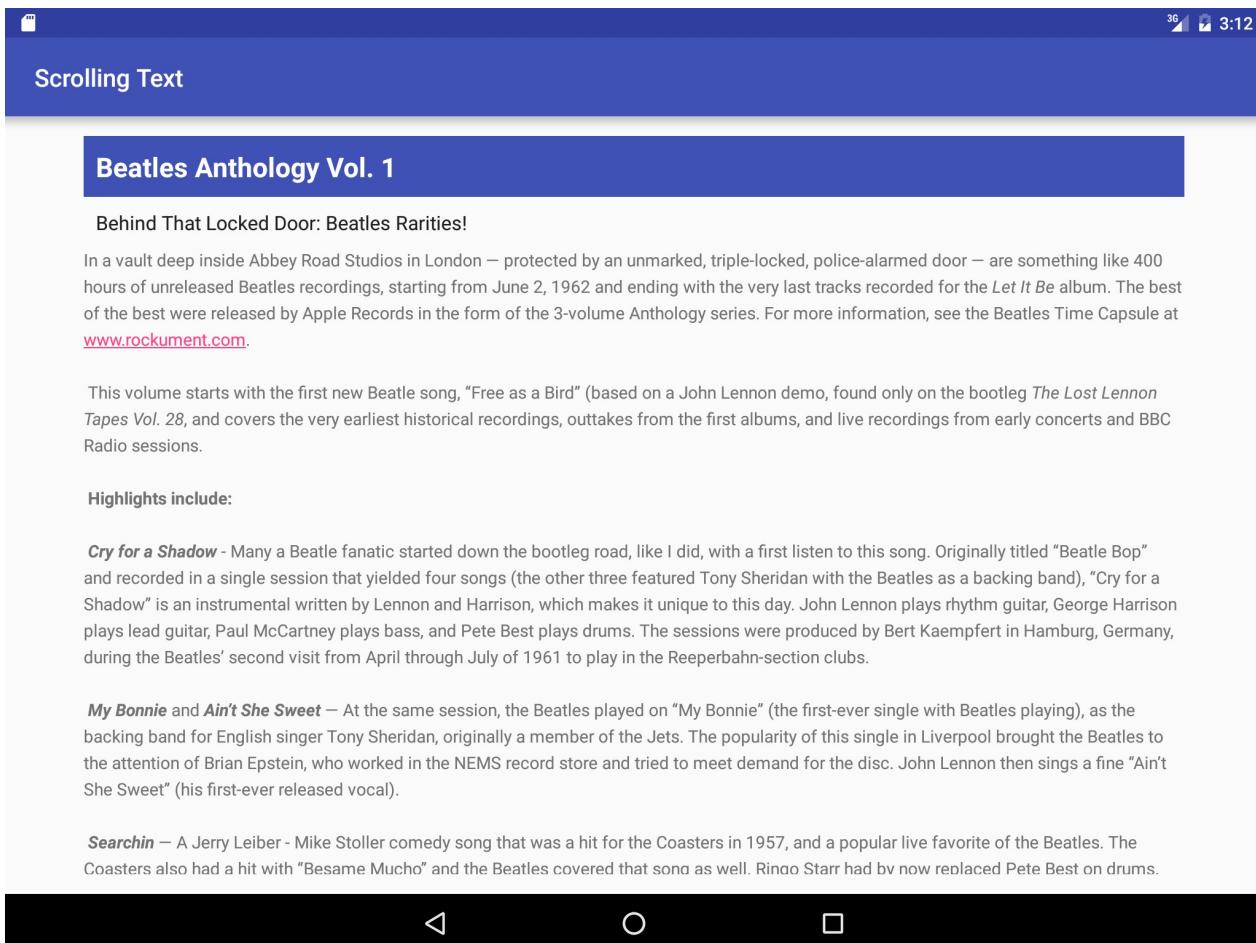
6. Rotate your device or emulator while running the app. Notice how the scrolling view widens to use the full display and still scrolls properly.
7. Run the app on a tablet or tablet emulator. Notice how the scrolling view widens to use

the full display and still scrolls properly.



In the above figure, the following appear:

1. An active web link embedded in free-form text
2. 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:id="@+id/article_heading"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        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:id="@+id/article_subheading"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    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:id="@+id/article"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:lineSpacingExtra="5sp"  
    android:autoLink="web"  
    android:text="@string/article_text"/>  
  
</ScrollView>  
</RelativeLayout>
```

Solution code

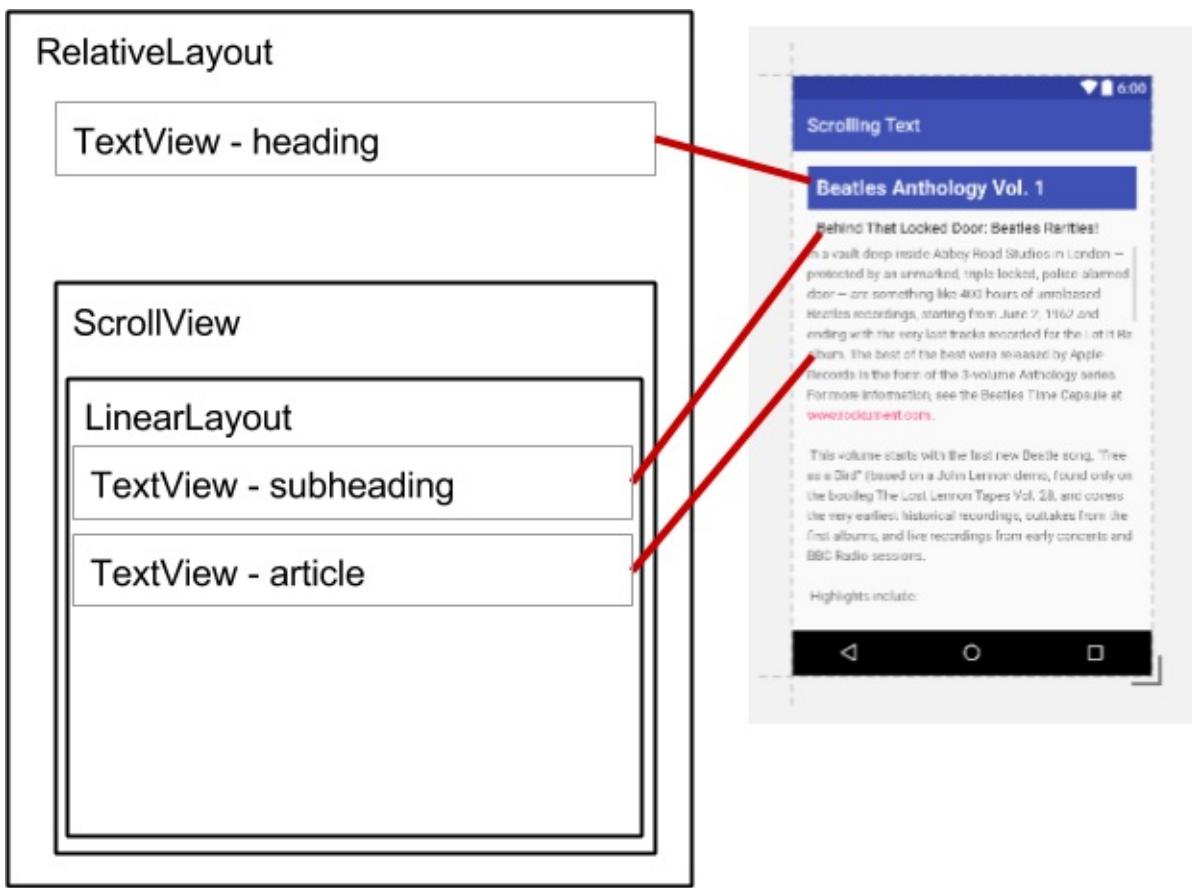
Android Studio project: [ScrollingText](#)

Task 3: Scroll multiple elements

As noted before, the ScrollView view group can contain only one child view (such as the `article` `TextView` you created); 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`.

as shown in the figure below, 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

Android Studio project: [ScrollingText2](#)

Coding challenge

Note: All coding challenges are optional and are not a prerequisite for later lessons.

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 light gray background. A dark blue rectangular box contains the heading 'Beatles Anthology Vol. 1' in large white text. Below this box is a paragraph of text. Further down is a larger block of text under the heading 'Leave My Kitten Alone'. At the bottom of the screen is a light gray button labeled 'ADD COMMENT' in bold black text. The bottom of the phone screen features a black navigation bar with 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

Challenge Solution code

Android Studio project: [ScrollingText3](#)

Summary

In this practical, you learned about Android Studio's view elements and how to scroll and nest code. You worked to:

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

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Text and Scrolling Views](#)

Learn more

Developer Documentation:

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

Other:

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

1.4: 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](#)
- [Related concept](#)
- [Learn more](#)

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; tapping the icon will launch the app.

What you should already KNOW

From previous practicals, you should be able to:

- Understand the basic work flow of Android Studio.

What you will LEARN

Where to find developer resources:

- Within Android Studio.
- In the official [Android developer documentation](#) on the web.
- Elsewhere on the Internet.

What you will DO

In this practical you will:

- 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 HelloToast 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 the API Guide to [Launcher Icons](#) to learn more about how to design effective launcher icons.
 - Learn how to monitor your app's resource usage in Android Studio.

Task 2. Use project templates

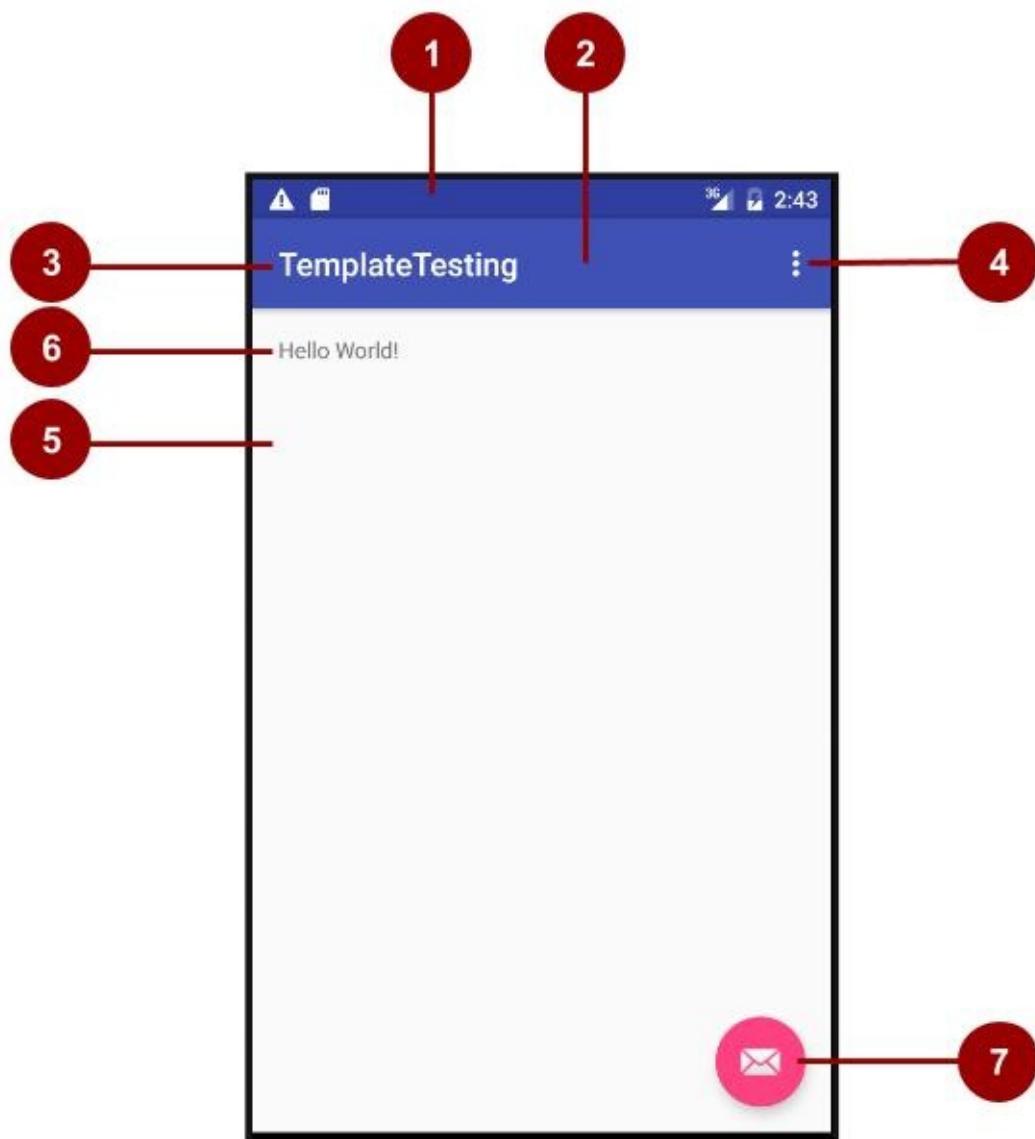
Android Studio provides templates for common and recommended app and activity designs. Using built-in templates saves time, and helps you follow design best practices.

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 provided by Android Studio to assist you in jump-starting your app development.

1. In Android Studio, 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	Status bar 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 hide the status bar , if necessary.
2	AppBarLayout > Toolbar App bar (also called Action bar) provides visual structure, standardized visual elements, and navigation. For backwards compatibility, the AppBarLayout in the template embeds a Toolbar widget with the same functionality. ActionBar class Challenge: App Bar Tutorial	activity_main.xml Look for <code>android.support.v7.widget.Toolbar</code> inside <code>android.support.design.widget.AppBarLayout</code> . Change the toolbar to change the appearance of its parent, the app bar.
3	Application name This is derived from your package name, but can be anything you choose.	AndroidManifest.xml <code>android:label="@string/app_name"</code>
4	Options menu overflow button Menu items for the activity, as well as global options, such as "Search" and "Settings" for the settings menu. Your app menu items go into this menu.	MainActivity.java <code>onOptionsItemSelected()</code> implements what happens when a menu item is selected. res > menu > menu_main.xml Resource that specifies the menu items for the options menu.
5	CoordinatorLayout CoordinatorLayout is a feature-rich layout that provides mechanisms for views to interact. Your app's user interface goes inside this view group.	activity_main.xml Notice that there are no views specified in this layout; rather, it includes another layout with <code><include layout="@layout/content_main" /></code> where the views are specified. This separates system views from the views unique to your app.
6	TextView In the example, used to display "Hello World". Replace this with the views for your app.	content_main.xml All your app's views are defined in this file.

7	Floating Action button (FAB)	activity_main.xml MainActivity.java > onCreate has a stub that sets an onClick listener on the FAB.
---	------------------------------	---

- 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.

- 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. In later lessons, the templates 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.)

- Create a new project or choose an existing project.
- In your project directory, in the Android view, **right-click** the folder with your **java** files.
- Choose **New > Activity > Gallery**.
- 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.

- In Android Studio, choose **File > New > Import Sample**.
- Browse the samples.
- 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**.

Note: The samples contained here are meant as a starting point for further development. We encourage you to design and build your own ideas into them.

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. On [Stack Overflow](#), you can even ask, "How do I setup and use ADB over WiFi?", or "What are the most common memory leaks in Android development?"
- 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?"

4.1. Search on Stack Overflow using tags

1. Go to [Stack Overflow](#)
2. In the search box, type [android].

The [] brackets indicate that you want to search for posts that have been tagged as being about Android.

3. You can combine tags and search terms to make your search more specific. Search for
 - [android] and [layout]
 - [android] "hello world"
4. [Read more](#) about the many ways in which you can search on Stackoverflow.

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.
- Android Studio provides templates for common and recommended app and activity designs. These templates offer working code for common use cases.
- When you create a project, you can choose a template for your first activity.
- While you are further developing your app, activities and other app components can be created from built-in templates.
- Android Studio contains many code samples that you can study, copy, and incorporate with your projects.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Resources to Help You Learn](#)

Learn more

Developer Documentation:

- [Official Android documentation](#)
- [Image Asset Studio](#)
- [Android Monitor page](#)
- [Official Android blog](#)

- [Android Developers blog](#)
- [Google I/O Codelabs](#)
- [Stack Overflow](#)
- [Android vocabulary](#)
- [Google Developer Training website](#)

Code

- [Source code for exercises on GitHub](#)
- [Android code samples for developers](#)

Videos

- [Android Developer YouTube channel](#)
- [Udacity online courses](#)

2.1: 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](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

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.

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, that new activity 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, that current activity 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 you can use in your activity to request an action from another activity (or other app component). 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 practicals, you should be able to:

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

What you will LEARN

You will learn to:

- Create a new activity in Android studio.
- Define parent and child activities for "Up" navigation.
- Start activities with explicit intents.
- Pass data between activities with intent extras.

What you will DO

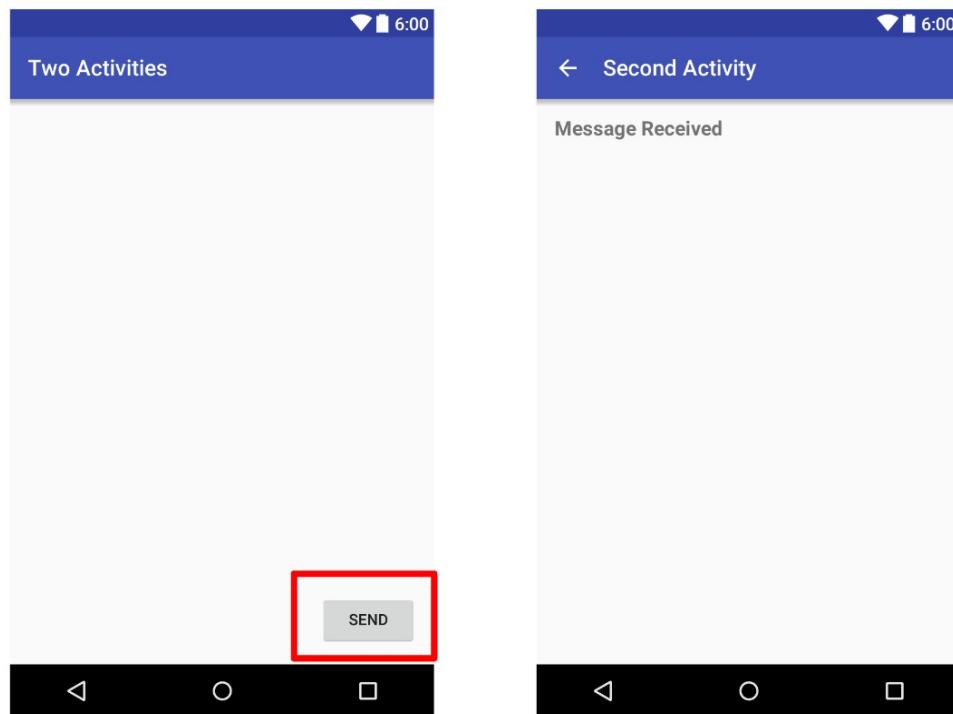
In this practical, you will:

- 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.

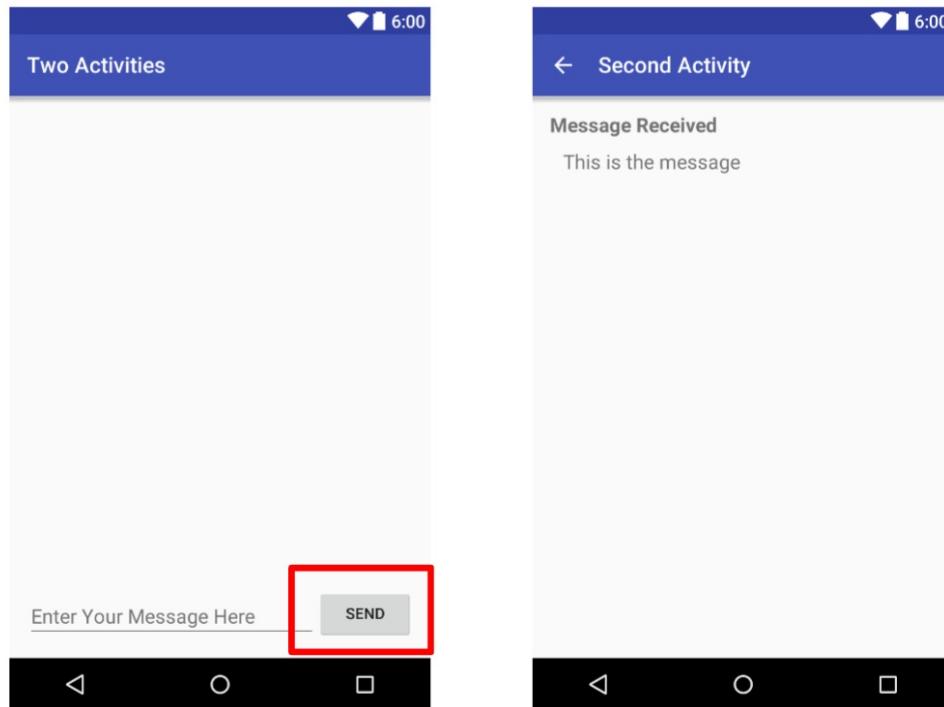
App Overview

In this chapter you will create and build an app called TwoActivities that, unsurprisingly, contains two activities. This app will be built in three stages.

In the first stage, 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.



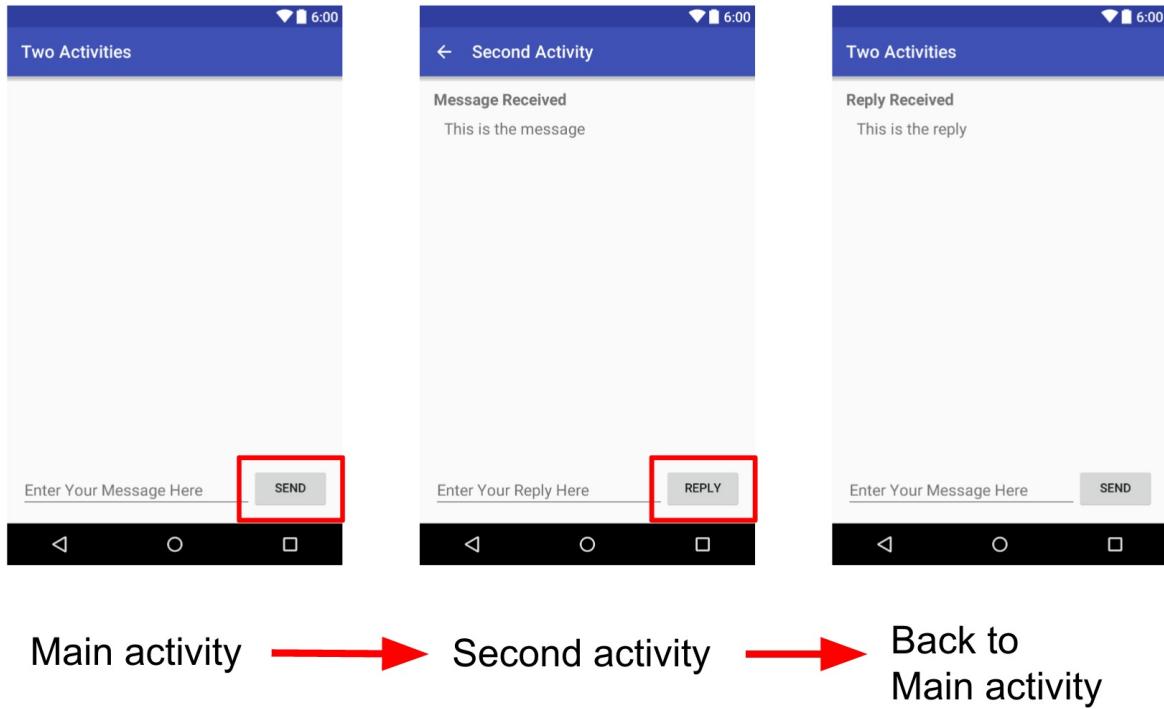
Main activity → Second activity



Main activity → Second activity

In final stage of the TwoActivities app, 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. At this point, use an intent here to pass the reply message back from the second activity to the main activity.

In final stage of the TwoActivities app, 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. At this point, 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 used 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

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

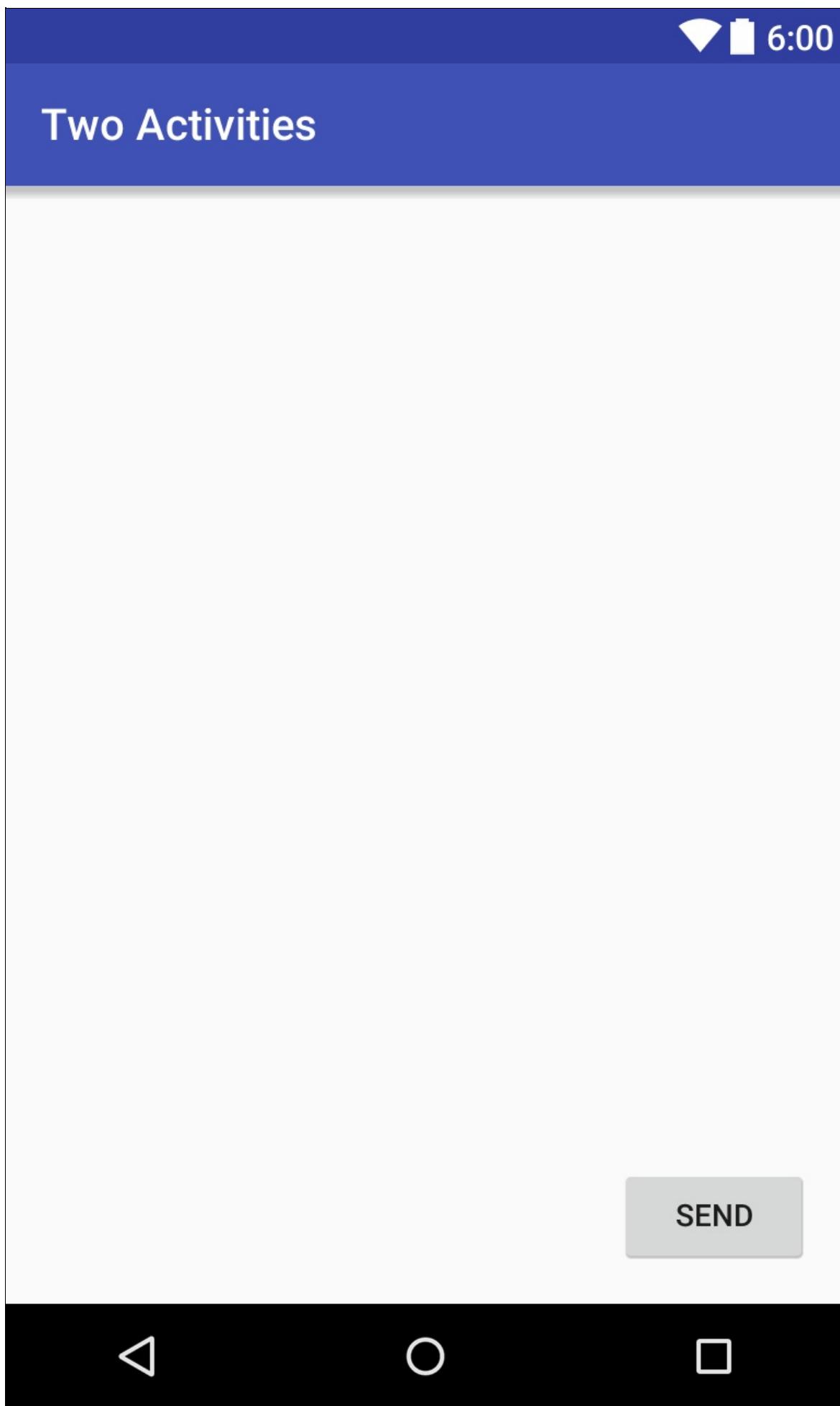
Attribute	Value
android:id	@+id/button_main
android:layout_width	wrap_content
android:layout_height	wrap_content
android:layout_alignParentRight	"true"
android:layout_alignParentBottom	"true"
android:layout_alignParentEnd	"true"
android:text	"Send"
android:onClick	"launchSecondActivity"

This may generate an error that "Method launchSecondActivity is missing in MainActivity." Please ignore this error for now. It will be addressed 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. Choose **Code > Reformat Code** to format the XML code, if necessary.
9. 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, you will 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. The definitions for that variable will be added 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 added to 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. As with the main activity, new activities you create in Android Studio also extend from the AppCompatActivity class.

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 the target of that intent is known, 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. Click the `app` folder for your project and 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 "upward" 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 "upward" 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 upward navigation. These attributes are required for older versions of Android. `android:parentActivityName` is only available for API levels 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>

```

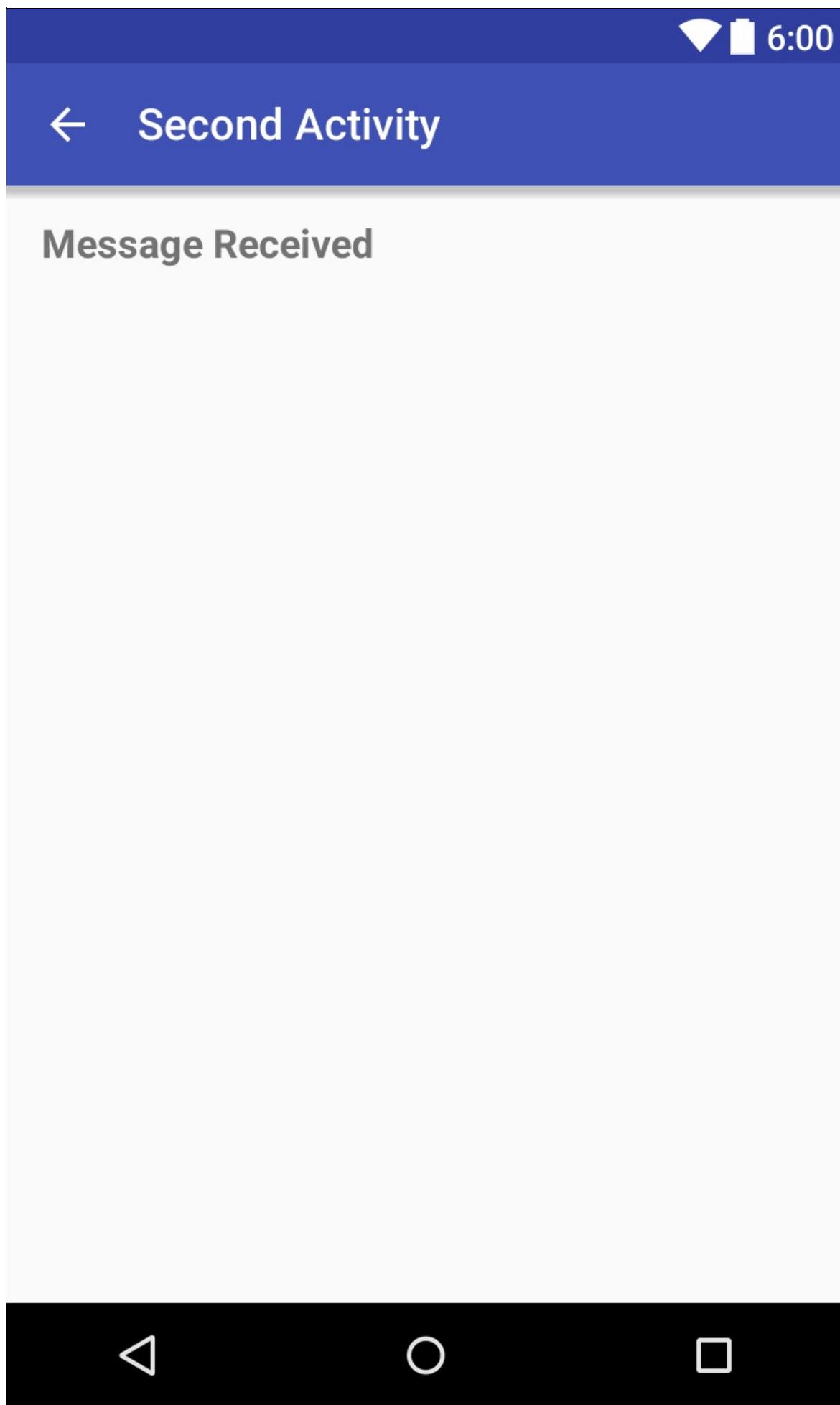
2.3 Define the layout for the second activity

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 main activity sends the intent and the Android system launches the second activity. That second activity appears on the screen. To return 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.

Coding challenge

Note: All coding challenges are optional.

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.

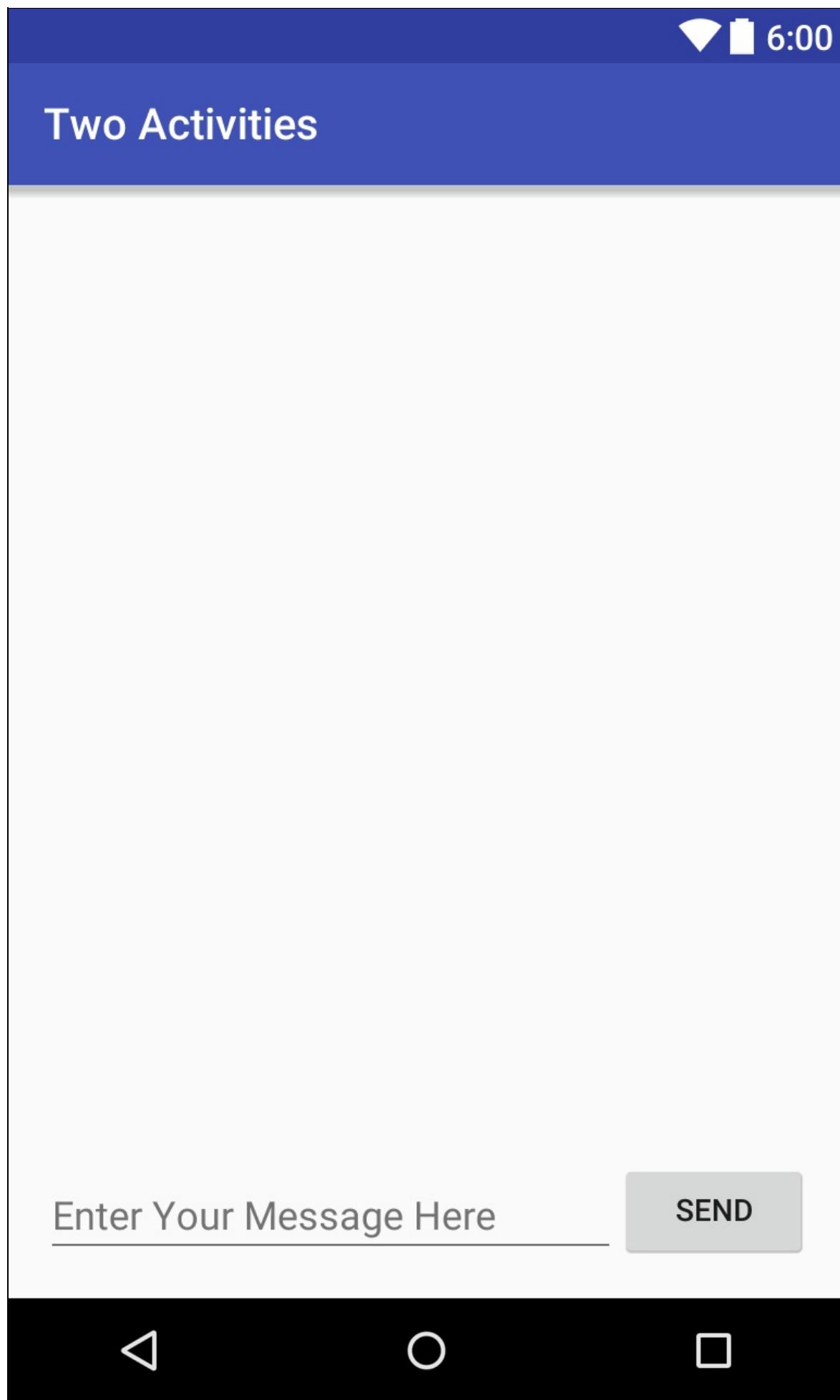
3.1 Add an EditText to the main activity layout

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. Delete the `android:text` attribute.
4. 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 intent *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 intent *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 to the second activity for the message

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 activity. 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"

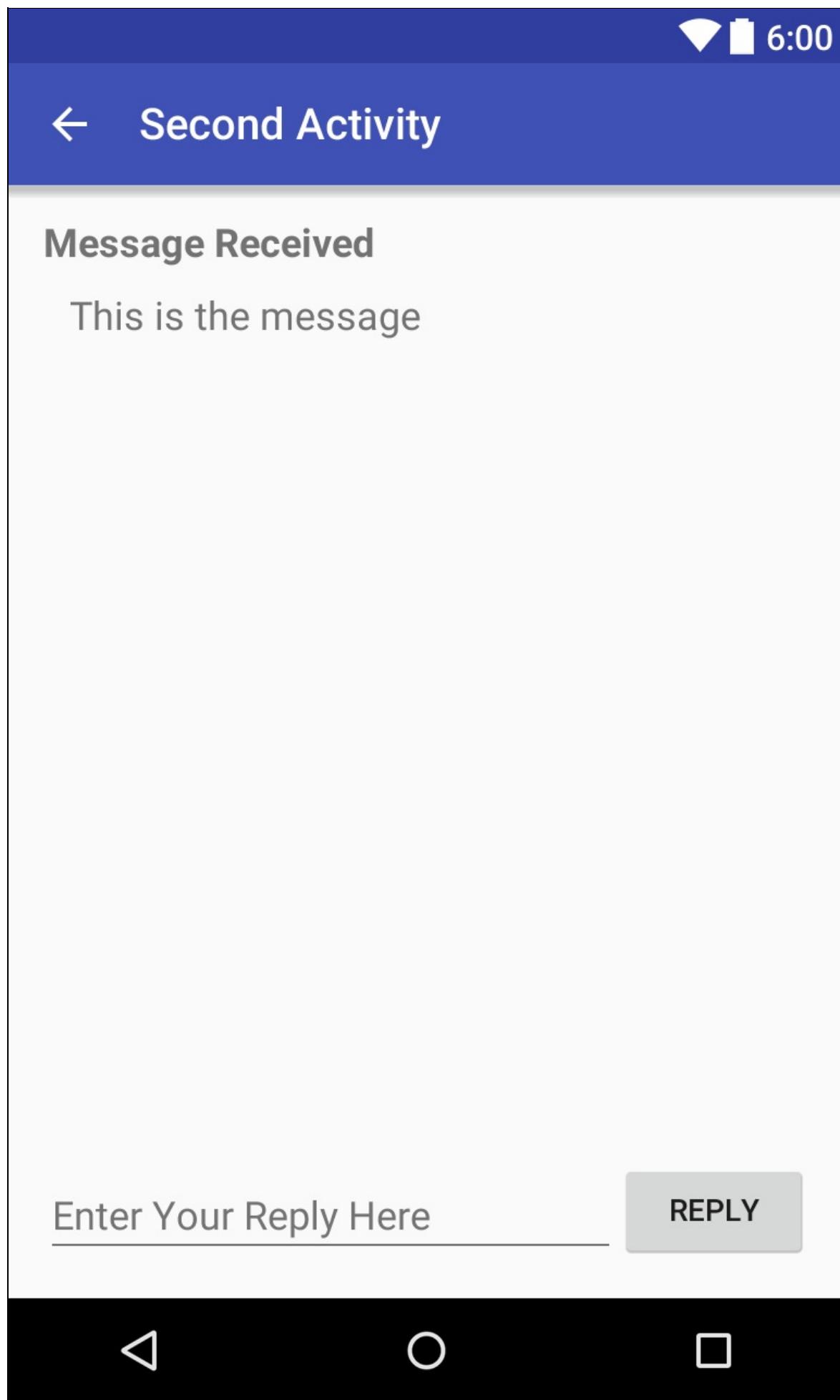
3. Open `res/values/strings.xml` and add string resources for the button text and the hint in the EditText:

```
<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 files open, and Android Studio generates a skeleton method for the onClick handler. You will implement this method in the next task.

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 received 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 this easier, you will 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. However, since you set the TextViews to invisible, 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 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 private variables to hold the reply header and reply `TextViews`:

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

4. In the `onCreate()` method, use `findViewById` to get references from the layout to the reply header and reply `TextView`. Assign those view instances to the private variables:

```
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:

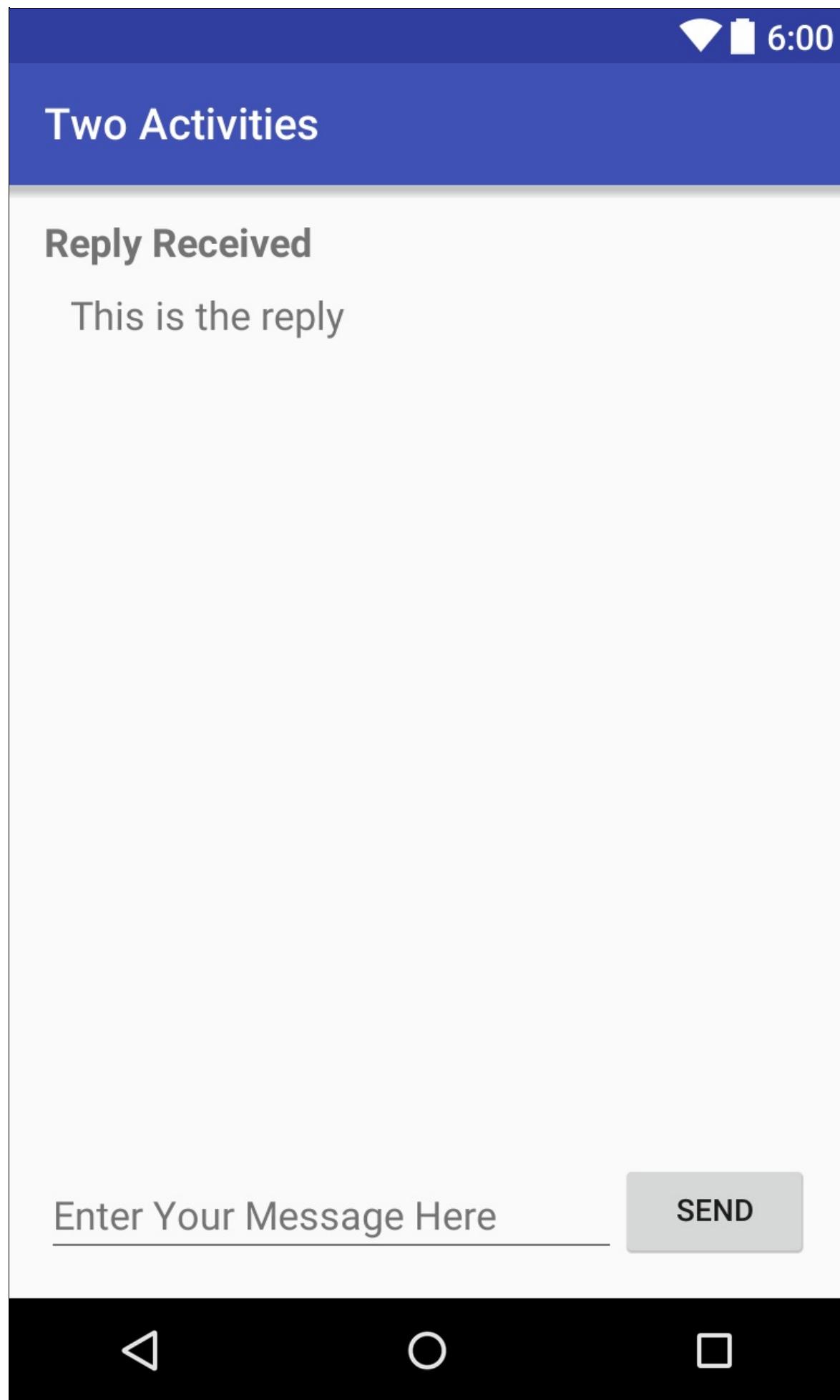
```
mReplyHeadTextView.setVisibility(View.VISIBLE);
```

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);
    }
}
}
}
```

Solution code

Android Studio project: [TwoActivities](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

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.

Summary

In this practical, you have learned that:

- 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 "upward" navigation within your app.
 - To implement an activity in your app, do the following:
- Create an activity Java class.
- Implement a user interface for that activity.
- Declare that new activity in the app manifest.
- When you create a new project for your app, or add a new activity to your app, in Android Studio (with File > New > Activity), template code for each of these tasks is provided for you.
- 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 data.
- 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

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Activities and Intents](#)

Learn more

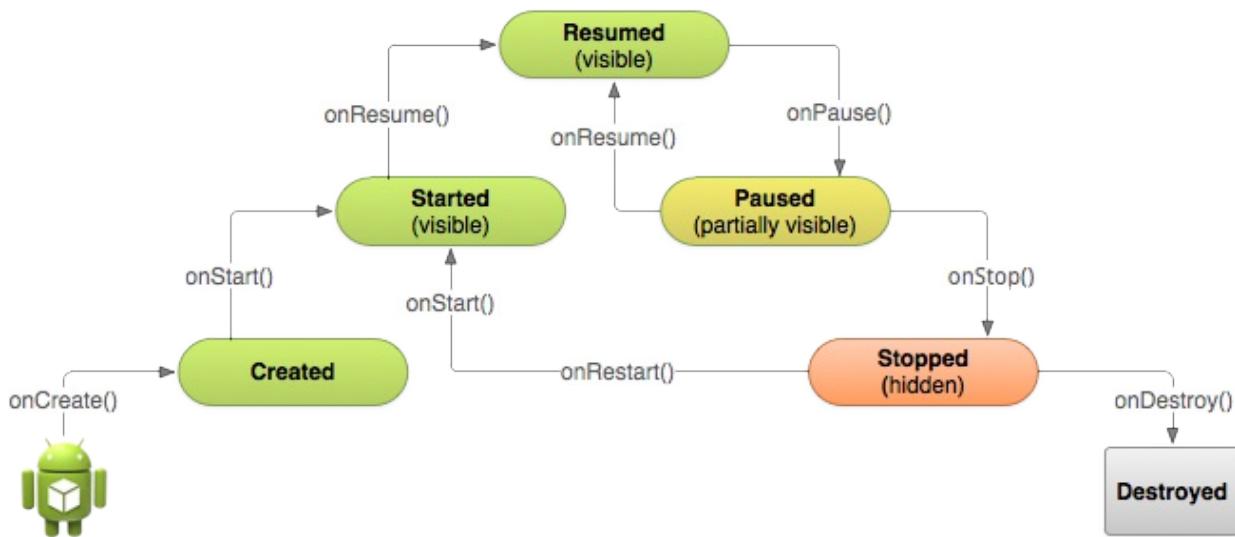
- [Android Application Fundamentals](#)
- [Starting Another Activity](#)
- [Activity \(API Guide\)](#)
- [Activity \(API Reference\)](#)
- [Intents and Intent Filters \(API Guide\)](#)
- [Intent \(API Reference\)](#)

2.2: 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
- Summary
- Related concept
- Learn more

In this practical you'll learn more about the *activity lifecycle*. The activity lifecycle is the set of states an activity can be in during its entire lifetime, from the time it is initially created to when it is destroyed and the system reclaims that activity's resources. 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. It's important to develop your app to prevent this to avoid user confusion. 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.

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. You will then begin working with these changes and exploring how to handle user input under these conditions..

What you should already KNOW

From the previous practicals, you should be able to:

- Create and running an app project in Android Studio.
- Add log statements to your app and viewing those logs in the Android Monitor (logcat).
- Understand and work with activities and intents, and be comfortable interacting with them.

What you will LEARN

You will learn to:

- Understand the activity lifecycle, and when activities are created, pause, stop, and are destroyed.
- Understand the lifecycle callback methods associated with activity changes.
- Understand the effect of actions such as configuration changes that can result in activity lifecycle events.
- Retain activity state across lifecycle events.

What you will DO

In this practical, you will:

- Extend the TwoActivities app from the previous practical to implement the various activity lifecycle callbacks to include logging statements.
- Observe the state changes as your app runs and as you interact with the activities in your app.

- Modify your app to retain the instance state of an activity that is unexpectedly recreated in response to user behavior or configuration change on the device.

App Overview

For this practical you'll add onto the TwoActivities app. The app looks and behaves roughly the same as it did in the last section: with two activities and two messages you can send 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 that you built in the last practical. If you'd prefer to keep the previous TwoActivities project 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 of 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.

4. Use the `onStart()` method as a template to implement the other lifecycle callbacks:

- `onPause()`
- `onRestart()`
- `onResume()`
- `onStop()`
- `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.

5. 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 the finish() method:

```
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");
}

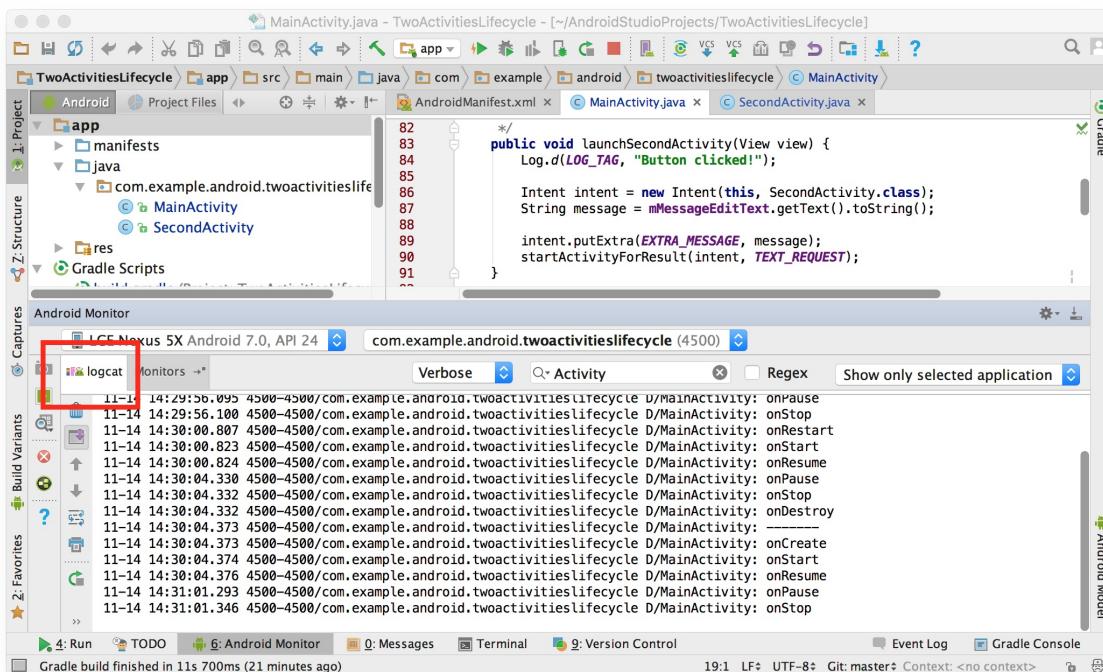
@Override
public void onStop() {
    super.onStop();
    Log.d(LOG_TAG, "onStop");
}

@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 that the lifecycle events 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.

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

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 might think. You may have noticed this set of activities in the last section when 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, their data, or the state of their progress in your app.

The state of each activity is stored as a set of key/value pairs in a `Bundle` object called the *activity instance state*. The system saves default state information to instance state bundle just before the activity is stopped, and passes that bundle to the new activity instance to restore.

To keep from losing data in your activities when they are unexpectedly destroyed and recreated, 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.

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 of the views in your layout are automatically saved across configuration changes, and the current value of an EditText is one of those cases.

The only activity states 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. 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".

We only save the state of those views that might 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: 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, which is called after onStart() after the activity is created.

Most of the time the better place to restore the activity state is in onCreate(), to ensure that your user interface including the state is available as soon as possible. It is sometimes convenient to do it in onRestoreInstanceState() after all of the initialization has been done, or to allow subclasses to decide whether to use your default implementation.

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 and your app starts, the 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 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);  
        }  
    }  
}
```

Solution code

Android Studio Project: [TwoActivitiesLifecycle](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

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.

Summary

- The Activity lifecycle is a set of states an activity migrates through, beginning when it is first created and ending 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 allows 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.
- A portion of the 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()`, which is the preferred way, or `onRestoreInstanceState()`.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Activity Lifecycle and Saving State](#)

Learn more

- [Activity \(API Guide\)](#)

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

2.3: Start Activities with 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](#)
- [Summary](#)
- [Related Concept](#)
- [Learn More](#)

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 practicals, you should be able to:

- Create and use activities.
- Create and send intents between activities.

What you will LEARN

You will learn to:

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

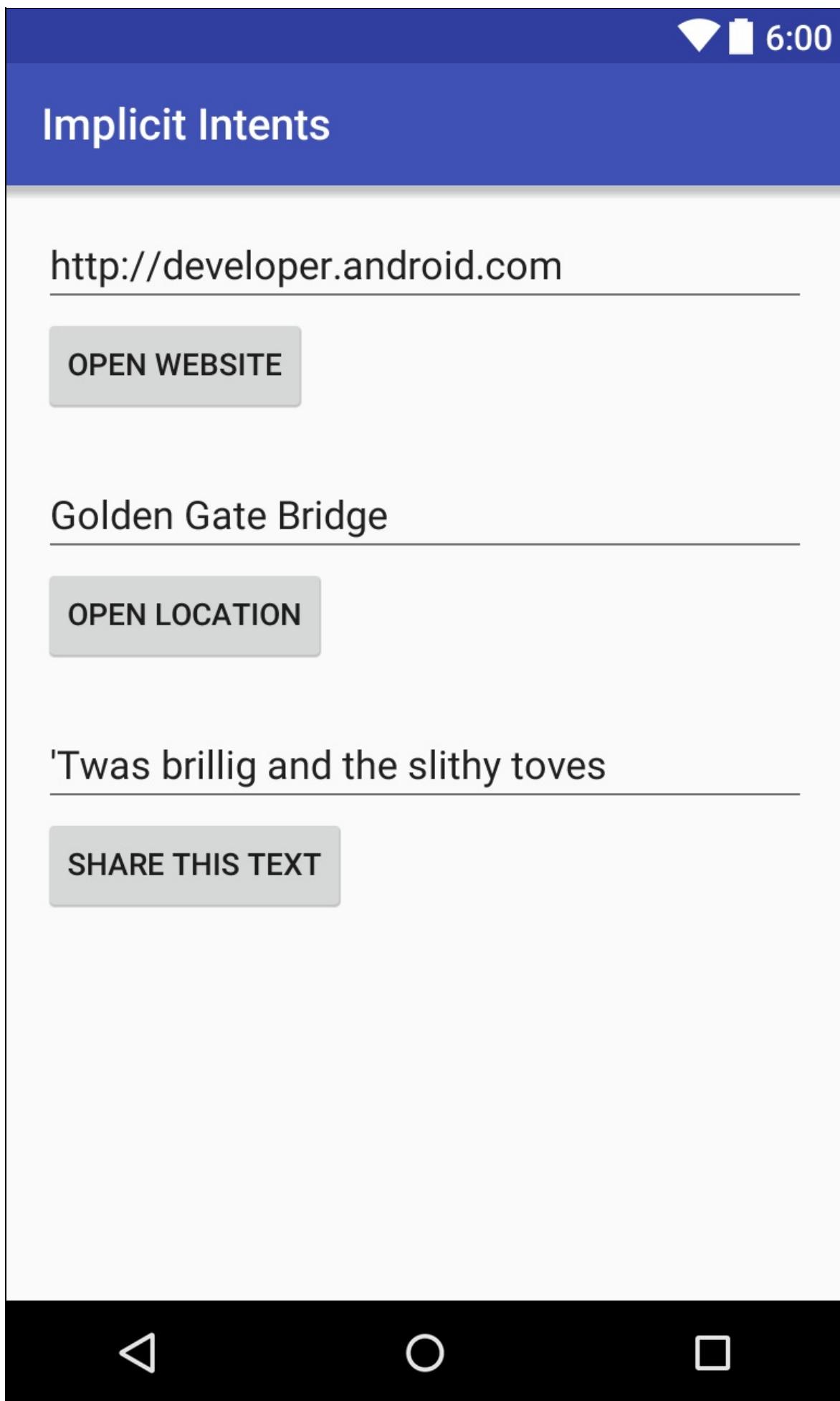
What you will DO

In this practical you will:

- 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 of the text fields are editable (EditText), but contain default values.



Task 1. Create new project and layout

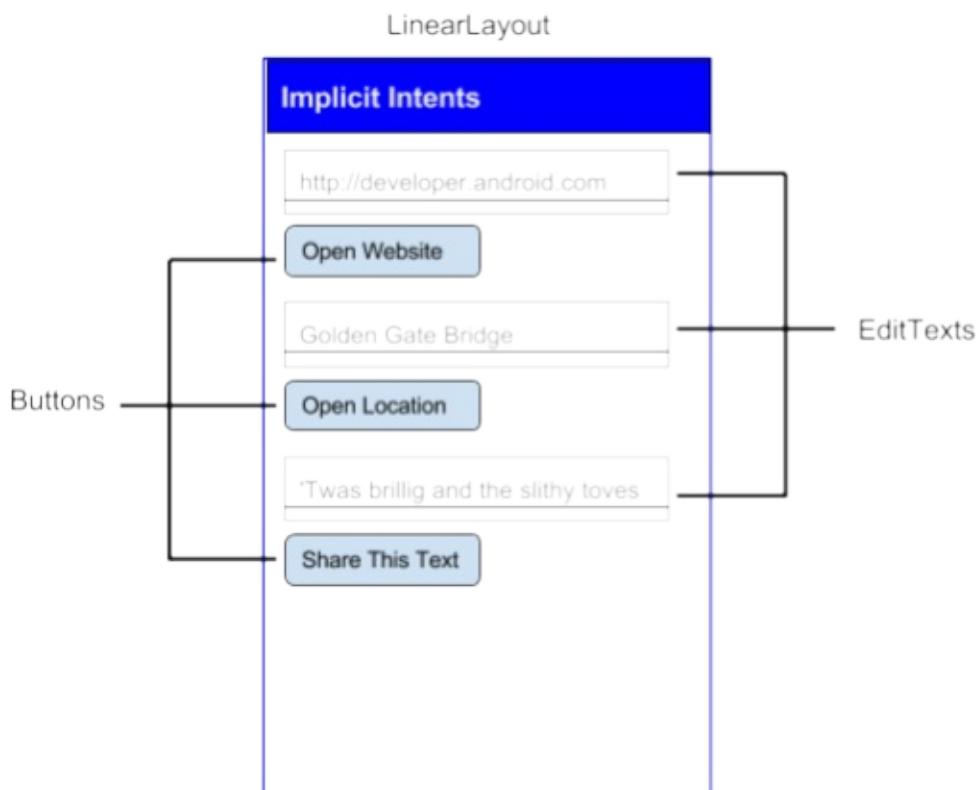
For this exercise, you'll create a new project and app called Implicit Intents with a new layout.

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:

Attribute (EditText)	Value (EditText)
android:id	"@+id/website_edittext"
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:text	"@string/edittext_uri"
Attribute (Button)	Value (Button)
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.
6. Use the same attributes as those in the previous step, but modify these attributes as noted below:

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"

7. 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_location_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        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:text="@string/button_share" />

```

```
        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 implement the on-click handler 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);
```

This intent constructor is different from the one you used to create an explicit intent. In your previous 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(getPackageManager()) != 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 implement the on-click handler method for the second button in the UI ("Open Location.") This method is almost identical to the `openWebsite()` method. 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();
```

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

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

7. Create a new Intent with Intent.ACTION_VIEW as the action and loc as the data.

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

8. 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:

</tr> </table> 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.

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.

Solution Code (not the entire class):

```
public void shareText(View view) {
    String txt = mShareTextEdit.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

So far, you've created apps that use both explicit and implicit intents in order 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. Change the existing ("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 to add an intent filter

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 any 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"	URLs 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 URLs. 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, the 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();  
}
```

- Inside that same if block, get the text view for the message:

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

- Also inside the if-block, set the text of that TextView to the URI:

```
textView.setText(uri_string);
```

- 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.

- 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.

- 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.

Solution code

Android Studio project: [ImplicitIntents](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

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.

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.
- 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.
- The ShareCompat.IntentBuilder class makes it easy to build implicit intents for sharing data to social media or email.

Related Concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Activities and Implicit Intents](#)

Learn More

- [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](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

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 able to:

- Create an Android Studio project, and work with EditText and Button views.
- Build and run your app in Android Studio, on both an emulator and on a device.
- Read and analyze a stack trace, including last on, first off.
- Add log statements and view the system log (logcat) in Android Monitor.

What you will LEARN

You will learn to:

- Run your app in debug mode in an emulator or on a device.
- Step through the execution of your app.
- Set and organize breakpoints.

- Examine and modify variables in the debugger.

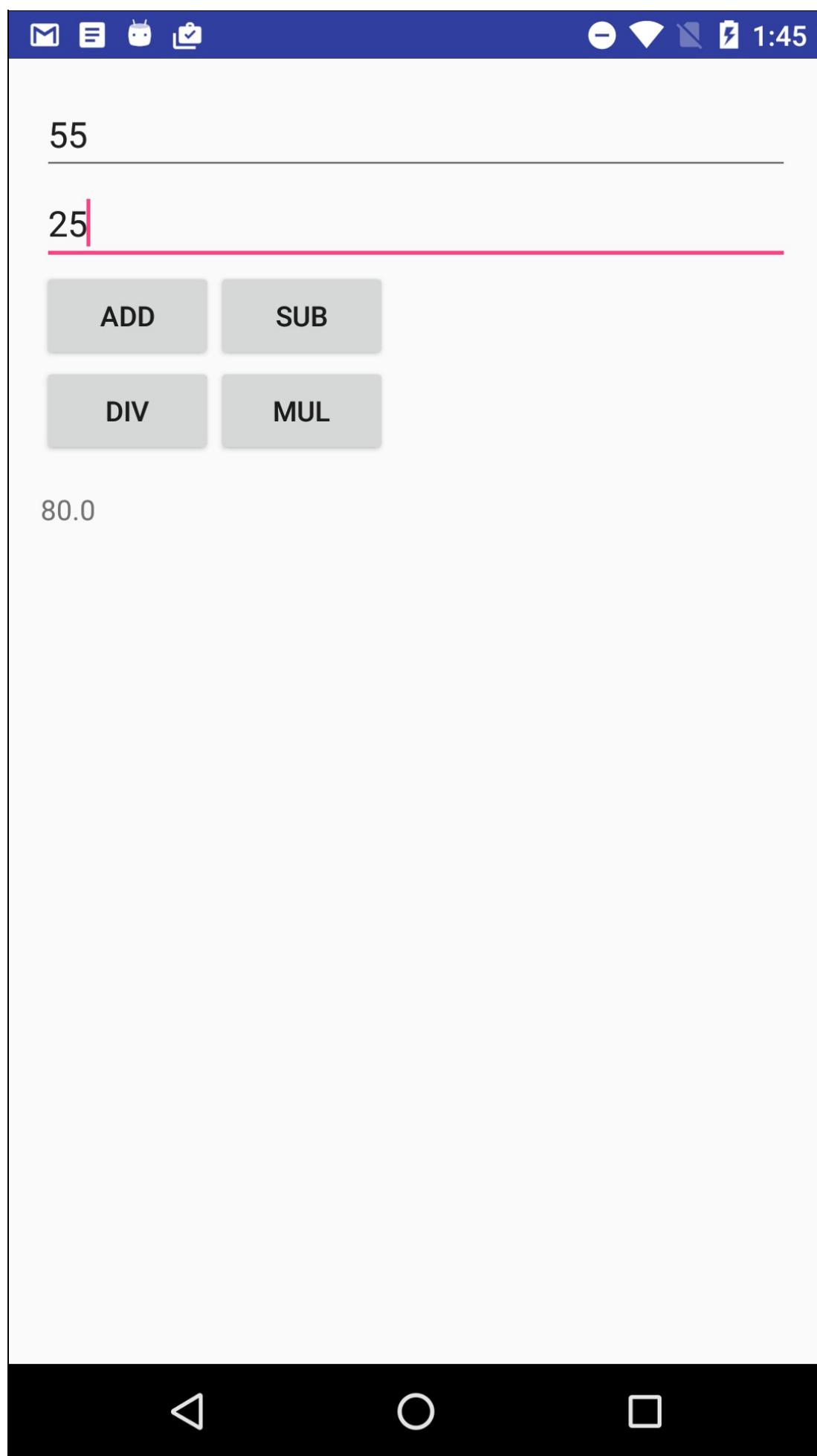
What you will DO

In this practical, you will:

- 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 [SimpleCalc](#). 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

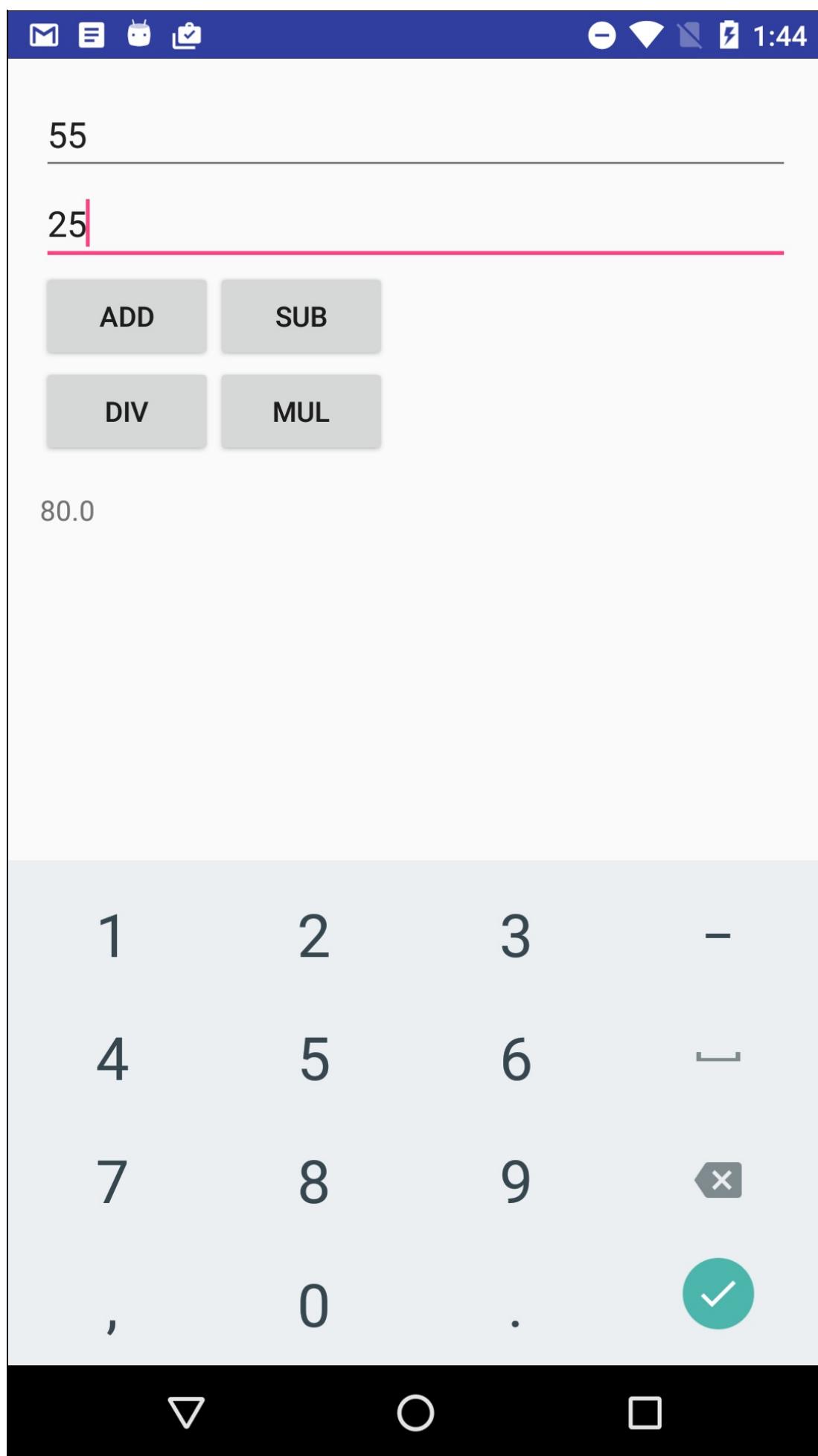
1. Download and unzip the [SimpleCalc](#) project folder.
2. Start Android Studio and select **File > Open**.
3. Navigate to the folder for SimpleCalc, select that folder file, and click **OK**.

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 which may include crashes in the app.

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 `TextView`s 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.



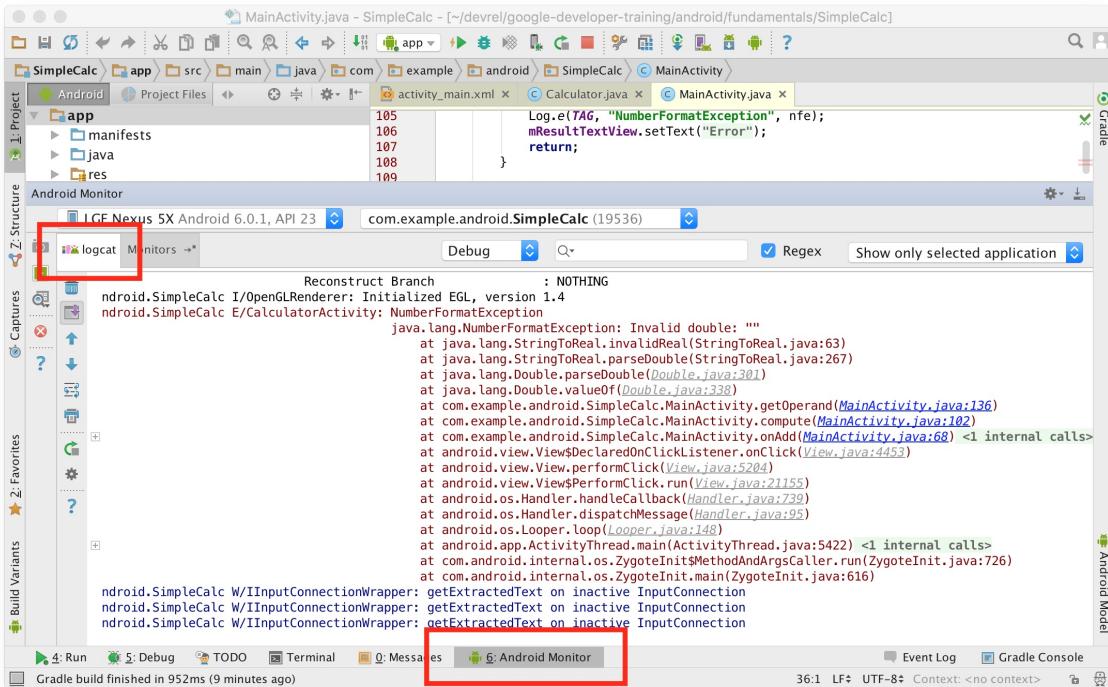
1.3 Explore the app code

1. Expand the app/java folder in the Android project view. 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. Upon examination, you can make the following observations:
 - 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. What observations can you make about the code and activity? Think about your answer and confirm the following:
 - 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 of 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 occurred.

If one or both of 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 occurred.



Coding Challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: Examine the stack trace and try to figure out what caused the error (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

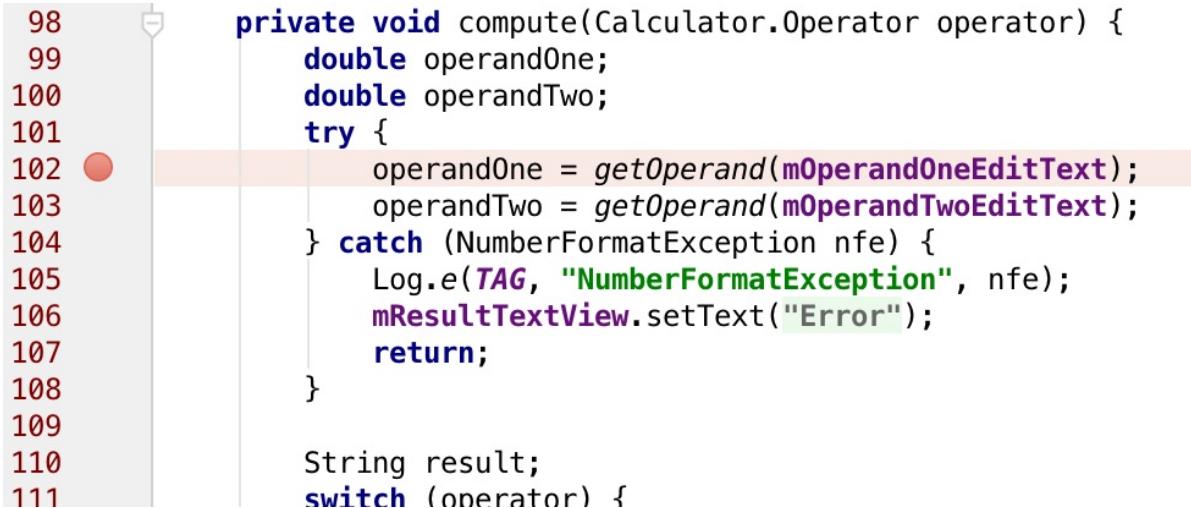
1. In Android Studio, select **Run > Debug app** or click the **Debug icon**  in the toolbar.

If your app is already running, you will be asked if you want to restart your app in debug mode. Click **Restart app**.

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).
3. 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 or clear a breakpoint at a line.

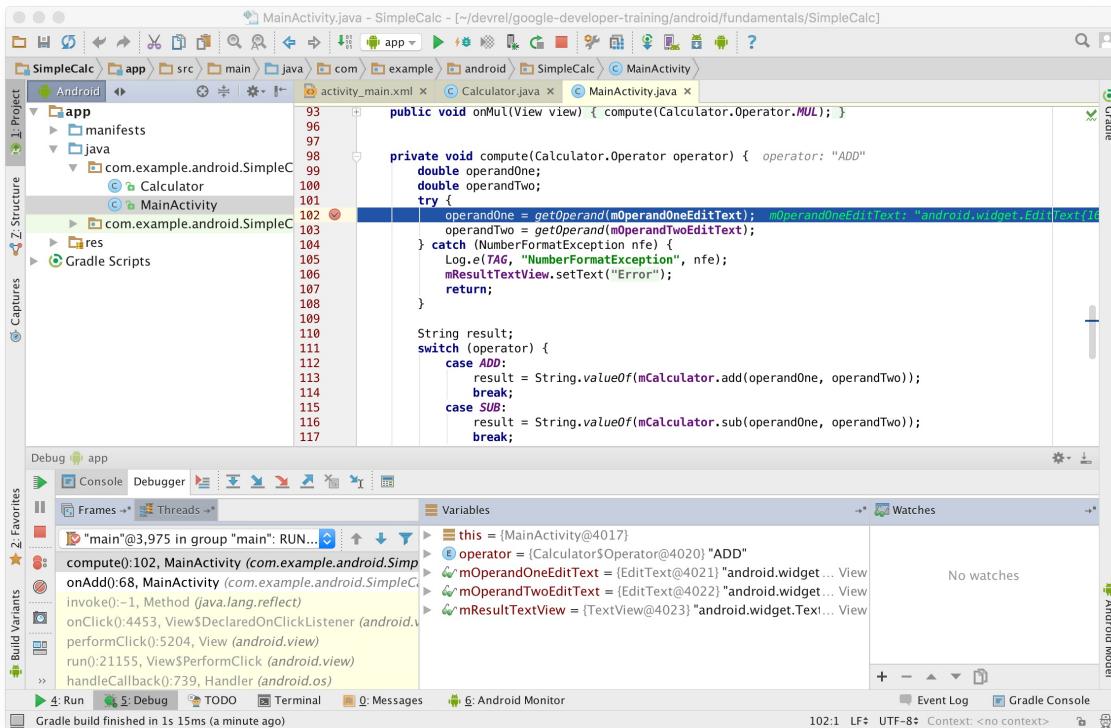


The screenshot shows the Java code for the compute() method. Line 102, which contains the opening brace of the try block, has a red circular breakpoint icon in the gutter. The code is as follows:

```
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  
110    String result;  
111    switch (operator) {
```

4. 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.



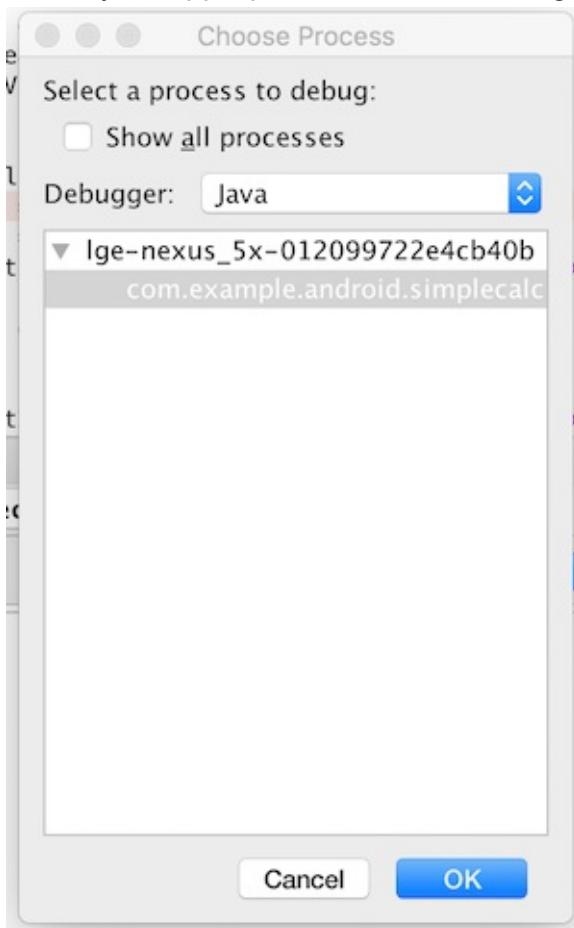
- Examine the the Debug window. It includes these parts:
- Frames panel:** shows the current execution stack frames for a given thread. The execution stack shows each class and method that have been called in your app and in the Android runtime, with the most recent method at the top. Threads appear in a drop down menu. Your app is currently running in the main thread, and that the app is executing the compute() method in MainActivity.
- Variables panel:** displays the variables in the current 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` that 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.
- 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.
- 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 that you set earlier, and the debugger shows the current state of the app. The current line is highlighted in your code.

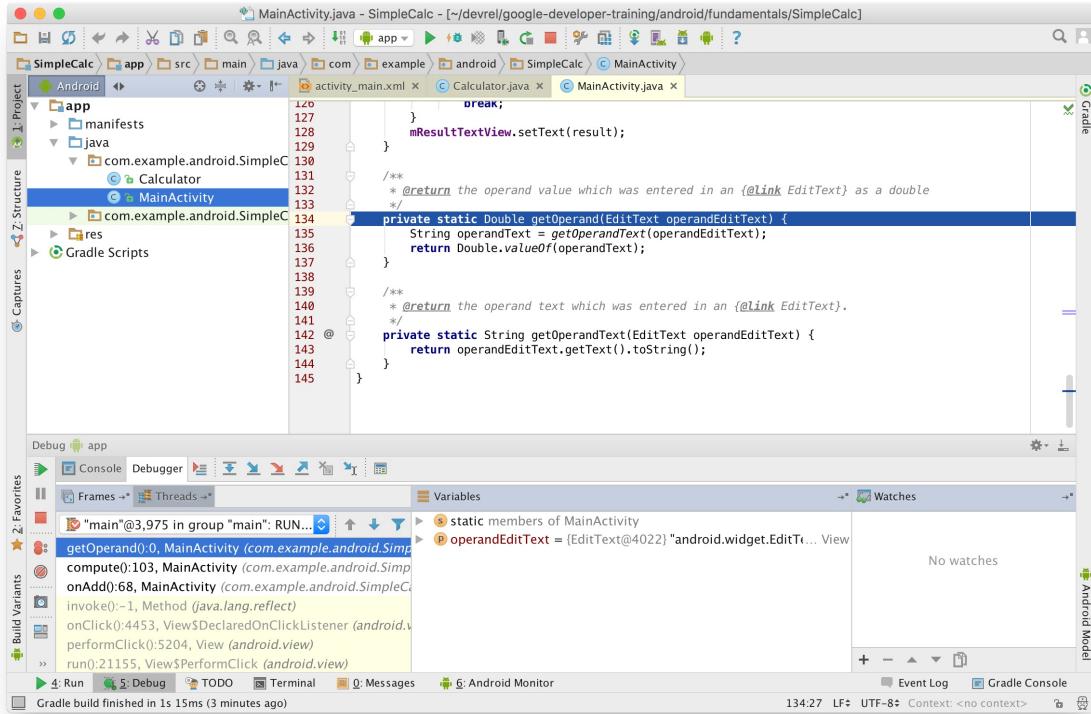
3. Click the **Step Over**  button at the top of the debugger window.

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.

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.
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 is often 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.

The Android Studio debugger lets you examine the state of the variables in your app as that app runs.

1. Run the SimpleCalc app in debug mode if it is not already running.
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 Code Fragment 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`.

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

Coding challenge

Note: All coding challenges are optional and not prerequisites for later lessons.

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.

Summary

- View logging information in Android Studio with the logcat tab of the Android Monitor pane.
- Run your app in debug mode by clicking the debug icon or choosing **Run > Debug app**.
- A breakpoint is a place in your code where you want to pause normal execution of your app to perform other actions. Set or clear a debugging breakpoint by clicking in the left gutter of the editor window immediately next to the target line
- The Debug window in Android Studio shows (stack) Frames, Variables in that frame and Watches (active tracking of a variable while the program runs).

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Debugging Your App](#)

Learn more

- [Debug Your App \(Android Studio User Guide\)](#)
- [Debugging and Testing in Android Studio \(video\)](#)

3.2: 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 SimpleCalc in Android Studio
- Task 2. Add more unit tests to CalculatorTest
- Coding challenges
- Summary
- Related Concept
- Learn More

Testing your code can help you catch bugs early on in development — when they are the least expensive to address — and improve the robustness of your code as your app gets larger and more complex. With tests in your code, you can exercise small portions of your app in isolation, and in an automatable and repeatable manner.

Android Studio and the Android Testing Support Library support several different kinds of tests and testing frameworks. In this practical you'll explore Android Studio's built-in functionality for testing, and learn how to write and run local unit 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" or stub) objects that pretend to behave like the framework equivalents. Unit tests are written with JUnit, a common unit testing framework for Java.

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 practical. You can modify that app in place, or [copy your project](#) into a new app.

Task 1. Explore and run SimpleCalc 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 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 SimpleCalc

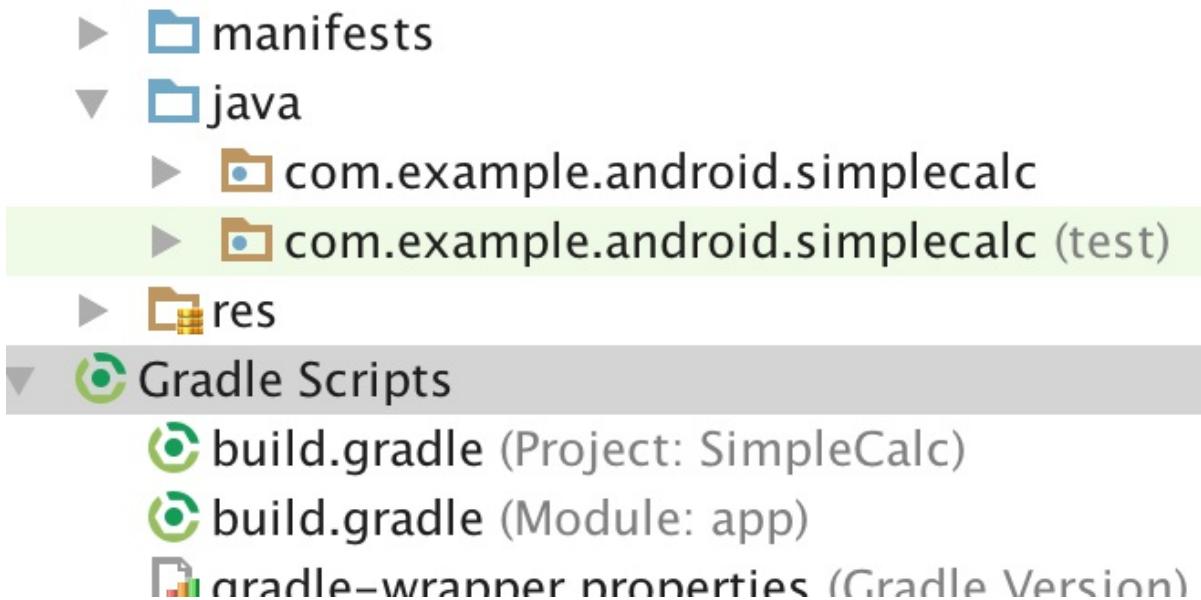
Source sets are a collection of related code in your project that are for different build targets or other "flavors" of your app. When Android Studio creates your project, it creates three source sets:

- 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.

The java folder in the Android view lists all the source sets in the app by package name (com.android.example.simplecalc), with **test** and **androidTest** shown in parentheses after the package name. In the SimpleCalc app, only the main and test source sets are used.



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:

- The only imports are from the org.junit, org.hamcrest, and android.test packages. There are no dependencies on the Android framework classes here.
- 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.
- 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.
- 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 assigns it to the `mCalculator` member variable.

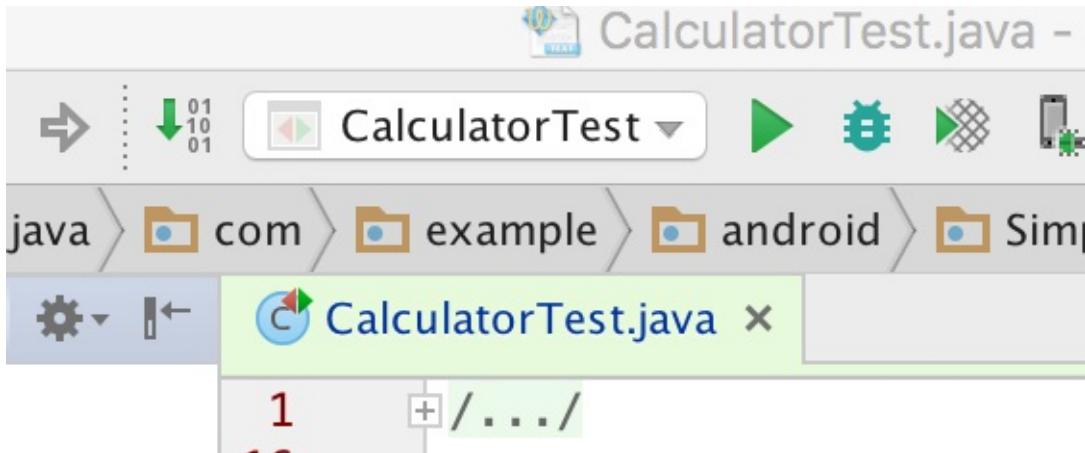
- The `addTwoNumbers()` method is an actual test, and is annotated with `@Test`. Only methods in a test class that have an `@Test` annotation are considered tests to the test runner. Note that by convention test methods do not include the word "test."
- The first line of `addTwoNumbers()` calls the `add()` method from the `Calculator` class. 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.
- The second line is the assertion for the test. Assertions are expressions that must evaluate and result in 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

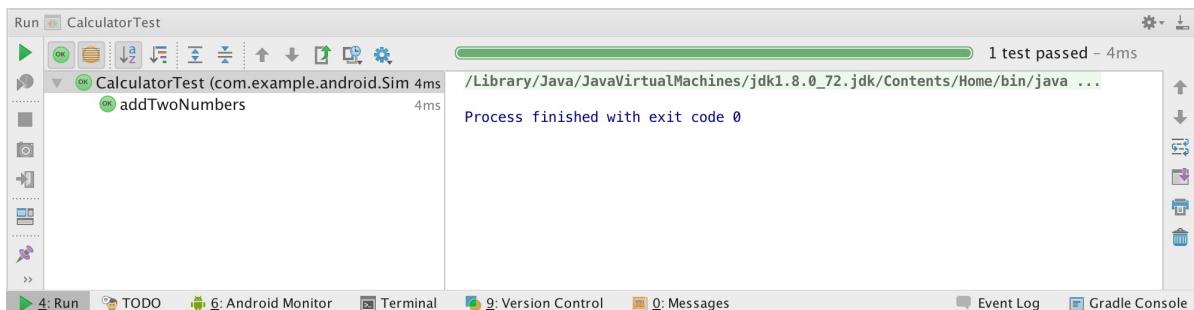
In this task you'll run the unit tests in the test folder and view the output for both successful and failed tests.

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 execution 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:

```
assertThat(resultAdd, is(equalTo(3d)));
```

3. In the run configurations dropdown at the top of the screen, select **CalculatorTest** (if it is not already selected) and click **Run** .

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.

4. Change the assertion in `addTwoNumbers()` back to the correct test and run your tests again to ensure they pass.
5. In the run configurations dropdown, select **app** to run your app normally.

Task 2. Add more unit tests to `CalculatorTest`

With unit testing, you take a small bit of code in your app such as a method or a class, and isolate it from the rest of your app, so that the tests you write make sure that one small bit of the code works in the way you'd expect. Typically unit tests call a method with a variety of different inputs, and verifies that the particular 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.

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.

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.

The `assertThat()` method is used with *matchers*. Matchers are the chained method calls in the second operand of this assertion (`is(equalTo())`). The available matchers you can use to build an assertion are defined by the hamcrest framework (Hamcrest is an anagram for matchers.) Hamcrest provides many basic matchers for most basic assertions. You can also define your own custom matchers for more complex assertions.

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. However, if your app was written such that the 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.

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.

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)));
}
```

Solution code

Android Studio project: [SimpleCalcTest](#)

Coding challenges

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge 1: Dividing by zero is always worth testing for, because it's 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 defined: Dividing a number by zero 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.

Challenge 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 mock 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

- Android Studio has built-in features for running local unit tests.
 - Local unit tests use the JVM of your local machine and do not use the Android framework.
 - Unit tests are written with JUnit, a common unit testing framework for Java.
 - The "test" folder (Android Studio's Project View) is where JUnit tests are located.
 - Local unit tests only need the packages: `org.junit`, `org.hamcrest` and `android.test`
 - The `@RunWith(JUnit4.class)` annotation tells the test runner to run tests in this class.
 - `@SmallTest`, `@MediumTest`, and `@LargeTest` annotations are conventions that make it easier to bundle similar groups of tests
 - The `@SmallTest` annotation indicates all the tests in a class are unit tests that have no dependencies and run in milliseconds.
- Instrumented tests are tests that run on an Android device or emulator.
 - Instrumented tests have access to the Android framework.
- A test runner is a library or set of tools that enables testing to occur and the results to be printed to the log.

Related Concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Testing Your App](#)

Learn More

- [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: 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](#)
- [Task 2. Implement button behavior](#)
- [Coding challenge](#)
- [Summary](#)
- [Related Concept](#)
- [Learn More](#)

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.
- Additional layout and user interface elements
- Support for different device form factors, such as TV or wearables
- Components to support material design elements
- Various other features such as palette support, annotations, percentage-based layout dimensions, and preferences.

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 build files).

What you will LEARN

- How to verify that the the Android support libraries are available in Android Studio.

- How to indicate support library classes in your app.
- How to tell the difference between the values for compileSdkVersion, targetSdkVersion, and minSdkVersion.
- How to recognize deprecated or unavailable APIs in your code.
- Where to find more information on the Android support libraries.

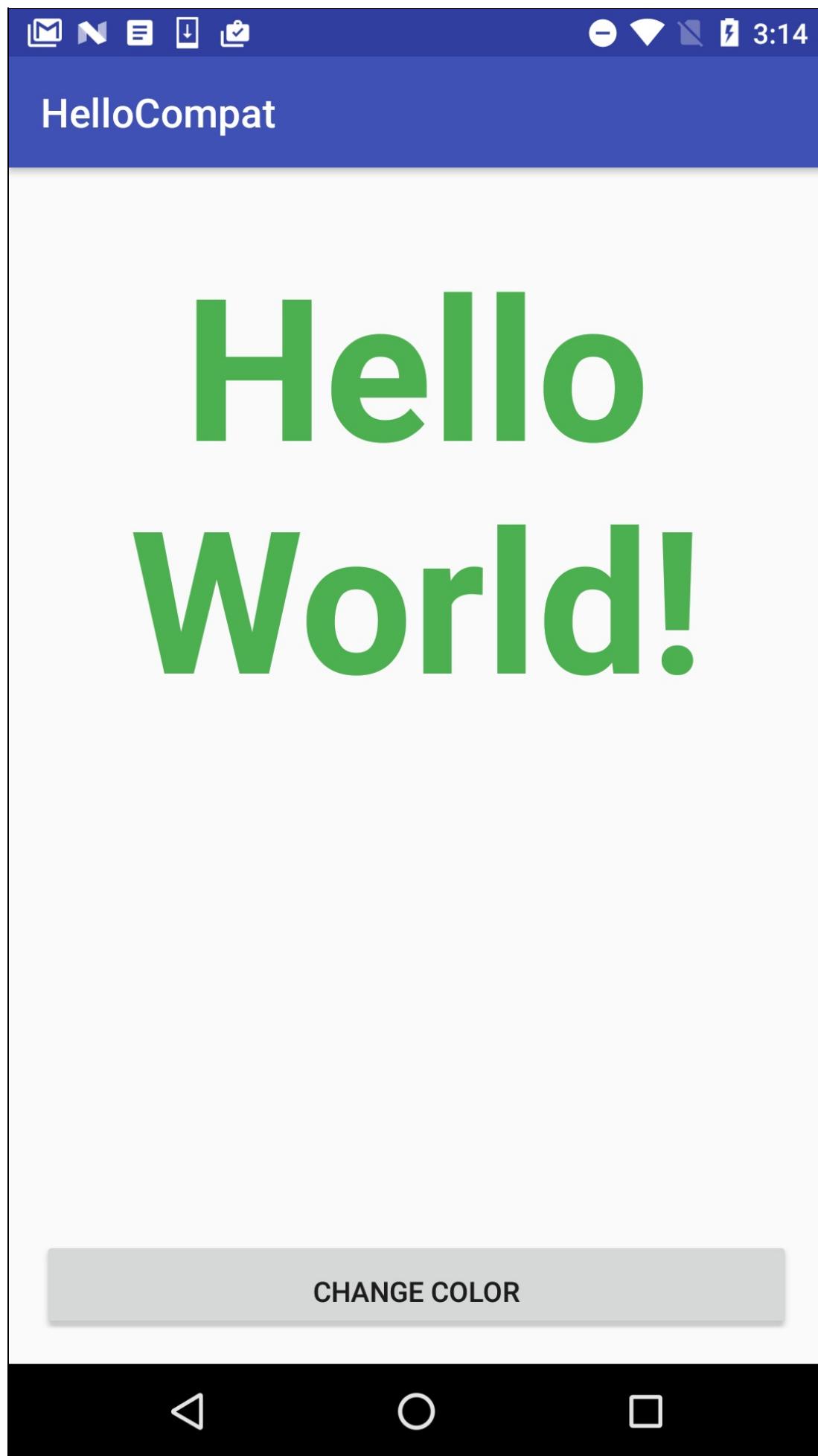
What you will DO

In this practical, you will:

- Create a new app with one textview and one button.
- Verify that the Android Support Library is available on your system.
- Explore the build.gradle for your app.
- Manage class or method calls that are unavailable for the version of Android your app supports.
- Use a compatibility class from the support library to provide backward-compatibility for your app.

App Overview

In this practical you'll create an app called HelloCompat with one textview that displays "Hello World" on the screen, and one button, that changes the color of the text. There are 20 possible colors, defined as resources in the color.xml file, and each button click randomly picks one of those colors.



The methods to get a color value from the app's resources have changed with different versions for the Android framework. This example uses the ContextCompat class, part of the Android support library, which allows you to use a method that works for all versions.

Task 1. Set up the project

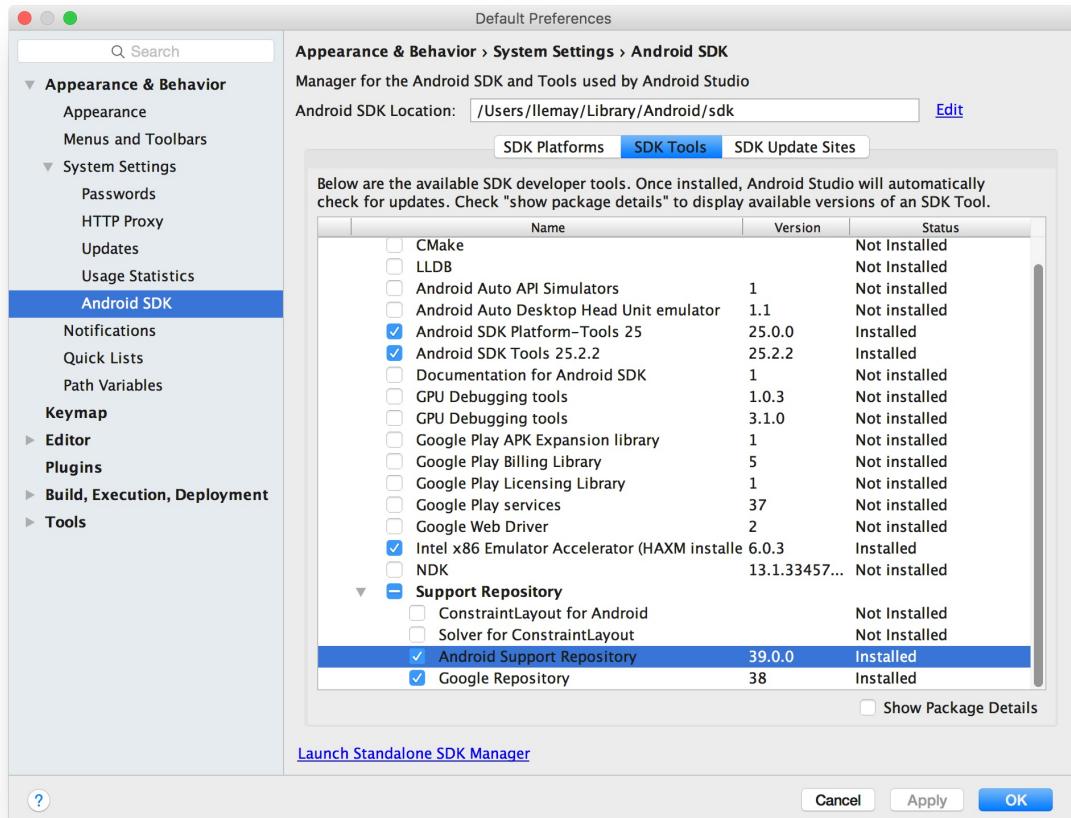
For this task you'll set up a new project for the HelloCompat app and implement the layout and basic behavior.

1.1 Verify that the Android Support Library is available

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. In Android Studio, select **Tools > Android > SDK Manager**, or click the SDK Manager icon.

The SDK Manager preference pane appears.

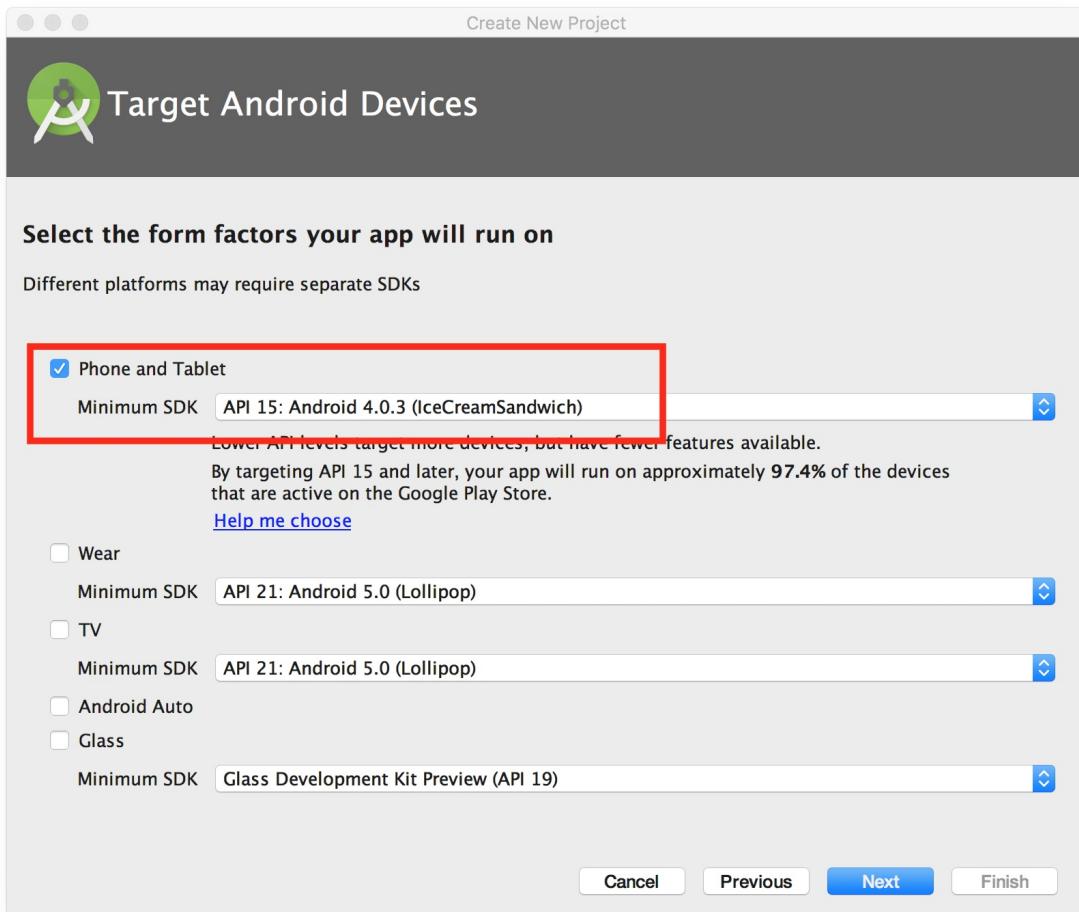


2. Click the **SDK Tools** tab and expand Support Repository.
3. Look for **Support Repository** in the list.
4. If **Installed** appears in the Status column, you're all set. Click **Cancel**.
5. If **Not installed** or **Update Available** appears, click the checkbox next to Android Support Repository. A download icon should appear next to the checkbox. Click **OK**.
6. Click **OK** again, and then **Finish** when the support repository has been installed.

1.2 Set up the Project and examine build.gradle

1. Create a new project called **HelloCompat**, and choose the Empty Activity template.

On the Target Android Devices page, note that API 15: Android 4.0.3 (IceCreamSandwich) is selected for the minimum SDK. As you've learned in previous lessons, this is the oldest version of the Android platform your app will support.



2. In Android Studio, make sure the **Project** pane is open and the **Android** tab is clicked.
3. 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.

4. Locate the `compileSdkVersion` line near the top of the file.

```
compileSdkVersion 24
```

The `compile` version is the Android framework version your app is compiled with in Android Studio. For new projects the compile version is the most recent set of framework APIs you have installed. This value affects *only* Android Studio itself and the warnings or errors you get in Android Studio if you use older or newer APIs.

5. Locate the `minSdkVersion` line in the `defaultConfig` section a few lines down.

```
minSdkVersion 15
```

The *minimum* version is the oldest Android API version your app runs under. It's the same number you chose in Step 1 when you created your project. The Google Play store uses this number to make sure your app can run on a given user's device. Android Studio also uses this number to warn you about using deprecated APIs.

6. Locate the `targetSdkVersion` line in the `defaultConfig` section.

```
targetSdkVersion 24
```

The *target* version indicates the API version your app is designed and tested for. If the API of the Android platform is higher than this number (that is, your app is running on a newer device), the platform may enable compatibility behaviors to make sure your app continues to work the way it was designed to. For example, Android 6.0 (API 23) provides a new runtime permissions model. If your app targets a lower API level, the platform falls back to the older install-time permissions model.

Although the target SDK can be the same number as the compile SDK, it is often a lower number that indicates the most recent version of the API for which you have tested your app.

7. Locate the `dependencies` section of `build.gradle`, near the end of the file.

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile(
        'com.android.support.test.espresso:espresso-core:2.2.2', {
            exclude group: 'com.android.support',
            module: 'support-annotations'
        })
    compile 'com.android.support:appcompat-v7:24.2.1'
    testCompile 'junit:junit:4.12'
}
```

The dependencies section for a new project includes several dependencies to enable testing with Espresso, JUnit, as well as the v7 appcompat support library. Note that the version numbers for these libraries in your project may be different than those shown here.

The v7 appcompat support library provides backward-compatibility for older versions of Android all the way back to API 9. It includes the v4 compat library as well, so you don't need to add both as a dependency.

8. Update the version numbers, if necessary.

If the current version number for a library is lower than the currently available library version number, Android Studio will highlight the line and warn you that a new version is available. ("a newer version of com.android.support:appcompat-v7 is available"). Edit the version number to the updated version.

Tip: You can also click anywhere in the highlight line and type **Alt-Enter** (**Option-Enter** on the Mac). Select "Change to XX.X.X" from the menu, where XX.X.X is the most up to date version available.

9. Update the compileSdkVersion number, if necessary.

The major version number of the support library (the first number) must match your compileSdkVersion. When you update the support library version you may also need to update compileSdkVersion to match.

10. Click **Sync Now** to sync your updated gradle files with the project, if prompted.
11. Install missing SDK platform files, if necessary.

If you update compileSdkVersion you may need to install the SDK platform components to match. Click **Install missing platform(s) and sync project** to start this process.

1.3 Add the layout and colors

In this task, modify the layout for the app.

1. Open `res/layout/activity_main.xml`. In the Layout Editor, click the Text tab to view the XML editor.
2. Modify the TextView element to have these attributes:

Attribute	Value
android:id	"@+id/hello_textview"
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:padding	"@dimen/activity_horizontal_margin"
android:gravity	"center"
android:textSize	"100sp"
android:textStyle	"bold"
android:text	"Hello World!"

3. Extract the string for "Hello World" into a string resource.
4. Add a Button view under the TextView, and add these attributes:

Attribute	Value
android:id	"@+id/color_button"
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:layout_alignParentBottom	"true"
android:paddingTop	"@dimen/activity_vertical_margin"
android:text	"Change Color"
android:onClick	"changeColor"

5. Extract the string for "Change Color" into a string resource.
6. Open `res/values/colors.xml`.
7. Add these color resources to the file:

```
<color name="red">#F44336</color>
<color name="pink">#E91E63</color>
<color name="purple">#9C27B0</color>
<color name="deep_purple">#673AB7</color>
<color name="indigo">#3F51B5</color>
<color name="blue">#2196F3</color>
<color name="light_blue">#03A9F4</color>
<color name="cyan">#00BCD4</color>
<color name="teal">#009688</color>
<color name="green">#4CAF50</color>
<color name="light_green">#8BC34A</color>
<color name="lime">#CDDC39</color>
<color name="yellow">#FFEB3B</color>
<color name="amber">#FFC107</color>
<color name="orange">#FF9800</color>
<color name="deep_orange">#FF5722</color>
<color name="brown">#795548</color>
<color name="grey">#9E9E9E</color>
<color name="blue_grey">#607D8B</color>
<color name="black">#000000</color>
```

These color values and names come from the recommended color palettes for Android apps defined at [Material Design - Style - Color](#). The codes indicate color RGB values in hexadecimal.

Solution Code (`activity_main.xml`)

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.android.hellocompat.MainActivity">

    <TextView
        android:id="@+id/hello_textview"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:padding="@dimen/activity_horizontal_margin"
        android:text="@string/hello_text_string"
        android:textSize="100sp"
        android:textStyle="bold" />

    <Button
        android:id="@+id/color_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:onClick="changeColor"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:text="@string/button_label" />
</RelativeLayout>
```

1.4 Add behavior to MainActivity

In this task you'll finish setting up the project by adding private variables and implementing `onCreate()` and `onSaveInstanceState()`.

1. Open `MainActivity.java`.
2. Add a private variable at the top of the class to hold the `TextView` object for the Hello World textview.

```
private TextView mHelloTextView;
```

3. Add the following color array just after the private variable:

```
private String[] mColorArray = {"red", "pink", "purple", "deep_purple",
    "indigo", "blue", "light_blue", "cyan", "teal", "green",
    "light_green", "lime", "yellow", "amber", "orange", "deep_orange",
    "brown", "grey", "blue_grey", "black"
};
```

Each of these color names correspond to the names of the color resources from color.xml.

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

```
mHelloTextView = (TextView) findViewById(R.id.hello_textview);
```

5. Also in onCreate(), restore the saved instance state, if any:

```
// restore saved instance state (the text color)
if (savedInstanceState != null) {
    mHelloTextView.setTextColor(savedInstanceState.getInt("color"));
}
```

6. Add the onSaveInstanceState() method to MainActivity to save the text color:

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

    // save the current text color
    outState.putInt("color", mHelloTextView.getCurrentTextColor());
}
```

Solution Code (not the whole class)

```
// Text view for Hello World.  
private TextView mHelloTextView;  
// array of color names, these match the color resources in color.xml  
private String[] mColorArray = {"red", "pink", "purple", "deep_purple",  
    "indigo", "blue", "light_blue", "cyan", "teal", "green",  
    "light_green", "lime", "yellow", "amber", "orange", "deep_orange",  
    "brown", "grey", "blue_grey", "black"  
};  
...  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    // Initialize the main text view  
    mHelloTextView = (TextView) findViewById(R.id.hello_textview);  
  
    // restore saved instance state (the text color)  
    if (savedInstanceState != null) {  
        mHelloTextView.setTextColor(savedInstanceState.getInt("color"));  
    }  
}  
...  
@Override  
public void onSaveInstanceState(Bundle outState) {  
    super.onSaveInstanceState(outState);  
  
    // save the current text color  
    outState.putInt("color", mHelloTextView.getCurrentTextColor());  
}
```

Task 2. Implement button behavior

The Change Color button in the HelloCompat app picks one of the 20 colors from the color.xml resource file at random and sets the color of the text to that color. In this task you'll implement the onClick() behavior for this handler.

2.1 Add the changeButton() onClick handler

1. Open `res/layout/activity_main.xml`, if it is not already open.
2. Click anywhere in the `android:onClick` attribute, inside the Button element.
3. Press **Alt-Enter** (**Option-Enter** on the Mac), and select **Create onClick event handler**.
4. Choose **MainActivity** and click **OK**.

This creates a placeholder method stub for the `changeColor()` method in `MainActivity.java`.

2.2 Implement the button action

1. Open MainActivity.java, if it is not already open.
2. In the changeColor() method, create a random number object.

```
Random random = new Random();
```

Use the Random class (a Java class) to generate simple random numbers.

3. Use the random instance to pick a random color from the mColorArray array:

```
String colorName = mColorArray[random.nextInt(20)];
```

The nextInt() method with the argument 20 gets another random integer between 0 and 19. Use that integer as the index of the array to get a color name.

4. Get the resource identifier (an integer) for the color name out of the resources:

```
int colorResourceName = getResources().getIdentifier(colorName,  
        "color", getApplicationContext().getPackageName());
```

When your app is compiled, the Android system converts the definitions in your XML files into resources with internal integer IDs. There are separate IDs for both the names and the values. This line matches the color strings from the colorName array with the corresponding color name IDs in the XML resource file. The getResources() method gets all the resources for your app. The getIdentifier() method looks up the color name (the string) in the color resources ("color") for the current package name.

5. Get the integer ID for the actual color from the resources and assign it to a colorRes variable:

```
int colorRes = getResources().getColor(colorResourceName);
```

The getResources() method gets the set of resources for your app, and the getColor() method retrieves a specific color from those resources by the ID of the color name.

Note that the getColor() method appears with a strikethrough in the Android Studio editor. If you hover your mouse over getColor(), the error "getColor(int) is deprecated" appears. In API 23, the getColor() method was modified to include a second argument for the app's theme. Since your app has a compileSdkVersion of 24 (or higher), Android Studio provides this warning that you're using an older, deprecated method.

You can still compile your app and it would still run on both new and old Android devices -- the deprecation warning is a warning, not an error. But it's best not to ignore warnings where they exist, as deprecated methods can result in unexpected behavior.

6. Change the colorRes assignment line to include the second argument to getColor():

```
int colorRes =  
    getResources().getColor(colorResourceName, this.getTheme());
```

You can use the getTheme() method to get the theme for the current application context. Only now with this change you'll note that getColor() has a red underlined highlight. If you hover over getColor() Android Studio reports: "Call requires API 23 (current min is 15)". Since your minSdkVersion is 15, you'll get this message if you try to use any APIs that were introduced after API 15. You can still compile your app, but because this new version of getColor() with two arguments is not available on devices prior to API 23, your app will crash when the user taps the Change button.

At this stage you could check for the platform version and use the right version of getColor() depending on where the app is running. (you will still get a warning for both calls in Android Studio). The better way to support both older and newer Android APIs without warnings is to use one of the compatibility classes in the support library.

7. Change the colorRes assignment line to use the ContextCompat class:

```
int colorRes = ContextCompat.getColor(this, colorResourceName);
```

ContextCompat provides many compatibility methods to address API differences in the application context and app resources. The getColor() method in ContextCompat takes two arguments: the current context (here, the activity instance, this), and the name of the color.

The implementation of this method in the support library hides the implementation differences in different versions of the API. You can call this method regardless of your compile SDK or minimum SDK versions with no warnings, errors, or crashes.

8. Set the color of the Hello World text view to the color resource ID:

```
mHelloTextView.setTextColor(colorRes);
```

9. Compile and run the app on a device or emulator.

The Change Color button should now change the color of the text in Hello World.

Solution Code (changeColor() method only)

```
public void changeColor(View view) {  
    // get a random color name from the color array (20 colors)  
    Random random = new Random();  
    String colorName = mColorArray[random.nextInt(20)];  
  
    // get the color identifier that matches the color name  
    int colorResourceName = getResources().getIdentifier(colorName, "color",  
        getApplicationContext().getPackageName());  
  
    // get the color ID from the resources  
    int colorRes = ContextCompat.getColor(this, colorResourceName);  
  
    // Set the text color  
    mHelloTextView.setTextColor(colorRes);  
}
```

Solution code

Android Studio project: [HelloCompat](#)

Coding Challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: Instead of using ContextCompat for to get the color resource, use a test of the values in the [Build](#) class to perform a different operation if the app is running on a device that supports less than API 23.

Summary

In this practical, you learned that:

- Android uses three directives to indicate how your app should behave for different API versions:
 - minSdkVersion: the minimum API version your app supports.
 - compileSdkVersion: the API version your app should be compiled with.
 - targetSdkVersion: the API version your app was designed for.
- The Android Support Library can be installed in the SDK manager
- You can add library dependencies for support libraries in the gradle.build file
- The ContextCompat class provides methods for compatibility with context and resource-related methods for both old and new API levels.

Related Concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [The Android Support Library](#)

Learn More

- [Android Support Library \(introduction\)](#)
- [Support Library Setup](#)
- [Support Library Features](#)
- [Supporting Different Platform Versions](#)
- [Picking your compileSdkVersion, minSdkVersion, and targetSdkVersion](#)
- [All the Things Compat](#)
- [API Reference \(all packages that start with android.support\)](#)

4.1: 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](#)
- [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](#)
- [Task 6: Use image views as buttons](#)
- [Task 7: Use radio buttons](#)
- [Coding challenge](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

You can customize input methods to make entering data easier for users.

In this practical, you'll learn to:

- Use different on-screen keyboards and controls for user input.
- Show an alert message that users can interact with.
- Provide interface elements for selecting a time and date.
- Use images as buttons to launch an activity.
- Add radio buttons for the user to select one item from a set of items.

What you should already KNOW

For this practical you should be able to:

- Create an Android Studio project from a template and generating the main layout.
- Run apps on the emulator or a connected device.
- Make a copy of an app project, and renaming the app.
- Create and editing UI elements using the Layout Editor and XML code.

- Access UI elements from your code using `findViewById()`.
- Convert the text in a view to a string using `getText().toString()`.
- Handle a button click.
- Display a toast message.
- Start an activity with another app using an implicit intent.
- Use an adapter to connect your data to a view, such as the RecyclerView in a previous lesson.

What you will LEARN

In this practical, you will learn to:

- Change the input methods to enable spelling suggestions, auto-capitalization, and password obfuscation.
- Change the generic on-screen keyboard to a phone keypad or other specialized keyboards.
- Add a spinner input control to show a dropdown menu with values, from which the user can select one.
- Add an alert with OK and Cancel for a user decision.
- Use date and time pickers and recording the selections.
- Use images as buttons to launch an activity.
- Add radio buttons for the user to select one item from a set of items.

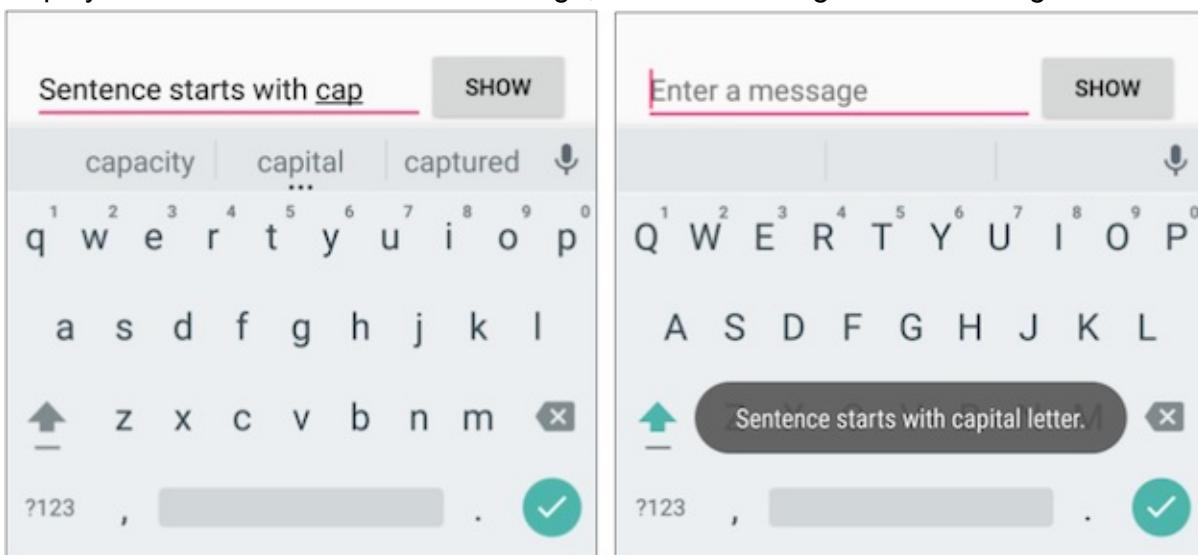
What you will DO

- 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.
- Create a new project to use images as buttons.

- Create a second activity and add radio buttons for selecting an option.
- Set `onClick` handlers for the images used as buttons to launch a second activity.

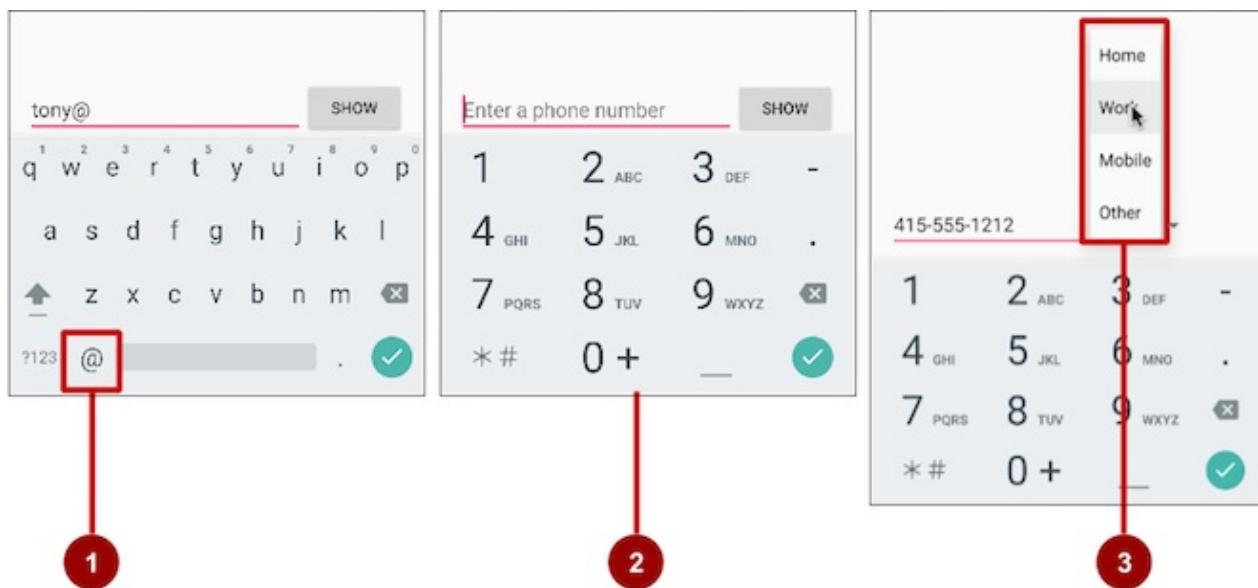
App Overview

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, as shown on the left side of the figure below. To keep the app simple, you'll display the entered text in a `toast` message, shown on the right side of the figure below.



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 shown on the left side and in the center of the figure below. 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.

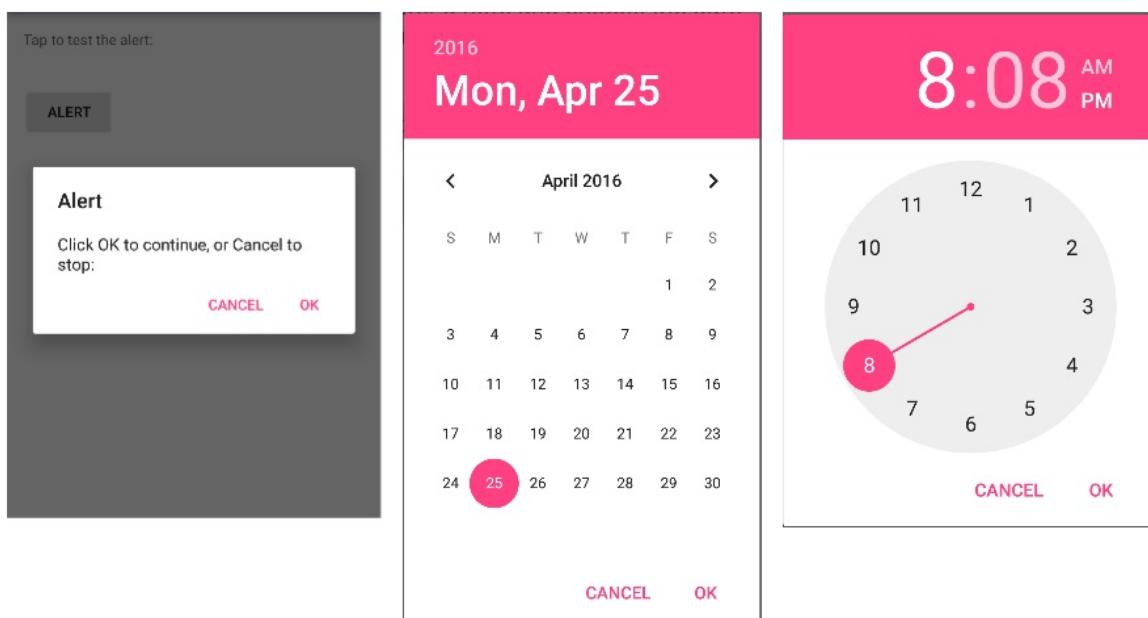
You will then copy the app to create Phone Number Spinner that offers a spinner input control for selecting the label (Home, Work, Other, Custom) for the phone number, as shown on the right side of the figure below.



The figure above shows the following:

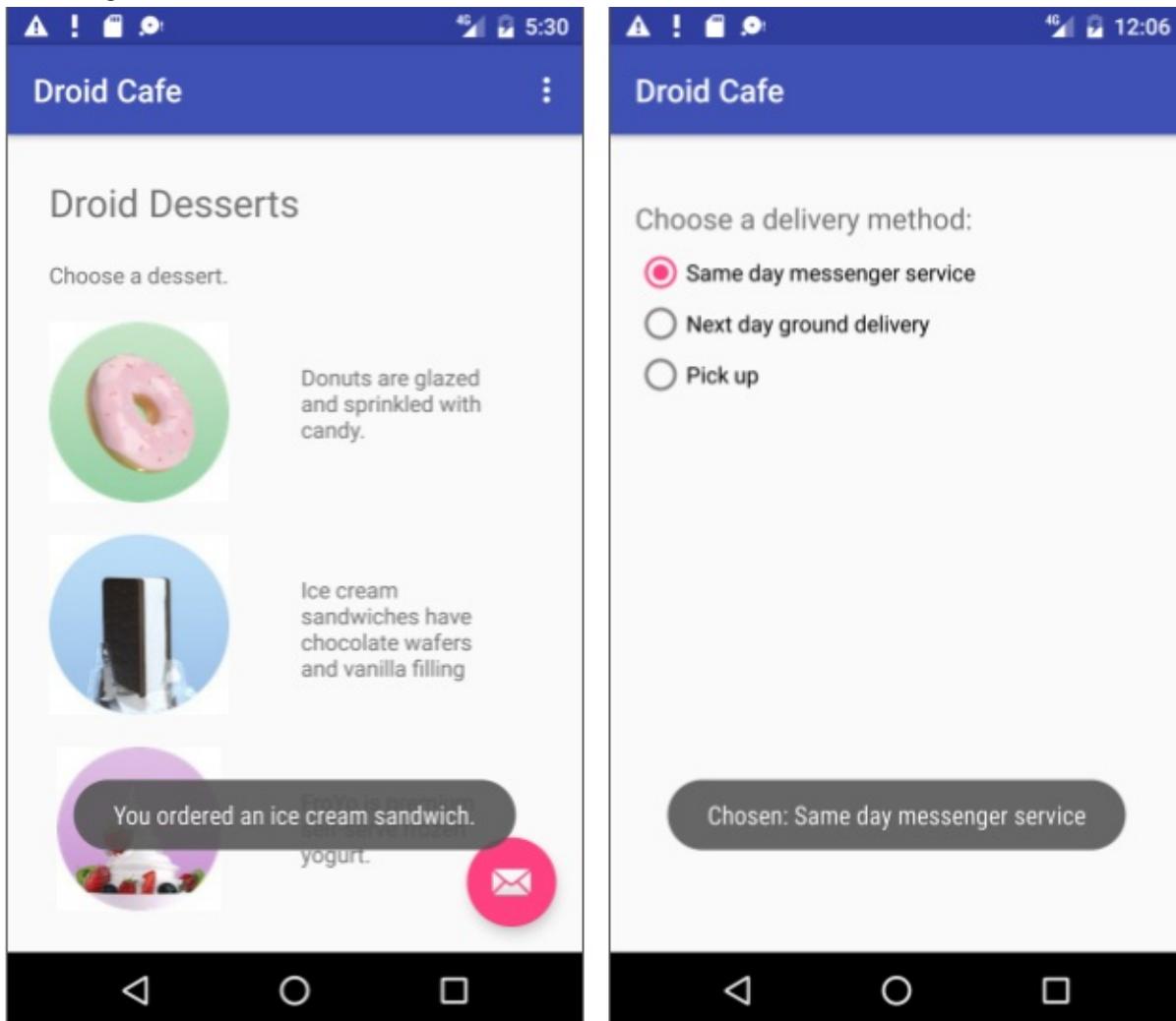
1. The email keyboard with the "@" symbol in an easy-to-find location
2. The phone keypad
3. The spinner

You'll also create Alert Sample to experiment with an alert dialog, shown on the left side of the figure below, and Date Time Pickers to experiment with a date picker and a time picker, shown in the center and on the right side of the figure below, and use the time and date selections in your app.



The last tasks involve creating an app from the Basic Activity template that lets a user tap image buttons to launch an activity, as shown on the left side of the figure below, and choose a single delivery option from radio-button choices for a food order, as shown on the right side

of the figure below.



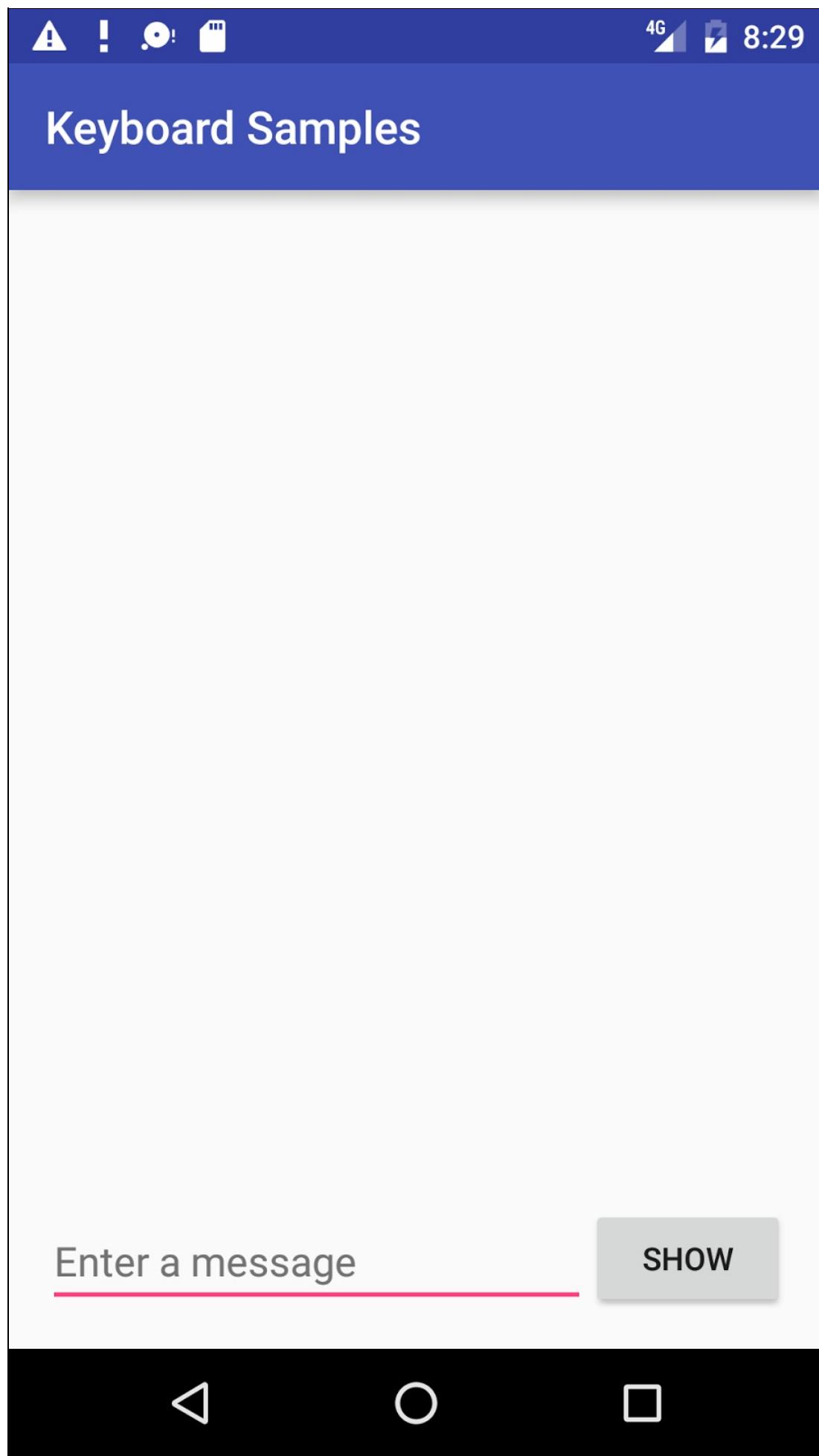
Task 1. Experiment with text entry keyboard attributes

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.

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. The app's layout will look like the following figure.



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 an 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 change the Resource name for the string value to **show**.

You extract string resources because it makes the app project more flexible for changing strings. 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. For example, the "Show" value for the resource named `show` could be changed to "Montrer" for the French version of the app.

5. Change the existing `TextView` element as follows:
 - i. Delete the `android:text` attribute that specified "Hello World!".
 - ii. Change the `TextView` tag to an `EditText` tag, and make sure the ending tag is `/>`.
 - iii. Add or change the following attributes:

EditText Attribute	TextView Old Value	EditText New Value
android:id		"@+id/editText_main"
android:layout_width	"wrap_content"	"match_parent"
android:layout_height	"wrap_content"	"wrap_content"
android:layout_alignParentBottom		"true"
android:layout_toLeftOf		"@+id/button_main"
android:hint		"Enter a message"

You learned about the `android:layout_toLeftOf` and `android:layout_alignParentBottom` attributes in a previous lesson. These layout-related attributes work with the [RelativeLayout](#) view group to position child views relative to each other or to the parent. The `android:hint` attribute sets the text to appear in the field that provides a hint for the user to provide input, such as "Enter a message".

6. Extract the string resource for the `android:hint` attribute value "Enter a message" to the resource name `enter`. The `activity_main.xml` layout file should now look like the following:

```

<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.keyboardsamples.MainActivity">

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

    <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:hint="@string/enter" />

</RelativeLayout>

```

7. Open `MainActivity.java` 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();
    }
}

```

8. Open `strings.xml` (in `app > res > values`), and edit the `app_name` value to "**Keyboard Samples**" (be sure to include a space between "Keyboard" and "Samples").
9. Run the app and examine how the keyboard works.

Tapping the **Show** button shows the 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 in a green circle, shown below, appears in the lower right corner of the keypad. This is known as the Return (or Enter) key, and it is used 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.

```
    android:inputType="textPassword"
```

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](#).

Solution code:

Android Project: [KeyboardSamples](#)

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"
```
2. Change the `android:hint` attribute to "**Enter a phone number**".
3. Extract the string resource for the `android:hint` value to `enter_phone`.
4. 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.

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](#).)

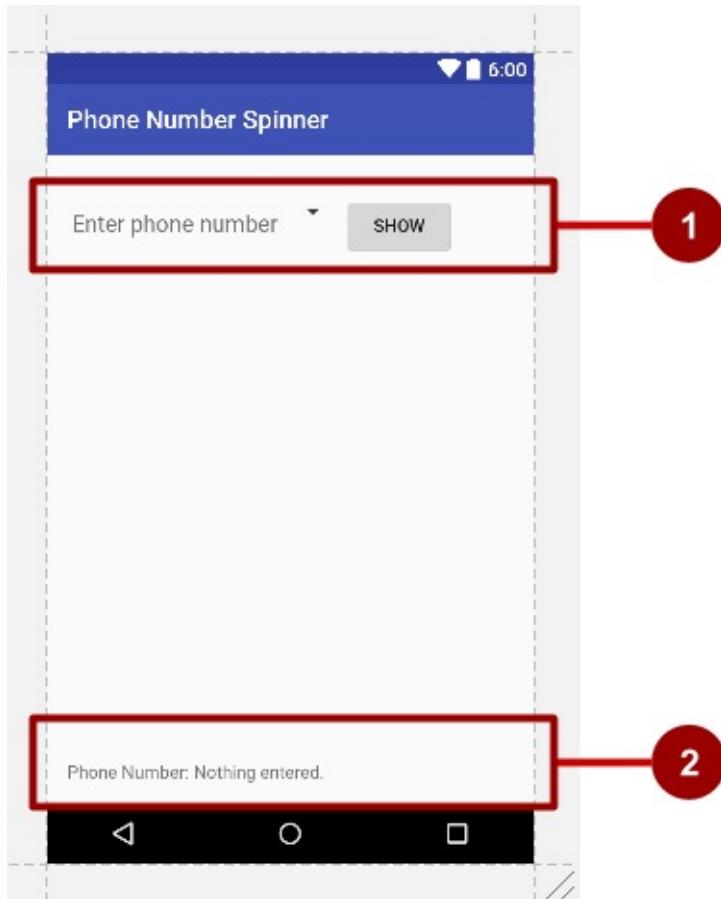
A *spinner* provides a quick way to select one value from a set. Touching the spinner displays a drop-down list with all available values, from which the user can select one. If you are providing only two or three choices, you might want to use radio buttons for the choices if you have room in your layout for them; however, with more than three choices, a spinner works very well, scrolls as needed to display items, and takes up little room in your layout.

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 Copy the KeyboardSamples project and modify the layout

Use the following figure as a guide for the main activity's layout:



In the above figure:

1. The first LinearLayout with an EditText view, a spinner icon, and the **Show** button.
2. The second LinearLayout with two TextViews.

Follow these steps:

1. Copy the **KeyboardSamples** project folder, rename it to **PhoneNumberSpinner**, and *refactor* it to populate the new name throughout the app project. (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 existing `EditText` and `Button` elements from the previous lesson within a `LinearLayout` with a horizontal orientation, placing the `EditText` element above the `Button`:

```

<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:

i. Remove the following attributes from the `EditText` element:

- `android:layout_toLeftOf`
- `android:layout_alignParentBottom`

ii. Remove the following attributes from the `Button` element:

- `android:layout_alignParentRight`
- `android:layout_alignParentBottom`

iii. Change three other attributes of the `EditText` element as follows:

EditText Attribute	Value
<code>android:layout_width</code>	"wrap_content"
<code>android:inputType</code>	"phone"
<code>android:hint</code>	"Enter phone number"

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 task you will add code that will fill the spinner list with values. The layout code for the `EditText`, `Spinner`, and `Button` elements within the `LinearLayout` should now look like this:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="@dimen/activity_vertical_margin"
    android:orientation="horizontal">

    <EditText
        android:id="@+id/editText_main"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:inputType="phone"
        android:hint="Enter phone number" />

    <Spinner
        android:id="@+id/label_spinner"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
    </Spinner>

    <Button
        android:id="@+id/button_main"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="showText"
        android:text="Show" />

</LinearLayout>
```

7. Add another `LinearLayout` below the `LinearLayout` you just created, 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 (refer to the figure above):

```
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:layout_alignParentBottom="true">

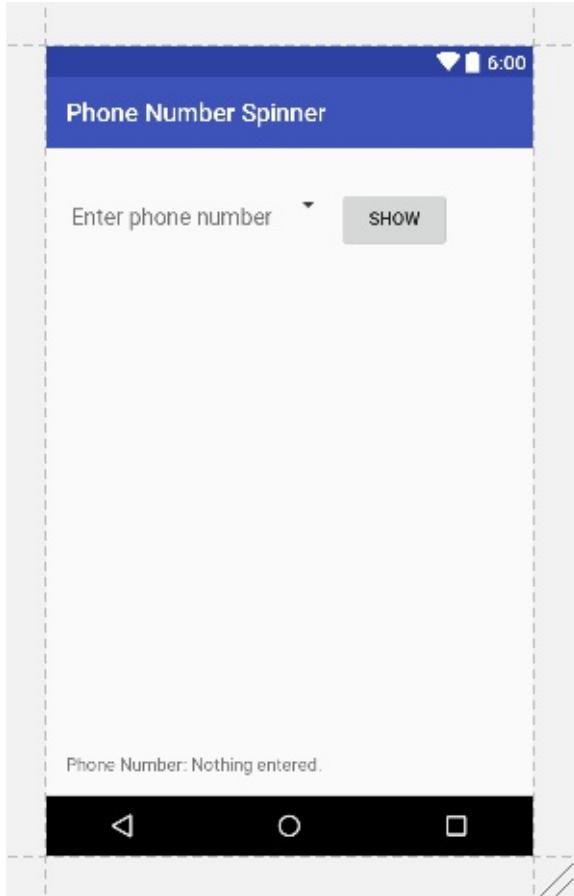
    <TextView
        ...
    <TextView
        ...
    </LinearLayout>
```

8. 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."

9. Check your layout by clicking the Preview tab on the right side of the layout window.



You should now have a screen (refer to the figure above) 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."

10. 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 well-defined static strings ("Home", "Work", etc), so you can use a text array defined in strings.xml to hold the values for it.

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. The reason why you need the `AdapterView` is because you need an adapter—specifically 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. You will learn more about this pattern of using an adapter to connect data in another lesson. This line should appear in your block of import statements:

```
import android.widget.AdapterView;
```

After typing **OnItemSelectedListener** in the above statement, wait a few seconds for a red light bulb to appear in the left margin.

3. 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. Both methods use the parameter `AdapterView<?>`. The `<?>` is a Java type wildcard, enabling the method to be flexible enough to accept any type of `AdapterView` as an argument.

4. Instantiate a spinner object in the `onCreate()` method using the `Spinner` element in the layout (`label_spinner`), and set its listener (`spinner.setOnItemSelectedListener`) in the `onCreate()` method. Add the code to the `onCreate()` method:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    // Create the spinner.  
    Spinner spinner = (Spinner) findViewById(R.id.label_spinner);  
    if (spinner != null) {  
        spinner.setOnItemSelectedListener(this);  
    }  
    ...  
}
```

5. Continuing to edit the `onCreate()` method, add a statement that creates the `ArrayAdapter` with the string array (`labels_array`) using the Android-supplied 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);  
...
```

The `simple_spinner_item` layout used in this step, and the `simple_spinner_dropdown_item` layout used in the next step, are the default pre-defined layouts provided by Android in the `R.layout` class. You should use these layouts unless you want to define your own layouts for the items in the spinner and the spinner's appearance.

6. Specify the layout for the spinner's choices to be `simple_spinner_dropdown_item`, and then apply the adapter to the spinner:

```

...
// Specify the layout to use when the list of choices appears.
adapter.setDropDownViewResource
        (android.R.layout.simple_spinner_dropdown_item);
// Apply the adapter to the spinner.
if (spinner != null) {
    spinner.setAdapter(adapter);
}
...

```

3.3 Add code to respond to the user's selections

When the user selects an item in the spinner, the `Spinner` object receives an on-item-selected event. To handle this event, you already implemented the `AdapterView.OnItemSelectedListener` interface in the previous step, adding empty `onItemSelected()` and `onNothingSelected()` callback methods.

In this step you will first declare `mSpinnerLabel` as the string to hold the selected item. You will then fill in the code for the `onItemSelected()` method to retrieve the selected item in the spinner, using `getItemAtPosition()`, and assign the item to `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`. You can also add a call to the `showText()` method you already added to the previous version of the app:

```

public void onItemSelected(AdapterView<?> adapterView, View view, int
        i, long l) {
    mSpinnerLabel = adapterView.getItemAtPosition(i).toString();
    showText(view);
}
```

Tip: By adding the `showText()` method to the above `onItemSelected()` method, you have enabled the spinner selection listener to display the spinner choice along with the phone number, so that you no longer need the **Show** button that called the `showText()` method.

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 = ;
```

1. Add `MainActivity.class.getSimpleName()` to use the simple name of the class for TAG:

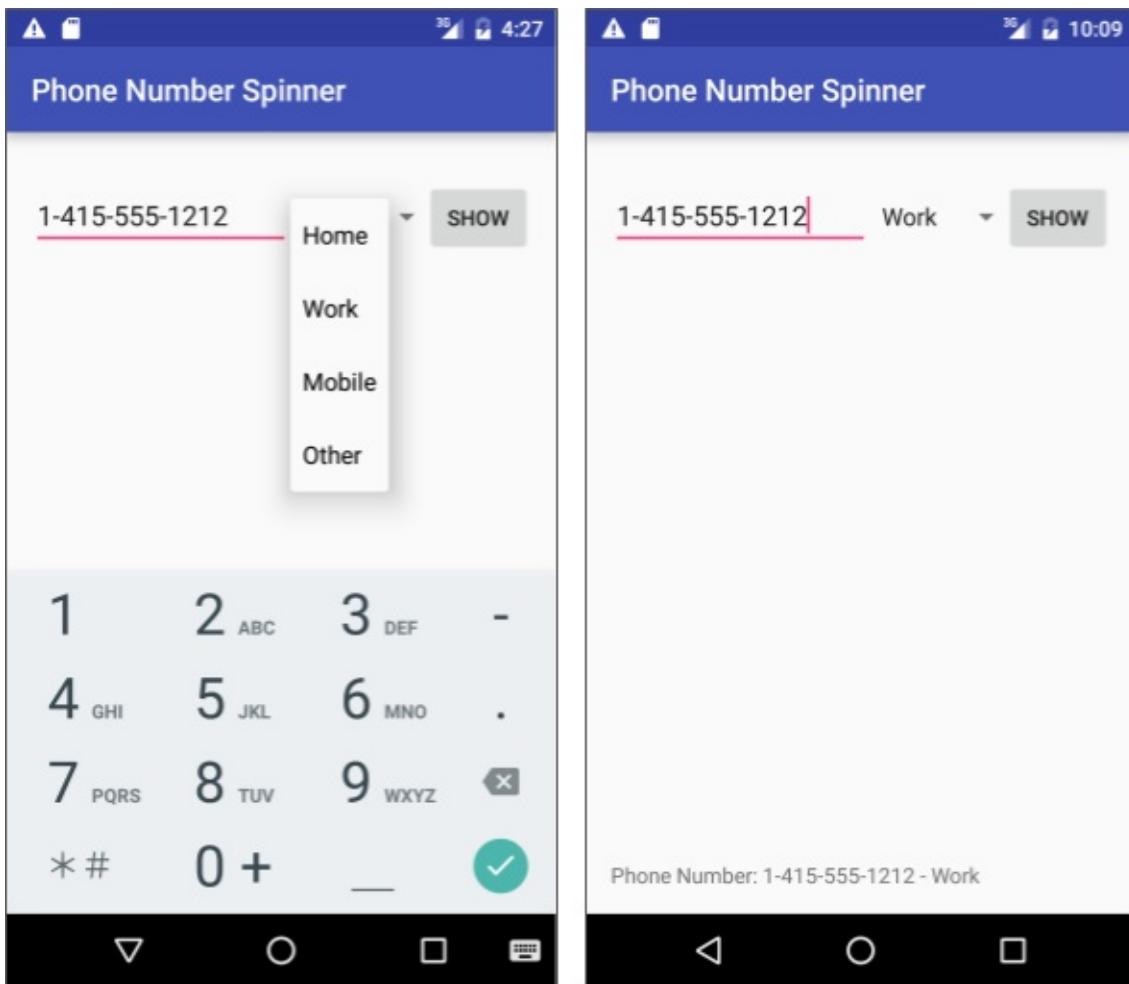
```
private static final String TAG = MainActivity.class.getSimpleName();
```

1. 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);
```

1. Run the app.

The spinner appears next to the phone entry field and shows the first choice (Home). Tapping the spinner reveals all the choices, as shown on the left side of the figure below. After entering a phone number and choosing a spinner item, a message appears at the bottom of the screen with the phone number and the selected spinner item, as shown on the right side of the figure below. (You can also tap the **Show** button to show both the phone number and the spinner item, but since this is redundant, you can now remove the **Show** button.)

**Solution Code:**

Android Studio project: [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.

In this practical, you will use 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 Studio 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>	<code>"Tap to test the alert."</code>

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>	<code>"Alert"</code>
<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 `button_top_margin`.

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. You use `AlertDialog.Builder` 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`. 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 only when the button is clicked, which is useful if the dialog is seen only rarely (when the user takes a certain path through the app). However, if the dialog appears often, 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 myAlertBuilder = new
        AlertDialog.Builder(MainActivity.this);
```

Note: If `AlertDialog.Builder` is not recognized as you enter it, click the red light bulb icon, and choose the support library version (`android.support.v7.app.AlertDialog`) for importing into your activity.

2. Set the title and the message for the alert dialog inside `onClickShowAlert()` after the code in the previous step:

```
...  
// Set the dialog title.  
myAlertBuilder.setTitle("Alert");  
// Set the dialog message.  
myAlertBuilder.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:

```
...  
// Set the dialog title.  
myAlertBuilder.setTitle(R.string.alert_title);  
// Set the dialog message.  
myAlertBuilder.setMessage(R.string.alert_message);  
...
```

4. Add the **OK** button to the alert with `setPositiveButton()` and using `onClickListener()`:

```
...  
// Add the buttons.  
myAlertBuilder.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.
myAlertBuilder.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:

```
...
myAlertBuilder.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();
    }
});
...
...
```

7. Extract the string resource for `"Cancel"` and `"Pressed Cancel"`. The statement should now be:

```
...
myAlertBuilder.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();
    }
});
...
...
```

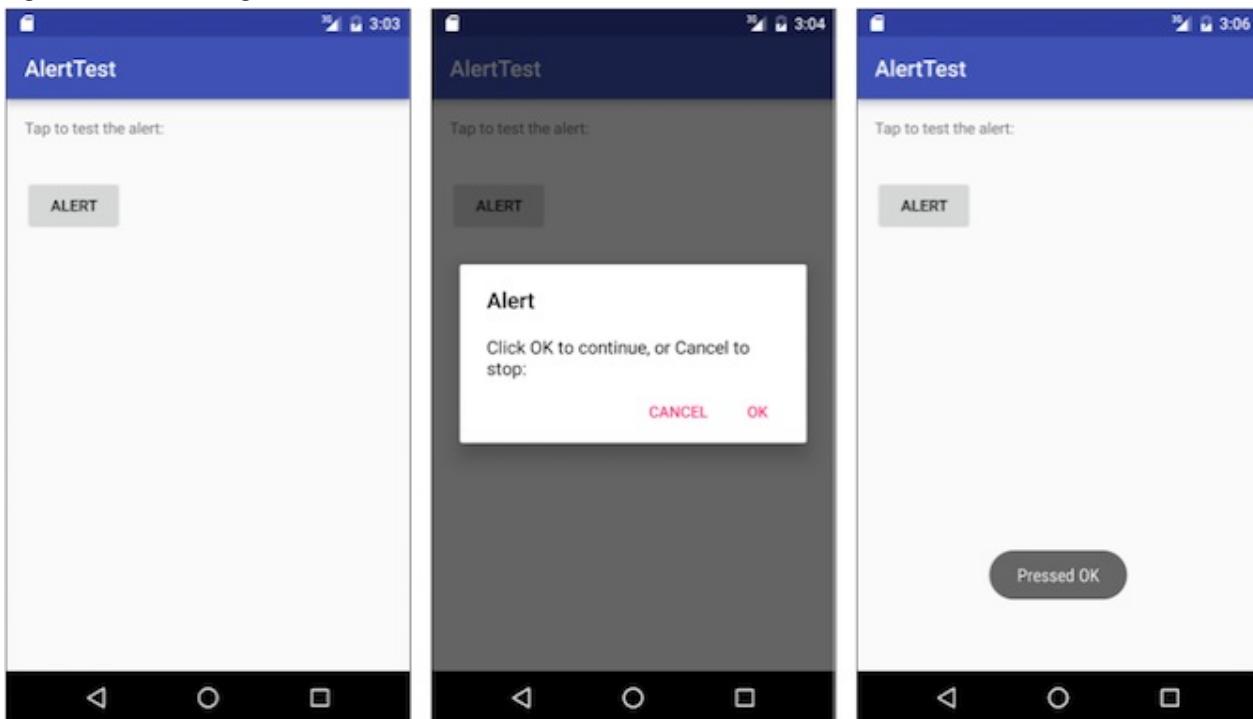
8. Add `show()`, which creates and then displays the alert dialog, to the end of `onClickShowAlert()`:

```
...
// Create and show the AlertDialog.
myAlertBuilder.show();
}
```

Tip: To learn more about onClickListener and other listeners, see [User Interface: Input Events](#).

9. Run the app.

You should be able to tap the **Alert** button, shown on the left side of the figure below, to see the alert dialog, shown in the center of the figure below. The dialog shows **OK** and **Cancel** buttons, and a toast message appears showing which one you pressed, as shown on the right side of the figure below.



Solution code:

Android Studio project: [AlertSample](#)

Task 5. Use a picker for user input

Android provides ready-to-use dialogs, called *pickers*, for picking a time or a date. You can use them to ensure that your users pick a valid time or date that is formatted correctly and adjusted to the user's local time and date. Each picker provides controls for selecting each part of the time (hour, minute, AM/PM) or date (month, day, year). You can read all about setting up pickers in [Pickers](#).

In this task you'll create a new project, and add the date picker and time picker. You will also learn how to use fragments. A *fragment* is a behavior or a portion of user interface within an activity. It's like a mini-activity within the main activity, with its own lifecycle, and it's used for building a picker. All the work is done for you. To learn about fragments, see [Fragments](#) in the API Guide.

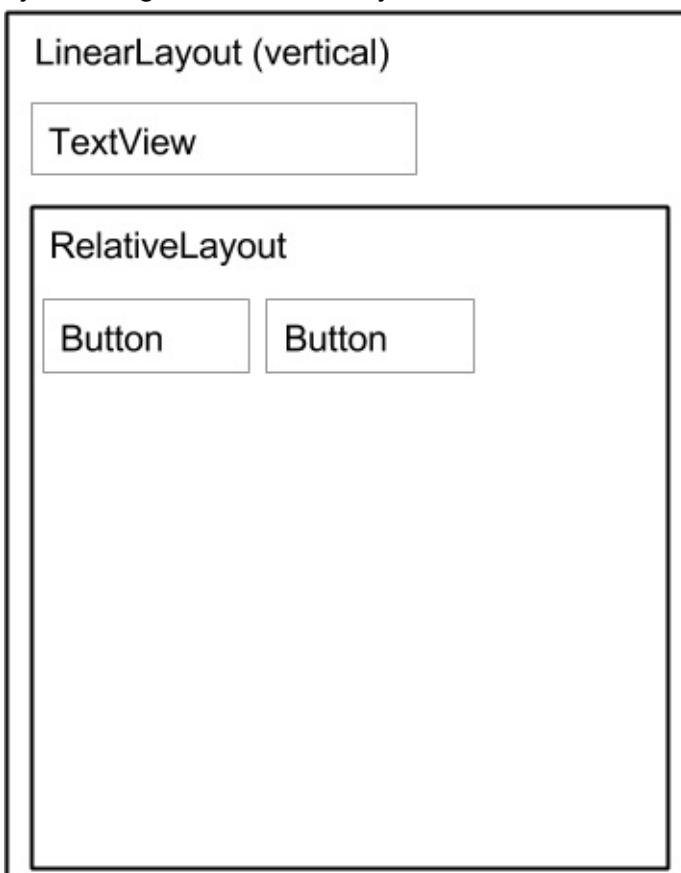
One benefit of using fragments for the pickers is that you can isolate the code sections for managing the date and the time for various locales that display date and time in different ways. The best practice to show a picker is to use an instance of [DialogFragment](#), which is a subclass of Fragment. A DialogFragment displays a dialog window floating on top of its activity's window. In this exercise, you'll add a fragment for each picker dialog and use DialogFragment to manage the dialog lifecycle.

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.

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 extract the text to the string resource `choose_datetime`.

TextView Attribute	Old Value	New Value
android:text	"Hello World"	"@string/choose_datetime"

5. Add the `android:textSize` attribute and enter a text size of **20sp**. Extract the `android:textSize` dimension to `text_size` .

TextView Attribute	Old Value	New Value
android:textSize		"@dimen/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
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:id	"@+id/button_date"
android:layout_marginTop	"12dp"
android:text	"Date"
android:onClick	"showDatePickerDialog"

Don't worry that the `showDatePickerDialog` reference is in red. The method hasn't been defined yet—you define it later.

8. Extract the string "Date" into the string resource `date_button` .
 9. Extract the dimension "12dp" for `android:layout_marginTop` to `button_top_margin` .
 10. Add the second `Button` element inside the `RelativeLayout` child with the following attributes:

Second Button Attribute	Value
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:id	"@+id/button_time"
android:layout_marginTop	"@dimen/button_top_margin"
android:layout_alignBottom	"@id/button_date"
android:layout_toRightOf	"@id/button_date"
android:text	"Time"
android:onClick	"showTimePickerDialog"

The `showTimePickerDialog` reference is in red. The method hasn't been defined yet —

you define it later.

11. Extract the string "Time" into the string resource `time_button`.
12. If you haven't already done so, click the **Preview** tab to show a preview of the layout. It should look like the code and figure below.

Solution code for the main 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: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="com.example.android.DateTimePicker.MainActivity">

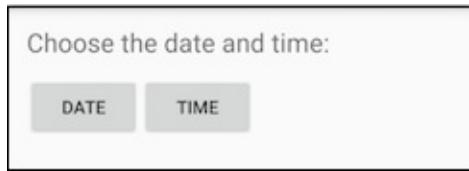
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="@dimen/text_size"
        android:text="@string/choose_datetime"/>

    <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="@dimen/button_top_margin"
            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="@dimen/button_top_margin"
            android:layout_alignBottom="@+id/button_date"
            android:layout_toRightOf="@+id/button_date"
            android:text="@string/time_button"
            android:onClick="showTimePickerDialog"/>

    </RelativeLayout>
</LinearLayout>
```



5.2 Create a new fragment for the date picker

In this exercise, you'll add a fragment for the date picker. A fragment is like a mini-activity within the main activity, with its own lifecycle.

1. Expand `app > java > com.example.android.DateTimePickerPickers` and select **MainActivity**.
2. Choose **File > New > Fragment > Fragment (Blank)**, and name the fragment **DatePickerFragment**. Uncheck all three checkbox options so that you do *not* create a layout XML, do *not* include fragment factory methods, and do *not* include interface callbacks. You don't need to create a layout for a standard picker. Click **Finish** to create the fragment.
3. Open **DatePickerFragment** and edit the `DatePickerFragment` class definition to extend `DialogFragment` and implement `DatePickerDialog.OnDateSetListener` to create a standard date picker with a listener. See [Picker](#) for more information about extending `DialogFragment` for a date picker:

```
public class DatePickerFragment extends DialogFragment
    implements DatePickerDialog.OnDateSetListener {
```

As you type **DialogFragment** and **DatePickerDialog.OnDateSetListener** Android Studio automatically adds the following in the `import` block at the top:

```
import android.app.DatePickerDialog;
import android.support.v4.app.DialogFragment;
```

In addition, a red bulb icon appears in the left margin after a few seconds.

4. Click the red bulb icon and choose **Implement methods** from the pop-up menu. A dialog appears with `onDateSet()` already selected and the "Insert @Override" option checked. Click **OK** to create the empty `onDateSet()` method. This method will be called when the user sets the date. After adding the empty `onDateSet()` method, Android Studio automatically adds the following in the `import` block at the top:

```
import android.widget.DatePicker;
```

The `onDateSet()` method's parameters should be `int year`, `int month`, and `int dayOfMonth`. Change the `dayOfMonth` parameter to `day` for brevity:

```
public void onDateSet(DatePicker view, int year, int month, int day)
```

5. Remove the empty public constructor for `DatePickerFragment`.
6. Replace `onCreateView()` with `onCreateDialog()` that returns `Dialog`, and annotate `onCreateDialog()` with `@NonNull` to indicate that the return value `Dialog` can't be null—any attempt to refer to the return value `Dialog` must be null-checked.

```
@NonNull  
@Override  
public Dialog onCreateDialog(Bundle savedInstanceState) {  
    ...  
}
```

7. 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. As you enter `Calendar`, specify the import to be `java.util.Calendar`.

```
@NonNull  
@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);  
}
```

Solution code for `DatePickerFragment`:

```

public class DatePickerFragment extends DialogFragment
    implements DatePickerDialog.OnDateSetListener {

    @NonNull
    @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);
    }

    public void onDateSet(DatePicker view, int year, int month, int day) {
        // Do something with the date chosen by the user.
    }
}

```

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.
2. 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.
3. 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`. `TimePickerFragment` performs the same tasks as the `DatePickerFragment`, but with time values:
 - o It extends `DialogFragment` and implements `TimePickerDialog.OnTimeSetListener` to create a standard time picker with a listener. See [Picker](#) for more information about extending `DialogFragment` for a time picker.
 - o 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. As you enter `calendar`, specify the import to be `java.util.Calendar`.
 - o It also defines the empty `onTimeSet()` method for you to add code to use the `hourOfDay` and `minute` the user selects. This method will be called when the user

sets the time:

```
public void onTimeSet(TimePicker view,
                      int hourOfDay, int minute) {
    // Do something with the time chosen by the user.
}
```

Note: As you make the changes, Android Studio automatically adds the following in the import block at the top:

```
import android.support.v4.app.DialogFragment;
import android.app.TimePickerDialog;
import android.widget.TimePicker;
import java.util.Calendar;
```

Solution code for TimePickerFragment:

```
public class TimePickerFragment extends DialogFragment
    implements TimePickerDialog.OnTimeSetListener {

    @NonNull
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the current time as the default values for the picker.
        final Calendar c = Calendar.getInstance();
        int hour = c.get(Calendar.HOUR_OF_DAY);
        int minute = c.get(Calendar.MINUTE);

        // Create a new instance of TimePickerDialog and return it.
        return new TimePickerDialog(getActivity(), this, hour, minute,
            DateFormat.is24HourFormat(getActivity()));
    }

    public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
        // Do something with the time chosen by the user.
    }
}
```

5.4 Modify the main activity

While much of the code in the main activity stays the same, you need to add methods that create instances of `FragmentManager` to manage each fragment and show each picker.

1. Create string resources in `strings.xml`:

```
<string name="date_picker">datePicker</string>
<string name="time_picker">timePicker</string>
```

2. Open **MainActivity**.
3. Add the `showDatePickerDialog()` and `showTimePickerDialog()` methods, referring to the code below. It creates an instance of `FragmentManager` to manage the fragment automatically, and to show the picker. For more information about fragments, see [Fragments](#).

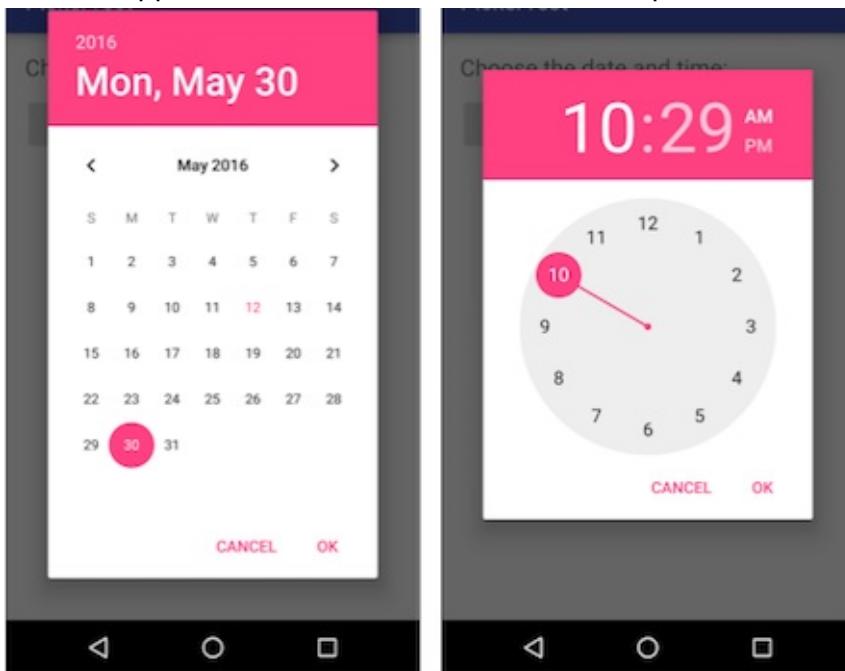
```
public class MainActivity extends AppCompatActivity {

    @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));
    }
}
```

4. 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 pass the date and time back to `MainActivity`, and convert the date and time to strings that you can show in a toast message.

1. Open `MainActivity` and add the `processDatePickerResult()` method signature that takes the `year`, `month`, and `day` as arguments:

```
public void processDatePickerResult(int year, int month, int day) {  
}
```

2. Add the following code to the `processDatePickerResult()` method to convert the `month`, `day`, and `year` to separate strings:

```
String month_string = Integer.toString(month+1);  
String day_string = Integer.toString(day);  
String year_string = Integer.toString(year);
```

Tip: The `month` integer returned by the date picker starts counting at 0 for January, so you need to add 1 to it to start showing months starting at 1.

3. Add the following after the above code to concatenate the three strings and include slash marks for the U.S. date format:

```
String dateMessage = (month_string + "/" +  
                     day_string + "/" + year_string);
```

4. Add the following after the above statement to display a toast message:

```
Toast.makeText(this, "Date: " + dateMessage,  
              Toast.LENGTH_SHORT).show();
```

5. Extract the hard-coded string "Date: " into a string resource named `date`. This automatically replaces the hard-coded string with `getString(R.string.date)`. The code for the `processDatePickerResult()` method should now look like this:

```
public void processDatePickerResult(int year, int month, int day) {  
    String month_string = Integer.toString(month + 1);  
    String day_string = Integer.toString(day);  
    String year_string = Integer.toString(year);  
    // Assign the concatenated strings to dateMessage.  
    String dateMessage = (month_string + "/" +  
                          day_string + "/" + year_string);  
    Toast.makeText(this, getString(R.string.date) + dateMessage,  
                  Toast.LENGTH_SHORT).show();  
}
```

6. Open `DatePickerFragment`, and add the following to the `onDateSet()` method to invoke the `processDatePickerResult()` method in `MainActivity` and pass it the `year`,

```
month , and day :

public void onDateSet(DatePicker view, int year, int month, int day) {
    // Set the activity to the Main Activity.
    MainActivity activity = (MainActivity) getActivity();
    // Invoke Main Activity's processDatePickerResult() method.
    activity.processDatePickerResult(year, month, day);
}
```

You use `getActivity()` which, when used in a fragment, returns the activity the fragment is currently associated with. You need this because you can't call a method in `MainActivity` without the context of `MainActivity` (you would have to use an `intent` instead, as you learned in a previous lesson). The activity inherits the context, so you can use it as the context for calling the method (as in `activity.processDatePickerResult()`).

7. The `TimePickerFragment` uses the same logic. Open **MainActivity** and add the `processTimePickerResult()` method signature that takes the `hourOfDay` and `minute` as arguments:

```
public void processTimePickerResult(int hourOfDay, int minute) {  
}
```

8. Add the following code to the `processTimePickerResult()` method to convert the `hourOfDay` and `minute` to separate strings:

```
String hour_string = Integer.toString(hourOfDay);  
String minute_string = Integer.toString(minute);
```

9. Add the following after the above code to concatenate the strings and include a colon for the time format:

```
String timeMessage = (hour_string + ":" + minute_string);
```

10. Add the following after the above statement to display a toast message:

```
Toast.makeText(this, "Time: " + timeMessage,  
        Toast.LENGTH_SHORT).show();
```

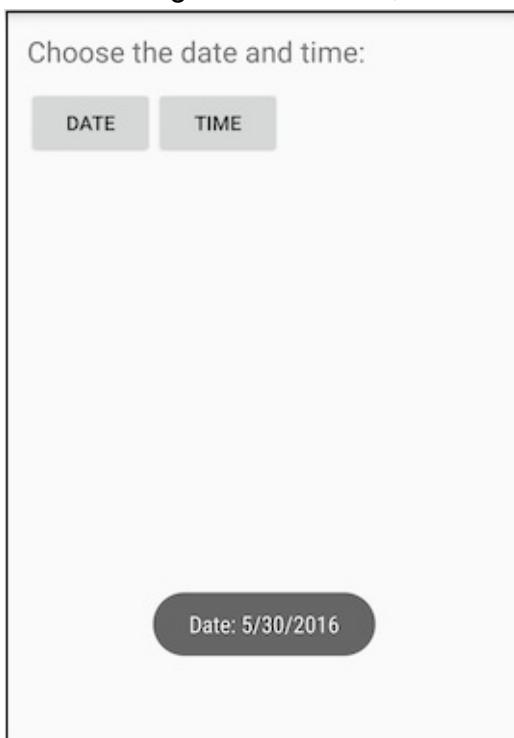
11. Extract the hard-coded string `"Time: "` into a string resource named `time`. This automatically replaces the hard-coded string with `getString(R.string.time)`. The code for the `processDatePickerResult()` method should now look like this:

```
public void processTimePickerResult(int hourOfDay, int minute) {  
    // Convert time elements into strings.  
    String hour_string = Integer.toString(hourOfDay);  
    String minute_string = Integer.toString(minute);  
    // Assign the concatenated strings to timeMessage.  
    String timeMessage = (hour_string + ":" + minute_string);  
    Toast.makeText(this, getString(R.string.time) + timeMessage,  
        Toast.LENGTH_SHORT).show();  
}
```

12. Open **TimePickerFragment**, and add the following to the `onTimeSet()` method to invoke the `processTimePickerResult()` method in `MainActivity` and pass it the `hourOfDay` and `minute`:

```
public void onTimeSet(TimePicker view, int hourOfDay, int minute) {  
    // Set the activity to the Main Activity.  
    MainActivity activity = (MainActivity) getActivity();  
    // Invoke Main Activity's processTimePickerResult() method.  
    activity.processTimePickerResult(hourOfDay, minute);  
}
```

13. You can now run the app. After selecting the date or time, the date or time appears in a toast message at the bottom, as shown in the figure below.



Solution Code:

Android Studio project: [DateTimePickers](#)

Task 6: Use image views as buttons

You can make a view clickable, as a button, by adding the `android:onClick` attribute in the XML layout. For example, you can make an image act like a button by adding `android:onClick` to the [ImageView](#).

Tip: If you are using multiple images as clickable images, arrange them in a viewgroup so that they are grouped together.

In this task you'll create a prototype of an app for ordering desserts from a café. After starting a new project based on the Basic Activity template, you'll modify the "Hello World" `TextView` with appropriate text, and add images to use for the "Add to order" buttons.

6.1 Start the new project

1. Start a new Android Studio project with the app name **Droid Cafe**. Choose the **Basic Activity** template, accept the default settings, 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` and extract the "Hello World" string in the `TextView` to use the `intro_text` resource name. Then open `strings.xml` and redefine the `intro_text` resource to use more descriptive text, such as **Droid Desserts**:

```
<string id="intro_text">Droid Desserts</string>
```

3. 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`.
4. 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`. You will use these resource names in subsequent steps.
5. 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 a dessert."

6. Extract the string resource for the `android:text` attribute to the resource name `choose_a_dessert`.

6.2 Add the images

1. The images named donut_circle.jpg, froyo_circle.jpg, and icecream_circle.jpg are provided with the starter apps in the [4_1_P_starter_images.zip](#) file, which you can unzip on your computer. To copy the images to your project, follow these steps:
 - i. Close your project.
 - ii. Copy the image files into your project's **drawable** folder. Find the **drawable** folder in a project by using this path: *project_name > app > src > main > res > drawable*
 - iii. Reopen your project.
2. Open **content_main.xml** file again and add an `ImageView` for the donut image to the layout under the `choose_dessert` view, using the following attributes:

ImageView Attribute for donut	Value
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"

3. Extract the `android:contentDescription` attribute value to the string resource `donuts`. You will use this string resource in the next step.
4. Add a `TextView` that will appear next to the donut image as a description, with the following attributes:

TextView Attribute	Value
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	"@string/donuts"

5. Extract the dimension resource for the `android:padding` attribute to the resource name `padding_wide`. You will use this resource name in subsequent steps.
6. Add a second `ImageView` to the layout for the ice cream sandwich, using the following attributes:

ImageView Attribute for ice_cream	Value
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 sandwiches have chocolate wafers and vanilla filling."
android:src	"@drawable/icecream_circle"

7. Extract the `android:contentDescription` attribute value to the string resource `ice_cream_sandwiches` .
8. Add a `TextView` that will appear next to the ice cream sandwich as a description, with the following attributes:

TextView Attribute	Value
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:padding	"@dimen/padding_wide"
android:layout_below	"@+id/donut"
android:layout_toRightOf	"@id/ice_cream"
android:text	"@string/ice_cream_sandwiches"

9. Add a third `ImageView` to the layout for the froyo, using the following attributes:

ImageView Attribute for ice_cream	Value
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:padding	"@dimen/padding_regular"
android:id	"@+id/froyo"
android:layout_below	"@id/ice_cream"
android:contentDescription	"FroYo is premium self-serve frozen yogurt."
android:src	"@drawable/froyo_circle"

10. Extract the `android:contentDescription` attribute value to the string resource `froyo` .

11. Add a `TextView` that will appear next to the froyo as a description, 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_wide"
<code>android:layout_below</code>	"@+id/ice_cream"
<code>android:layout_toRightOf</code>	"@+id/froyo"
<code>android:text</code>	"@string/froyo"

6.3 Add onClick methods for the image views

You can add the `android:onClick` attribute to any View to make it clickable as a button. In this step you will add `android:onClick` to the images in the `content_main.xml` layout. You need to also add a method for the `android:onClick` attribute to call. The method, for this task, displays a toast message showing which image was tapped. (In a later task, you will modify the method to launch another activity called `OrderActivity`.)

1. Add the following string resources to the `strings.xml` file for the strings to be shown in the toast message:

```
<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 `displayToast()` method for displaying a toast message:

```
public void displayToast(String message) {
    Toast.makeText(getApplicationContext(), message,
        Toast.LENGTH_SHORT).show();
}
```

3. Add the following `showFoodOrder()` method to the end of `MainActivity` (before the closing bracket). For this task, use the `displayToast()` method to display a toast message:

```
/**  
 * Displays a toast message for the food order  
 * and starts the OrderActivity activity.  
 * @param message Message to display.  
 */  
public void showFoodOrder(String message) {  
    showToast(message);  
}
```

Tip: The first four lines are a comment in the Javadoc format, which makes the code easier to understand and also helps generate documentation for your code if you use [Javadoc](#). It is a best practice to add such a comment to every new method you create. For more information about how to write comments, see [How to Write Comments for the Javadoc Tool](#).

Although you could have added this method in any position within **MainActivity**, it is best practice to put your own methods *below* the methods already provided in **MainActivity** by the template.

1. Add the following methods to the end of **MainActivity** (you can add them before `showFoodOrder()`):

```
/**  
 * Shows a message that the donut image was clicked.  
 */  
public void showDonutOrder(View view) {  
    showFoodOrder(getString(R.string.donut_order_message));  
}  
  
/**  
 * Shows a message that the ice cream sandwich image was clicked.  
 */  
public void showIceCreamOrder(View view) {  
    showFoodOrder(getString(R.string.ice_cream_order_message));  
}  
  
/**  
 * Shows a message that the froyo image was clicked.  
 */  
public void showFroyoOrder(View view) {  
    showFoodOrder(getString(R.string.froyo_order_message));  
}
```

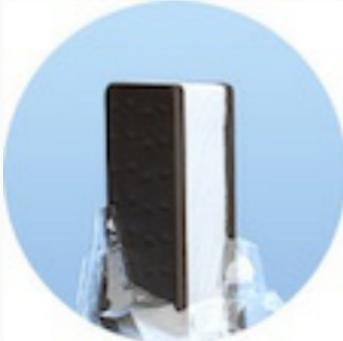
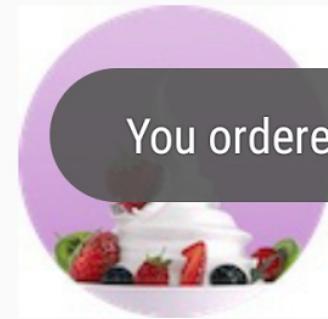
2. Add the `android:onClick` attribute to the three `ImageViews` in `content_main.xml`:

```
<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"/>
```

3. Run the app.

Clicking the donut, ice cream sandwich, or froyo image displays a toast message about the order, as shown in the figure below.

The screenshot shows a mobile application interface for "Droid Cafe". At the top, there is a blue header bar with the text "Droid Cafe" on the left and three vertical dots on the right. The main content area has a light gray background. A large title "Droid Desserts" is centered at the top of the content area. Below it, the text "Choose a dessert." is displayed. There are three dessert items listed, each with an image and a description:

-  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.

A red circular button with a white envelope icon is located in the bottom right corner of the content area. The bottom of the screen features a black navigation bar with three icons: a triangle pointing left, a circle, and a square.

Task 7: Use radio buttons

Radio buttons are input controls that 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.

Later in this practical you will add another activity and screen layout for setting the delivery options for a food order, and use radio buttons for the delivery choices.

For an overview and more sample code for radio buttons, see [Radio Buttons](#).

7.1 Add another activity

As you learned in a previous lesson, an *activity* represents a single screen in your app in which your user can perform a single, focused task. You already have one activity, `MainActivity.java`. You will now add another activity for setting the delivery options for an order, and use an explicit intent to launch the second activity.

1. Right-click the `com.example.android.droidcafe` folder in the left column 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.

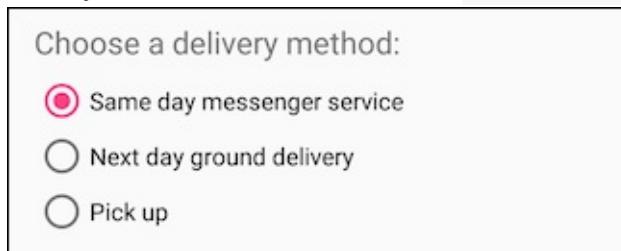
2. Open **MainActivity**. Change the `showFoodOrder()` method to make an explicit intent to start `OrderActivity` :

```
public void showFoodOrder(String message) {  
    displayToast(message);  
    Intent intent = new Intent(this, OrderActivity.class);  
    startActivity(intent);  
}
```

3. Run the app. Clicking an image button now launches the second activity, which is a blank screen. (The toast message appears briefly on the blank screen.)

7.2 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`. After editing the layout file, the layout for the radio buttons in `OrderActivity` should look like the figure below.



Since radio button selections are mutually exclusive, you will group them together inside a `RadioGroup`. By grouping them together, the Android system ensures that only one radio button can be selected at a time.

Note: The order in which you list the `'RadioButton'` elements determines the order that they appear on the screen.

1. Open `activity_order.xml`, and add a `TextView` element with the id `order_intro_text`:

TextView Attribute	Value
<code>android:id</code>	<code>"@+id/order_intro_text"</code>
<code>android:layout_width</code>	<code>"match_parent"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:layout_marginTop</code>	<code>"24dp"</code>
<code>android:layout_marginBottom</code>	<code>"6dp"</code>
<code>android:textSize</code>	<code>"18sp"</code>
<code>android:text</code>	<code>"Choose a delivery method:"</code>

2. Extract the string resource for `"Choose a delivery method:"` to be `choose_delivery_method`.
3. Extract the dimension resources for the margin values:
4. `"24dp"` to `text_margin_top`
5. `"6dp"` to `text_margin_bottom`
6. `"18sp"` to `intro_text_size`
7. Add a `RadioGroup` to the layout underneath the `TextView` you just added:

```
<RadioGroup
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:layout_below="@+id/order_intro_text">
</RadioGroup>
```

8. 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.

RadioButton #1 Attribute	Value
android:id	<code>"@+id/sameday"</code>
android:layout_width	<code>"wrap_content"</code>
android:layout_height	<code>"wrap_content"</code>
android:text	<code>"Same day messenger service"</code>
android:onClick	<code>"onRadioButtonClicked"</code>

RadioButton #2 Attribute	Value
android:id	<code>"@+id/nextday"</code>
android:layout_width	<code>"wrap_content"</code>
android:layout_height	<code>"wrap_content"</code>
android:text	<code>"Next day ground delivery"</code>
android:onClick	<code>"onRadioButtonClicked"</code>

RadioButton #3 Attribute	Value
android:id	<code>"@+id/pickup"</code>
android:layout_width	<code>"wrap_content"</code>
android:layout_height	<code>"wrap_content"</code>
android:text	<code>"Pick up"</code>
android:onClick	<code>"onRadioButtonClicked"</code>

9. 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`

7.3 Add the radio button click 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.

Ordinarily your app would display some message regarding which type of delivery was chosen. You will accomplish this with a toast message by creating a method called `displayToast()` in `OrderActivity`.

In the `onRadioButtonClicked()` method you will use a `switch case` block to check if a radio button has been clicked. At the end of the `switch case` block, you will add a `default` statement that displays a `log` message if none of the radio buttons were checked.

1. Open `strings.xml` and create the following string resources:
 - i. A resource named `chosen` for the string "chosen: " (include the space after the colon and the quotation marks).
 - ii. A resource named `nothing_clicked` for the string "`onRadioButtonClicked: Nothing clicked.`"
2. Open `OrderActivity` and add the following statement to define `TAG_ACTIVITY` for the `log` message:

```
private static final String TAG_ACTIVITY =  
    OrderActivity.class.getSimpleName();
```

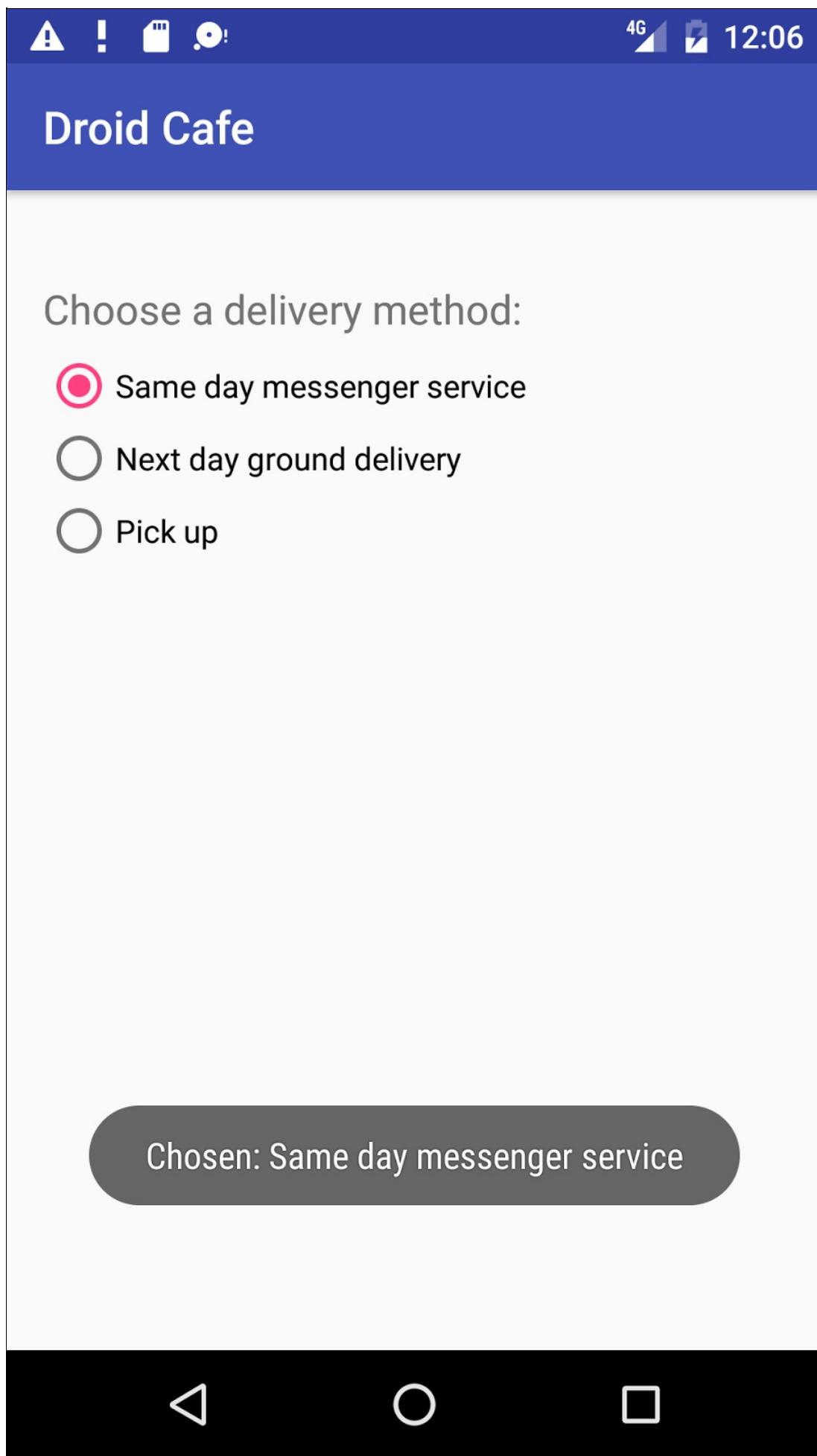
3. Add the following `displayToast` method to `OrderActivity`:

```
public void displayToast(String message) {  
    Toast.makeText(getApplicationContext(), message,  
        Toast.LENGTH_SHORT).show();  
}
```

4. 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;  
        default:  
            Log.d(TAG_ACTIVITY, getString(R.string.nothing_clicked));  
            break;  
    }  
}
```

5. Run the app. Tap an image to see the OrderActivity activity, which shows the delivery choices. Tap a delivery choice, and you will see a toast message at the bottom of the screen with the choice, as shown in the figure below.



Solution code

Android Studio project: [DroidCafe Part 1](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: You can also perform an action directly from the keyboard and replace the

Return (Enter) key with a "send" 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](#).

The next step is to override `onEditorAction()` and 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 the `dialNumber()` method 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!");
    }
}

```

*Summary

In this practical, you learned how to:

- Set up XML layout attributes to control the keyboard for an `EditText` element:
 - Use the `textAutoCorrect` value for the `android:inputType` attribute to change the keyboard so that it suggests spelling corrections.
 - Use the `textCapSentences` value for the `android:inputType` attribute to start each new sentence with a capital letter.
 - Use the `textPassword` value for the `android:inputType` attribute to hide a password when entering it.
 - Use the `textEmailAddress` value for the `android:inputType` attribute to show an email keyboard rather than a standard keyboard.
 - Use the `phone` value for the `android:inputType` attribute to show a phone keypad rather than a standard keyboard.
 - Challenge: Use 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.

- Use a `Spinner` input control to provide a drop-down menu, and writing code to control it:
 - Use an `ArrayAdapter` to assign an array of text values as the spinner menu items.
 - Implement the `AdapterView.OnItemSelectedListener` interface, which requires also adding the `onItemSelected()` and `onNothingSelected()` callback methods to activate the spinner and its listener.
 - Use the `onItemSelected()` callback method to retrieve the selected item in the spinner menu using `getItemAtPosition`.
- Use `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.
- Use the standard date and time pickers:
 - Add a fragment for a date picker, and extending the `DialogFragment` class to implement `DatePickerDialog.OnDateSetListener` for a standard date picker with a listener.
 - Add a fragment for a time picker, and extending the `DialogFragment` class to implement `TimePickerDialog.OnTimeSetListener` for a standard time picker with a listener.
 - Implement the `onDateSet()`, `onTimeSet()`, and `onCreateDialog()` methods.
 - Use the `onFinishDateDialog()` and `onFinishTimeDialog()` methods to retrieve the selected date and time.
- Use images in a project:
 - Copy an image into the project, and defining an `ImageView` element to use it.
 - Add the `android:onClick` attribute to make the `ImageView` elements clickable like buttons. You can make any View clickable with the `android:onClick` attribute.
- Use radio buttons:
 - Create a second activity.
 - Add `RadioButton` elements within a `RadioGroup` in the second activity.
 - Create radio button handlers.
 - Launch the second activity from an image click.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [User Input Controls](#)

Learn more

- Android API Guide, "Develop" section:
 - [Specifying the Input Method Type](#)
 - [Text Fields](#)
 - [Input Controls](#)
 - [Spinners](#)
 - [Dialogs](#)
 - [Fragments](#)
 - [Input Events](#)
 - [Pickers](#)
 - [DateFormat](#)
 - [ImageView](#)
 - [Radio Buttons \(User Interface section\)](#)
- Material Design Specification:
 - [Dialogs design guide](#)

4.2: Using an Options Menu

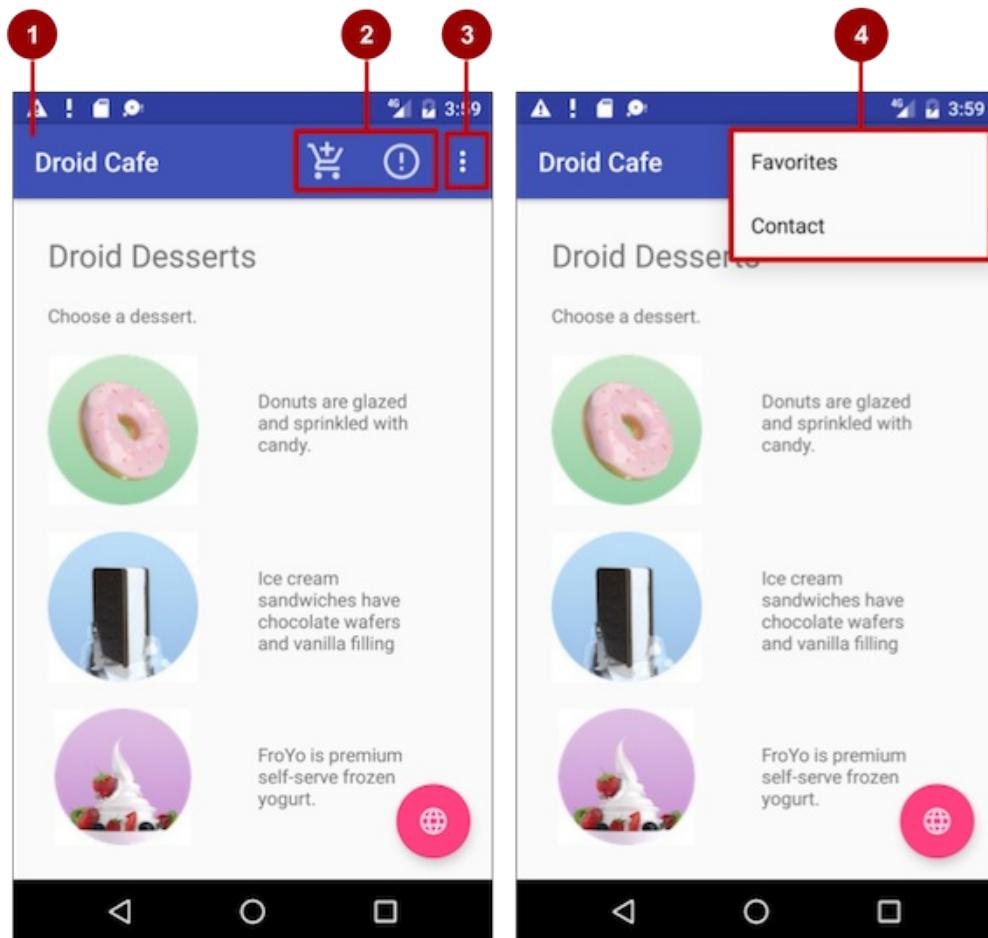
Contents:

- [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](#)
- [Coding Challenge #1](#)
- [Task 3: Handle the selected menu item](#)
- [Coding Challenge #2](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

The *app bar* (also called the *action bar*) is a dedicated space at the top of each activity screen. When you create an activity from a template (such as Basic Activity Template), an app bar is automatically included for the activity in a CoordinatorLayout root view group at the top of the view hierarchy.

The *options menu* in the app bar provides navigation to other activities in the app, or the primary options that affect using the app itself—but not ones that perform an action on an element on the screen. For example, your options menu might provide the user choices for navigating to other activities, such as placing an order, or for actions that have a global impact on the app, such as changing settings or profile information.

In this practical you'll learn about setting up the app bar and options menu in your app (shown in the figure below).



In the above figure:

1. **App bar.** The app bar includes the app title, the options menu, and the overflow button.
2. **Options menu action icons.** The first two options menu items appear as icons in the app bar.
3. **Overflow button.** The overflow button (three vertical dots) opens a menu that shows more options menu items.
4. **Options overflow menu.** After clicking the overflow button, more options menu items appear in the overflow menu.

Options menu items appear in the options overflow menu (see figure above). However, you can place some items as icons—as many as can fit—in the app bar. Using the app bar for the options menu makes your app consistent with other Android apps, allowing users to quickly understand how to operate your app and have a great experience.

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.

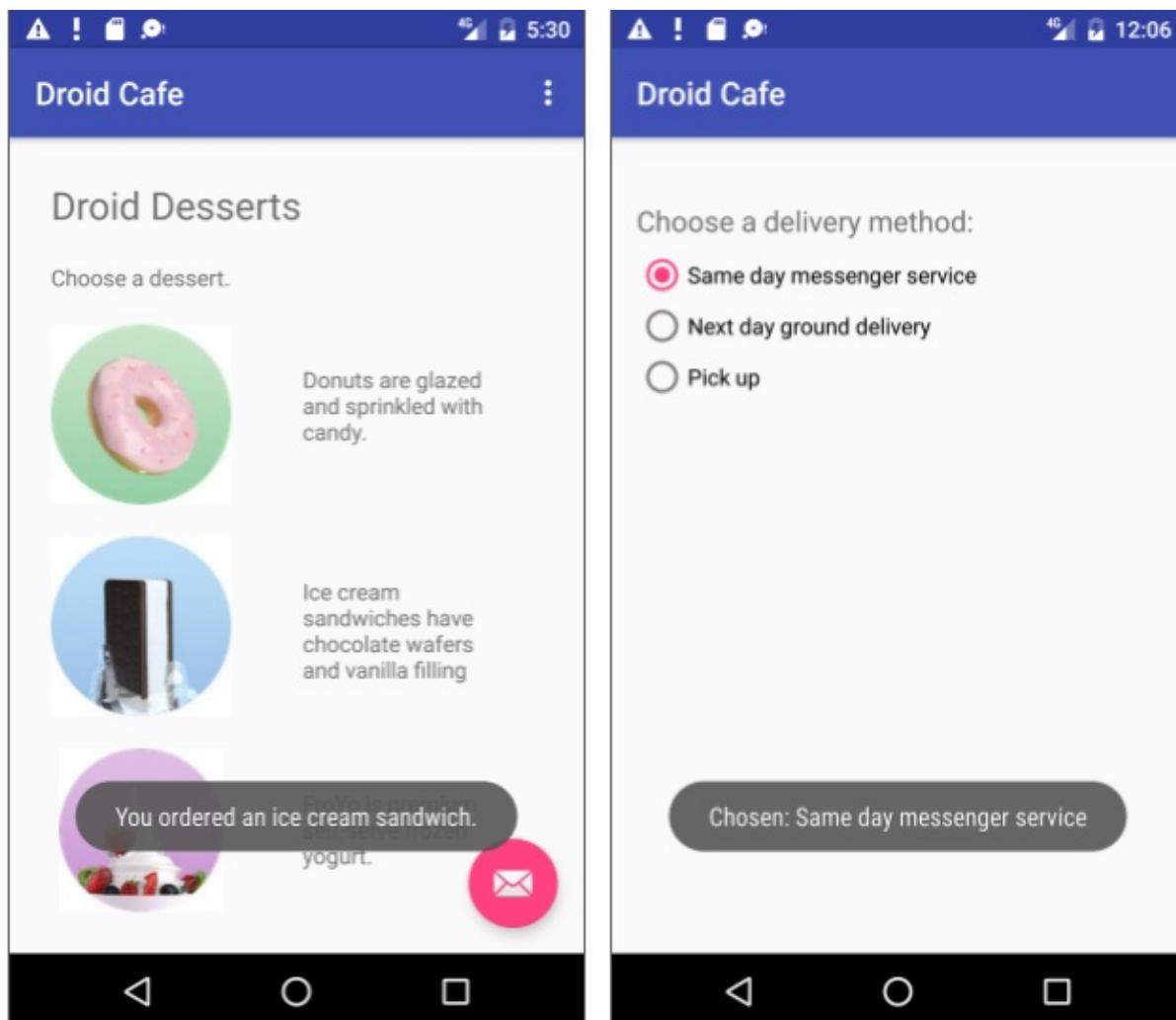
What you will DO

- Continue adding features to the Droid Cafe project from the previous practical.
- 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.

App Overview

In the previous practical you created an app called Droid Cafe, shown in the figure below, using the Basic Activity template. This template also provides a skeletal options menu in the app bar at the top of the screen. You will learn how to:

- Set up the app bar.
- Modify the options menu.
- Add icons for some of the menu items.
- Show the icon for the menu item in the app bar rather than the overflow menu.
- Show the item in the overflow menu, depending on the screen size and orientation.



For this exercise you are using the [v7 appcompat](#) support library's [Toolbar](#) as an app bar. There are other ways to implement an app bar. For example, some themes set up an [ActionBar](#) as an app bar by default. But using the appcompat [Toolbar](#) makes it easy to set up an app bar that works on the widest range of devices, and also gives you room to customize your app bar later on as your app develops.

To read more about design considerations for using the app bar, see [App Bar](#) in the Material Design Specification.

To start the project from where you left off in the previous practical, download:

Android Studio project: [DroidCafe Part 1](#)

Task 1. Add items to the options menu

You will open the Droid Cafe project from the previous practical, and add menu items to the options menu in the app bar at the top of the screen.

1.1 Examine the app bar code

1. Open the Droid Cafe project from the previous practical. The project includes the following layout files in the `res > layout` folder:
 - i. `activity_main.xml`: The main layout for MainActivity, the first screen the user sees.
 - ii. `content_main.xml`: The layout for the content of the MainActivity screen, which (as you will see shortly) is *included* within `activity_main.xml`.
 - iii. `activity_order.xml`: The layout for OrderActivity, which you added in the previous practical.
2. Open `content_main.xml`. In the previous practical, you added some TextViews and ImageViews to the root view group, which is a `RelativeLayout`.

The layout *behavior* for the `RelativeLayout` is set to

```
@string/appbar_scrolling_view_behavior
```

, which controls the scrolling behavior of the screen in relation to the app bar at the top. Right-click (Control-click) this string resource and choose **Go To > Declaration** to see the string resource's actual value, which is defined in a file called "values.xml". This file is generated by Android Studio, not visible in the Project: Android view and should not be edited. The actual value of

`@string/appbar_scrolling_view_behavior` in values.xml 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 Specification.

3. Open `activity_main.xml` to see the main layout, which uses a CoordinatorLayout layout with an embedded `AppBarLayout` layout. The `CoordinatorLayout` and the `AppBarLayout` tags require fully qualified names that specify `android.support.design`, which is the Android Design Support Library.

AppBarLayout is a vertical LinearLayout which uses the `Toolbar` class in the support library, instead of the native ActionBar, to implement an app bar. The app bar is a section at the top of the display that can display the activity title, navigation, and other interactive items. The native ActionBar behaves differently depending on the version of Android running on the device. For this reason, if you are adding an options menu, you should use the [v7 appcompat](#) support library's `Toolbar` as an app bar. Using the `Toolbar` makes it easy to set up an app bar that works on the widest range of devices, and also gives you room to customize your app bar later on as your app develops. Toolbar includes the most recent features, and works for any device that can use the support library.

The Toolbar within this layout has the id `toolbar`, and is also specified, like the `AppBarLayout`, with a fully qualified name (`android.support.v7.widget`):

```
<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.

Tip: 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* 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.

4. Run the app. Notice the bar at the top of the screen showing the name of the app (Droid Cafe). It also shows the *action overflow* button (three vertical dots) on the right side. Tap the overflow button to see the options menu, which at this point has only one menu option, **Settings**.
5. Examine the `AndroidManifest.xml` file. The `.MainActivity` activity is set to use the `NoActionBar` theme:

```
    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).

The reason these values are set is because you are defining the app bar in your layout (`activity_main.xml`) with `AppBarLayout`, rather than using an `ActionBar`. Using one of the `NoActionBar` themes prevents the app from using the native `ActionBar` class to provide

the app bar. The native `ActionBar` class behaves differently depending on what version of the Android system a device is using. By contrast, the most recent features are added to the support library's version of `Toolbar` and available on any device that can use the support library. For this reason, you should use the support library's `Toolbar` class to implement your activities' app bars instead of `ActionBar`. Using the support library's `Toolbar` ensures that your app has consistent behavior across the widest range of devices.

6. Look at `MainActivity`, 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);  
    ...
```

After setting the content view to the `activity_main.xml` layout, the `onCreate()` method sets `toolbar` to be the `Toolbar` defined in the `activity_main.xml` layout. It then 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 the 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

You will add the following menu items to the options menu of the Droid Cafe app:

- **Order:** Go to the Order Activity screen to see the food order.
- **Status:** Check the status of a food order.
- **Favorites:** Show favorite foods.
- **Contact:** Contacting the restaurant. Since you don't need the existing **Settings** item, you will change **Settings** to **Contact**.

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 of its menu items in an XML menu resource. You can then inflate the menu resource (load it as a `Menu` object) in your activity or fragment:

- 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 `action_settings` (the **Settings** choice), which is defined as:

```
<item
    android:id="@+id/action_settings"
    android:orderInCategory="100"
    android:title="@string/action_settings"
    app:showAsAction="never" />
```

In Android Studio, the `android:title` attribute shows the string value "Settings" even though the string is defined as a resource. Android Studio displays the value so that you can see at-a-glance what the value is without having to open the `strings.xml` resource file. If you click on this string, it changes to show the string resource

`"@string/action_settings"`.

- Change the following attributes of the `action_settings` item to make it the `action_contact` item (don't change the existing `android:orderInCategory` attribute):

Attribute	Value
<code>android:id</code>	<code>"@+id/action_contact"</code>
<code>android:title</code>	"Contact"
<code>app:showAsAction</code>	"never"

- Extract the hard-coded string "contact" into the string resource `action_contact`.
- Add a new 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>	"10"
<code>android:title</code>	"Order"
<code>app:showAsAction</code>	"never"

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.

- Extract the hard-coded string "order" into the string resource `action_order`.
- Add two more menu items the same way with the following attributes:

Status Item Attribute	Value
android:id	"@+id/action_status"
android:orderInCategory	"20"
android:title	"Status"
app:showAsAction	"never"

Favorites Item Attribute	Value
android:id	"@+id/action_favorites"
android:orderInCategory	"40"
android:title	"Favorites"
app:showAsAction	"never"

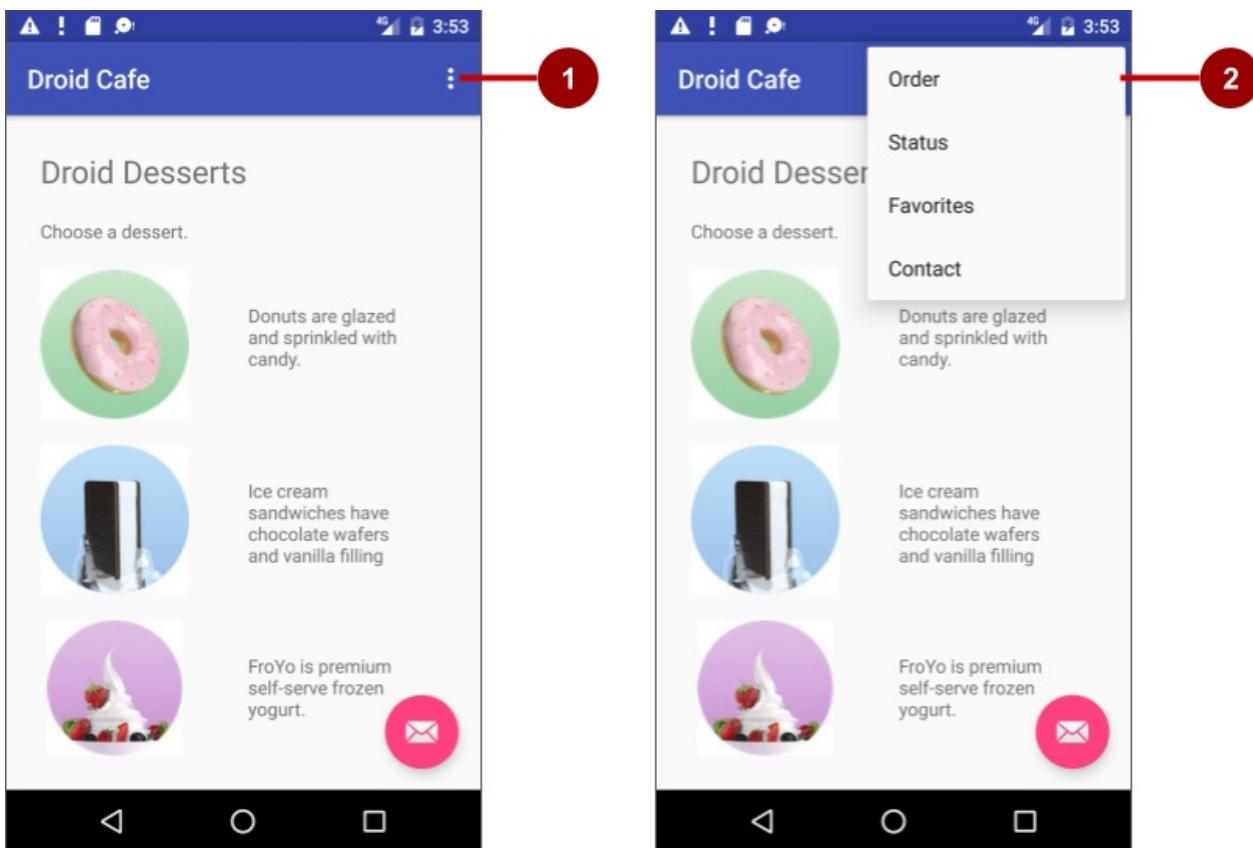
7. Extract "Status" into the resource `action_status`, and "Favorites" into the resource `action_favorites`.
8. You will display a toast message with an action message depending on which menu item the user selects. Add the following string names and values in `strings.xml` for these messages:

```
<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**, 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, shown on the left side of the figure below, to see the options menu, shown on the right side of the figure below. You will soon add callbacks to respond to items selected from this menu.



In the above figure:

1. Tap the overflow icon in the app bar to see the options menu.
2. The options menu drops down from the app bar.

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	<code>orderInCategory</code> attribute
Order	10
Status	20
Favorites	40
Contact	100

Task 2. Add icons for menu items

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.

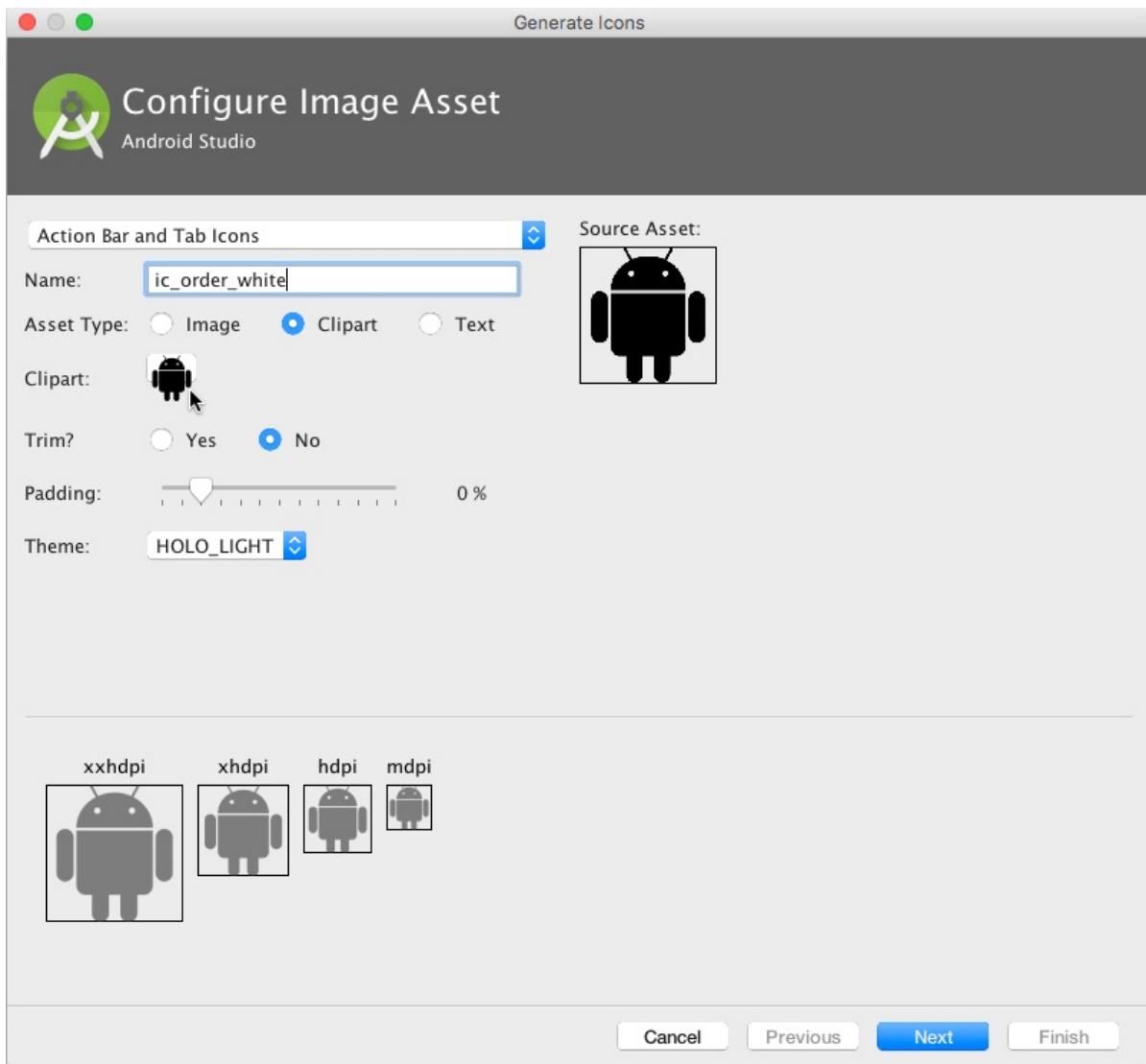
In this example, let's assume the **Order** and **Status** actions are considered the most frequently used. **Favorites** is occasionally used, and **Contact** is the least frequently used. You can set icons for these actions, and specify the following:

- **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 Control-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 [Create App Icons with Image Asset Studio](#) for a complete description.)



5. Click the clipart image (the Android logo next to "Clipart:") 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 shopping cart 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 app 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 app bar as an icon:

- `"always"` : Always appears in the app bar. (If there isn't enough room it may overlap

with other menu icons.)

- "ifRoom" : Appears in the app bar if there is room.
- "never" : Never appears in the app bar; its text appears in the overflow menu.

Follow these steps to show some of the menu items as icons:

1. 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:

Order Item Attribute	Old Value	New Value
android:icon		"@drawable/ic_order_white"
app:showAsAction	"never"	"always"

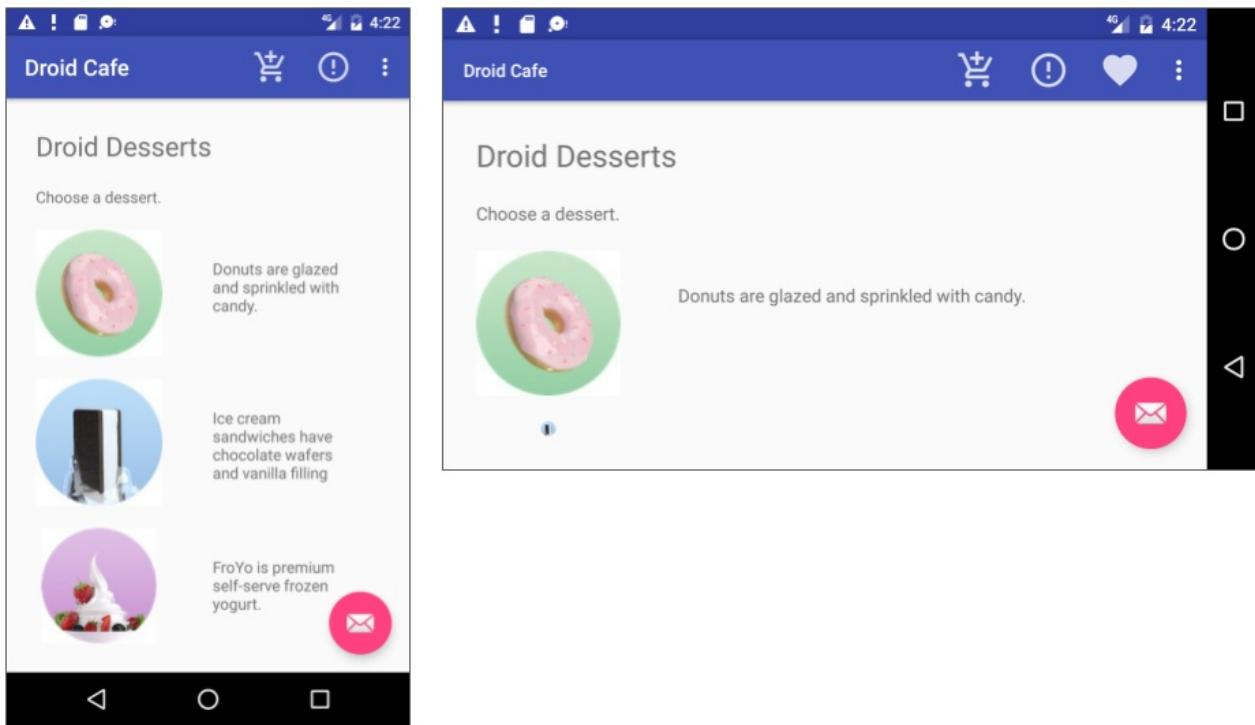
Status Item Attribute	Old Value	New Value
android:icon		"@drawable/ic_status_white"
app:showAsAction	"never"	"always"

Favorites Item Attribute	Old Value	New Value
android:icon		"@drawable/ic_favorites_white"
app:showAsAction	"never"	"ifRoom"

2. Run the app. You should now see at least two icons in the app bar: the icon for **Order** and the icon for **Status** as shown in the figure below. If your device or the emulator is displaying in vertical orientation, the **Favorites** and **Contact** options appear in the overflow menu.
3. 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 for **Order**, **Status**, and **Favorites**.

Tip: How many action buttons will fit in the app bar? It depends on the orientation and the size of the device screen. Fewer buttons appear in a vertical orientation, as shown on the left side of the figure below, compared to a horizontal orientation as shown on the right side

of the figure below. Action buttons may not occupy more than half of the main app bar's width.



Coding Challenge #1

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge 1: 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. For more information, see [Snackbar](#).

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 about floating action buttons, see [FloatingActionButton](#).

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. If you haven't already added the following method (in the previous lesson) for displaying a toast message, add it now:

```
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. You will add a `switch case` block to determine which menu item was selected, and what `message` to create for each selected item. (Rather than creating a `message` for each item, you could implement an event handler for each item that performs an action, 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);  
}
```

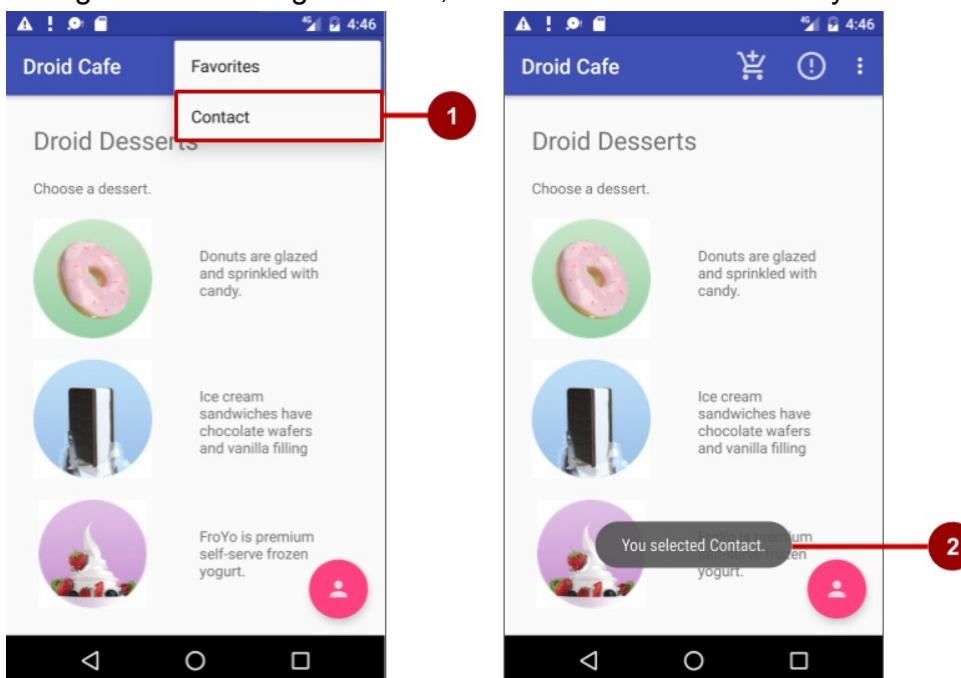
2. Replace the `if` statement and the assignment to `id` with the following `switch case` block that sets the appropriate `message` based on the menu item's `id`:

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_order:
            showToast(getString(R.string.action_order_message));
            return true;
        case R.id.action_status:
            showToast(getString(R.string.action_status_message));
            return true;
        case R.id.action_favorites:
            showToast(getString(R.string.action_favorites_message));
            return true;
        case R.id.action_contact:
            showToast(getString(R.string.action_contact_message));
            return true;
        default:
            // Do nothing
    }
    return super.onOptionsItemSelected(item);
}

```

3. Run the app. You should now see a different toast message on the screen, as shown on the right side of the figure below, based on which menu item you choose.



In the above figure:

1. Selecting the **Contact** item in the options menu.
2. The toast message that appears.

Solution code (includes coding challenge #1)**Android Studio project:** [DroidCafe Part 2](#)

Coding Challenge #2

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge 2: In the previous challenge, you changed the icon for the floating action button that appears at the bottom of the `MainActivity` screen in your app.

For this challenge:

1. Change the icon for the floating action button again, but this time to an appropriate icon for a map, such as the world icon.
2. In `MainActivity`, replace the action to display a snackbar with an implicit intent to launch the Maps app when the floating action button is tapped.
3. Add the following specific coordinates (for Google headquarters) and the zoom level (12) to a string resource called `google_mtv_coord_zoom12`:

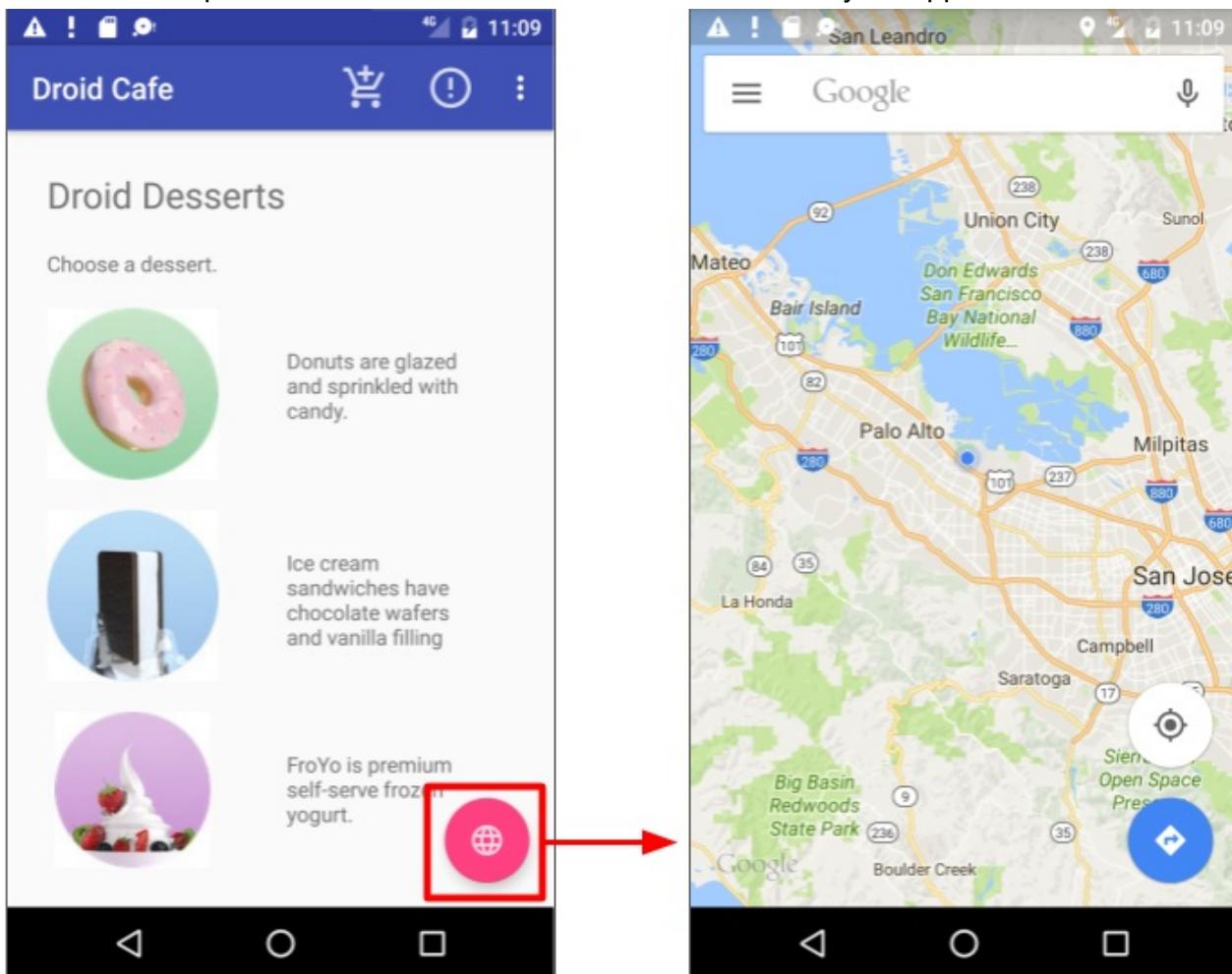
```
<string name="google_mtv_coord_zoom12">geo:37.422114, -122.086744?z=12</string>
```

4. Add the following method to start the Maps app, which passes the above string as `data` using an implicit intent:

```
public void displayMap() {  
    Intent intent = new Intent();  
    intent.setAction(Intent.ACTION_VIEW);  
    // Using the coordinates for Google headquarters.  
    String data = getString(R.string.google_mtv_coord_zoom12);  
    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.

After tapping the floating action button to go to the Maps app, as shown in the figure below, the user can tap the **Back** button below the screen to return to your app.



Solution code (includes coding challenge #2)

Android Studio Project: [DroidCafe Part 3](#)

You will finish the DroidCafe app in the next lesson.

Summary

In this practical, you learned how to do the following:

- Set 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.
- Use icons in a project:
 - Adding icons to a project and using them to show menu items in the app bar.
 - Challenge: Changing the icon for a floating action button, and changing the `Snackbar.make` code.
 - Challenge: Making an implicit intent to launch the Maps app with specific coordinates.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Menus](#)

Learn more

- Android Developer Reference:
 - [AppBarLayout](#)
 - [Toolbar](#)
 - [Menus](#)
- 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](#)

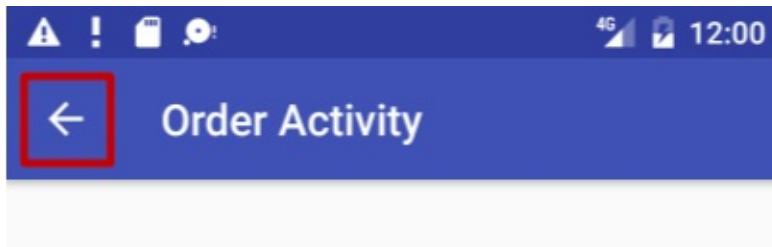
4.3: Using the App Bar and Tabs for Navigation

Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App Overview
- Task 1: Add an Up button for ancestral navigation
- Task 2: Use tab navigation with swipe views
- Coding challenges
- Summary
- Related concept
- Learn more

In the early stages of developing an app, you should determine the paths users should take through your app in order to do something, such as placing an order or browsing through content. Each path enables users to navigate across, into, and back out from the different tasks and pieces of content within the app.

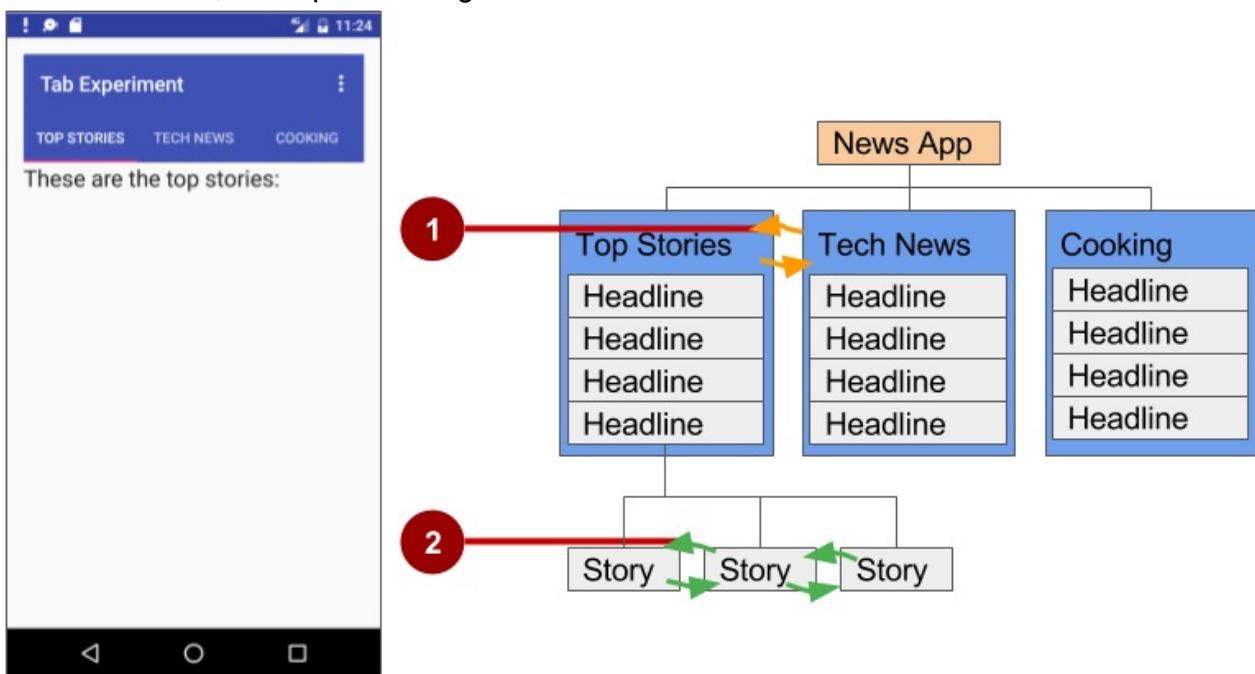
In this practical, you'll learn how to add an **Up** button (a left-facing arrow) to the app bar of your app, as shown below, to navigate from a child screen *up* to the parent screen.



The **Up** button is always used to navigate to the parent screen in the hierarchy. It differs from the **Back** button (the triangle at the bottom of the screen), which provides navigation to whatever screen the user viewed previously.

This practical also introduces *tab navigation*, in which 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, as shown in the diagram below. Tabs provide navigation to and from the sibling screens Top Stories, Tech News, and Cooking without having to navigate up to the parent. Tabs can also provide navigation to and from stories, which are sibling screens under the Top Stories parent.

Tabs are most appropriate for four or fewer sibling screens. The user can tap a tab to see a different screen, or swipe left or right to see a different screen.



In the above figure:

1. Lateral navigation from one category screen to another
2. Lateral navigation from one story screen to another

What you should already KNOW

From the previous chapters, you should be able to:

- Create and run apps in Android Studio.
- Create and edit UI elements using the Layout Editor, entering XML code directly, and accessing elements from your Java code.
- Add menu items and icons to the options menu in the app bar.

What you will LEARN

In this practical, you will learn to:

- Add the **Up** button to the app bar.
- Set up an app with tab navigation and swipe views.

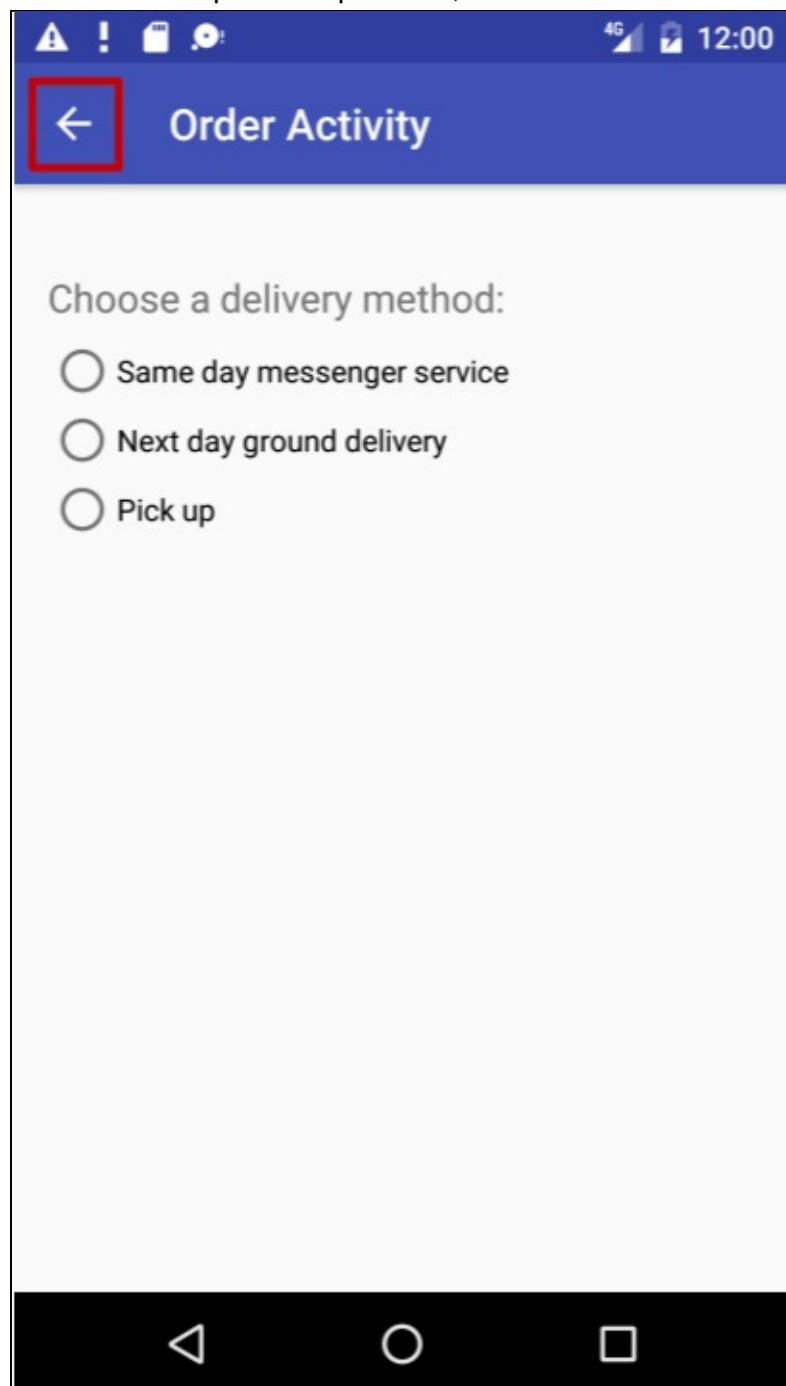
What you will DO

- Continue adding features to the Droid Cafe project from the previous practical.
- Provide the **Up** button in the app bar to navigate to the previous screen within an activity.
- Create a new app with tabs for navigating activity screens that can also be swiped.

App Overview

In the previous practical you created an app called Droid Cafe in three parts, using the Basic Activity template. This template also provides an app bar at the top of the screen. You will learn how to add an **Up** button (a left-facing arrow) to the app bar for up navigation from the second activity (`OrderActivity`) to the main activity (`MainActivity`). This will complete the Droid Cafe app.

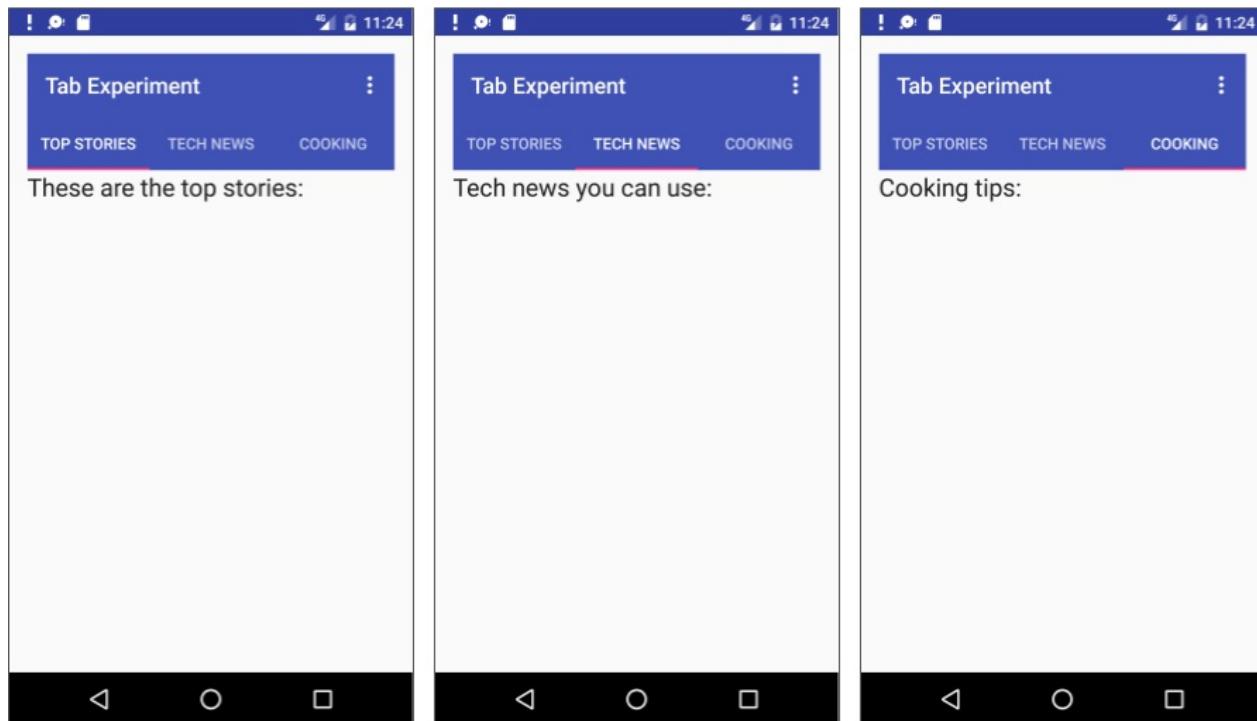
To start the project from where you left off in the previous practical, download the Android



Studio project [DroidCafe Part 3](#).

The second app you will create for tab navigation will show three tabs below the app bar to navigate to sibling screens. As the user taps a tab, the screen shows a content screen depending on which tab is tapped. The user can also swipe left and right to visit the content screens. Swiping views is handled automatically by the ViewPager class.

The second app you will create for tab navigation will show three tabs below the app bar to navigate to sibling screens. As the user taps a tab, the screen shows a content screen depending on which tab is tapped. The user can also swipe left and right to visit the content screens. Swiping views is handled automatically by the ViewPager class.



Task 1. Add an Up button for ancestral navigation

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 way to do this is to provide an **Up** button on the app bar for all activities that are children of the parent activity.

The **Up** button provides ancestral "up" navigation, enabling the user to go *up* from a child page to the parent page. The **Up** button is the left-facing arrow on the left side of the app bar, as shown on the left side of the figure below.

When the user touches the **Up** button, the app navigates to the parent activity. The diagram on the right side of the figure below shows how the **Up** button is used to navigate within an app based on the hierarchical relationships between screens.

1. Navigating from the first-level siblings to the parent.
2. Navigating from second-level siblings to the first-level child screen acting as a parent screen

Tip: The **Back** button (the triangle at the bottom of the screen) differs from the **Up** button. The **Back** button provides navigation to whatever screen you viewed previously. If you have several child screens that the user can navigate through using a lateral navigation pattern (as described in the next section), the **Back** button would send the user back to the previous child screen, not to the parent screen. Use an **Up** button if you want to provide ancestral navigation from a child screen back to the parent screen. 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` attribute for a title for the activity screen, such as `"Order Activity"`:

1. If you don't already have the Droid Cafe app open from the previous practical, download the Android Studio project [DroidCafe Part 3](#) and rename the project to **DroidCafe**.
2. Open the **DroidCafe** project.
3. Open **AndroidManifest.xml**.
4. Change the activity element for `OrderActivity` to the following:

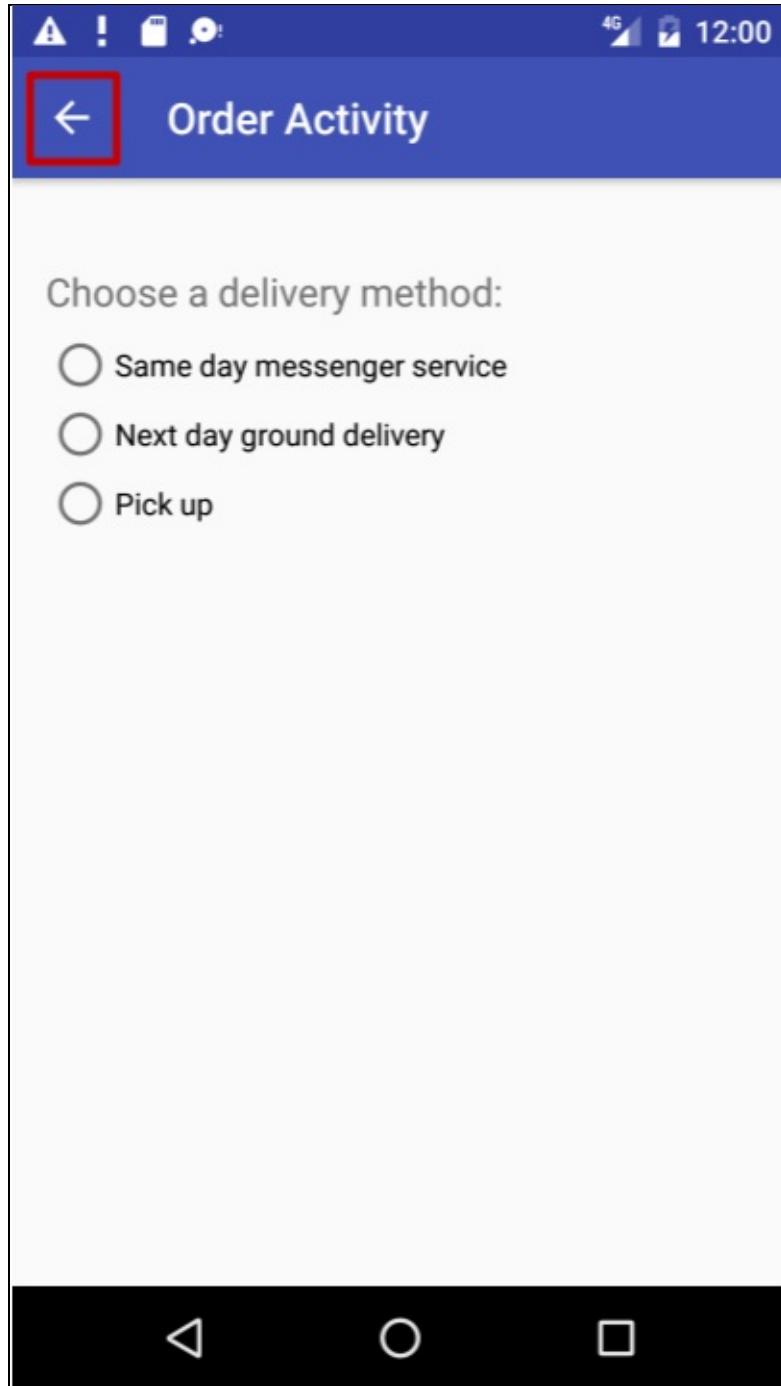
```
<activity android:name=".OrderActivity"
    android:label="Order Activity"
    android:parentActivityName="com.example.android.
                                droidcafe.MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

5. Extract the `android:label` value `"Order Activity"` to a string resource named `title_activity_order`.
6. Run the app.

```
title_activity_order .
```

6. Run the app.

The Order Activity screen now includes the **Up** button (highlighted in the figure below) in the app bar to navigate back to the parent activity.

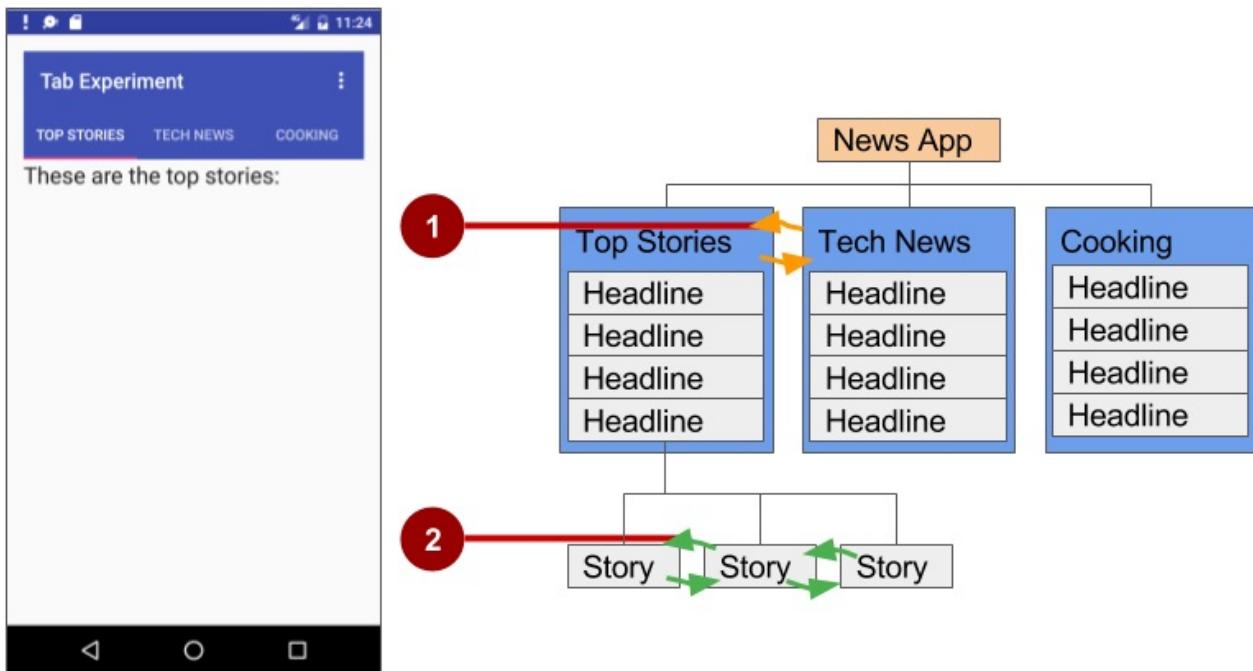


Solution code:

Android Studio project: [DroidCafe](#)

Task 2. Use tab navigation with swipe views

With lateral navigation, you enable the user to go from one sibling to another (at the same level in a multitier hierarchy). For example, if your app provides several categories of stories (such as Top Stories, Tech News, and Cooking, as shown in the figure below), you would want to provide your users the ability to navigate from one category to the next, without having to navigate back up to the parent screen. Another example of lateral navigation is the ability to swipe left or right in a Gmail conversation to view a newer or older one in the same inbox.



In the above figure:

1. Lateral navigation from one category screen to another
2. Lateral navigation from one story screen to another

You can implement lateral navigation with *tabs* that represent each screen. Tabs appear across the top of a screen, as shown on the left side of the figure above, in order to provide 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. Tab navigation is often combined with the ability to swipe child screens left-to-right and right-to-left.

The primary class used for displaying tabs is [TabLayout](#) in the Android Design Support Library. It provides a horizontal layout to display tabs. You can show the tabs below the app bar, and use the [PagerAdapter](#) class to populate screens "pages" inside of a [ViewPager](#). [ViewPager](#) is a layout manager that lets the user flip left and right through screens. This is a common pattern for presenting different screens of content within an activity—use an *adapter* to fill the content screen to show in the activity, and a *layout manager* that changes the content screens depending on which tab is selected.

You supply an implementation of a [PagerAdapter](#) to generate the screens that the view shows. ViewPager is most often used in conjunction with [Fragment](#). By using fragments, you have a convenient way to manage the lifecycle of each screen "page".

To use classes in the Android Support Library, add `com.android.support:design:xx.xx.x` (in which `xx.xx.x` is the newest version) to the **build.gradle (Module: app)** file.

The following 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.

2.1 Create the layout for tab navigation

1. Create a new project using the Empty Activity template. Name the app **Tab Experiment**.
2. Edit the **build.gradle (Module: app)** file, and add the following lines (if they are not already added) to the `dependencies` section:

```
compile 'com.android.support:design:25.0.1'  
compile 'com.android.support:support-v4:25.0.1'
```

If Android Studio suggests a version with a higher number, edit the above lines to update the version. Also, if Android Studio suggests a newer version of `compileSdkVersion`, `buildToolsVersion`, and/or `targetSdkVersion`, edit them to update the version.

3. In order to use a Toolbar rather than an action bar and app title, add the following statements to the **res > values > styles.xml** file to hide the action bar and the title:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">  
    ...  
    <item name="windowActionBar">false</item>  
    <item name="windowNoTitle">true</item>  
</style>
```

4. In the **activity_main.xml** layout, remove the `TextView` supplied by the template, and add a `Toolbar`, a `TabLayout`, and a `ViewPager` within the root layout. The layout should look like the code below.

As you type the `app:popupTheme` attribute for `Toolbar` as shown below, `app` will be in red if you didn't add the following statement to `RelativeLayout`:

```
<RelativeLayout xmlns:app="http://schemas.android.com/apk/res-auto"
```

You can click on `app` and press **Option-Return**, and Android Studio automatically adds the statement.

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"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.android.tabexperiment.MainActivity">

    <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"/>

</RelativeLayout>
```

2.2 Create a layout and class for each fragment

1. Add a fragment representing each tabbed screen: **TabFragment1**, **TabFragment2**, and **TabFragment3**. To add each fragment:
 - i. Click **com.example.android.tabexperiment** in the project view.
 - ii. Choose **File > New > Fragment > Fragment (Blank)**.
 - iii. Name the fragment **TabFragment1**.
 - iv. Check the "Create layout XML?" option, and change the Fragment Layout Name for the XML file to **tab_fragment1**.
 - v. *Uncheck* the "Include fragment factory methods?" and the "include interface callbacks?" options. You don't need these methods.
 - vi. Click **Finish**.
- vii. Repeat the above steps, using **TabFragment2** and **TabFragment3** for Step C, and **tab_fragment2** and **tab_fragment3** for Step D.

Each fragment (**TabFragment1**, **TabFragment2**, and **TabFragment3**) is created with its class definition set to extend `Fragment`. Also, each fragment *inflates* the layout associated with the screen (`tab_fragment1`, `tab_fragment2`, and `tab_fragment3`), using the familiar resource-inflate design pattern you learned in a previous chapter with the options menu.

For example, **TabFragment1** 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);  
    }  
}
```

Android Studio automatically includes the following import statements:

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

2. Edit each fragment layout XML file (**tab_fragment1**, **tab_fragment2**, and **tab_fragment3**):
 - i. Change the Root Tag to `RelativeLayout`.
 - ii. Add a `TextView` with text such as "These are the top stories".
 - iii. Set the text appearance with `android:textAppearance="? android:attr/textAppearanceLarge"`.
 - iv. Repeat the above steps for each fragment layout XML file, entering different text for the `TextView` in step B.
3. Examine each fragment layout XML file. For example, **tab_fragment1** should look like this:

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="These are the top stories:"
        android:textAppearance="?android:attr/textAppearanceLarge"/>

</RelativeLayout>
```

4. In the fragment layout XML file **tab_fragment1**, extract the string for `"These are the top stories:"` into the string resource `tab_1`. Do the same for the strings in **tab_fragment2**, and **tab_fragment3**.

2.3 Add a PagerAdapter

The adapter-layout manager pattern lets you provide different screens of content within an activity—use an *adapter* to fill the content screen to show in the activity, and a *layout manager* that changes the content screens depending on which tab is selected.

1. Add a new `PagerAdapter` class to the app that extends `FragmentStatePagerAdapter` and defines the number of tabs (`mNumOfTabs`):

```
public class PagerAdapter extends FragmentStatePagerAdapter {
    int mNumOfTabs;
    public PagerAdapter(FragmentManager fm, int NumOfTabs) {
        super(fm);
        this.mNumOfTabs = NumOfTabs;
    }
}
```

While entering the above code, Android Studio automatically imports:

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

If **FragmentManager** in the above code is in red, a red lightbulb icon should appear when you click on it. Click the lightbulb icon and choose **Import class**. Import choices appear. Select the following import choice:

FragmentManager (android.support.v4)

Choosing the above imports the following:

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

Also, Android Studio underlines the class definition for **PagerAdapter** and, if you click on **PagerAdapter**, displays a red bulb icon. Click the icon and choose **Implement Methods**, and then click **OK** to implement the already selected `getItem()` and `getCount()` methods.

2. Change the newly added `getItem()` method to the following, which uses a `switch case` block to return the fragment to show based on which tab is clicked

```
@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;
    }
}
```

3. Change the newly added `getCount()` method to the following to return the number of tabs:

```
@Override  
public int getCount() {  
    return mNumOfTabs;  
}
```

2.4 Inflate the Toolbar and TabLayout

Since you are using tabs that fit underneath the app bar, you have set up the app bar and `Toolbar` in the `activity_main.xml` layout in the first step of this task. Now you need to inflate the `Toolbar` (using the same method described in a previous chapter about the options menu), and create an instance of `TabLayout` to position the tabs.

1. Inflate the `Toolbar` in the `onCreate()` method in **MainActivity.java**:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);  
    setSupportActionBar(toolbar);  
  
    // Create an instance of the tab layout from the view.  
    ...  
}
```

In the above code, `Toolbar` is in red, and a red lightbulb icon should appear. Click the icon and choose **Import class**. Import choices appear. Select **Toolbar (android.support.v7.widget.Toolbar)**, and the following import statement appears in your code:

```
import android.support.v7.widget.Toolbar;
```

2. Open **strings.xml**, and create the following string resources:

```
<string name="tab_label1">Top Stories</string>  
<string name="tab_label2">Tech News</string>  
<string name="tab_label3">Cooking</string>
```

3. At the end of the `onCreate()` method, 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.
tabLayout.addTab(tabLayout.newTab().setText(R.string.tab_label1));
tabLayout.addTab(tabLayout.newTab().setText(R.string.tab_label2));
tabLayout.addTab(tabLayout.newTab().setText(R.string.tab_label3));
// Set the tabs to fill the entire layout.
tabLayout.setTabGravity(TabLayout.GRAVITY_FILL);

// Use PagerAdapter to manage page views in fragments.
...
```

2.5 Use PagerAdapter to manage screen views

1. Below the code you added to the `onCreate()` method in the previous task, add the following code to use `PagerAdapter` to manage screen (page) views in the fragments:

```
...
// 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 = new PagerAdapter
    (getSupportFragmentManager(), tabLayout.getTabCount());
viewPager.setAdapter(adapter);
// Setting a listener for clicks.
...
```

2. At the end of the `onCreate()` method, set a listener ([TabLayoutOnPageChangeListener](#)) to detect if a tab is clicked, and create the `onTabSelected()` method to set the `ViewPager` to the appropriate tabbed screen. The code should look as follows:

```
...
// 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) {
    }
});
}
```

3. Run the app. Tap each tab to see each "page" (screen). You should also be able to swipe left and right to visit the different "pages".

Solution code

Android Studio Project: [Tab Experiment \(including coding challenge 1\)](#)

Android Studio Project: [NavDrawer Experiment Experiment \(coding challenge 2\)](#)

Coding challenges

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge 1: When you created the layout for tab navigation in the first step of the previous lesson, you established a `Toolbar` for the app bar in the `activity_main.xml` layout file. Add an options menu to the app bar as a challenge.

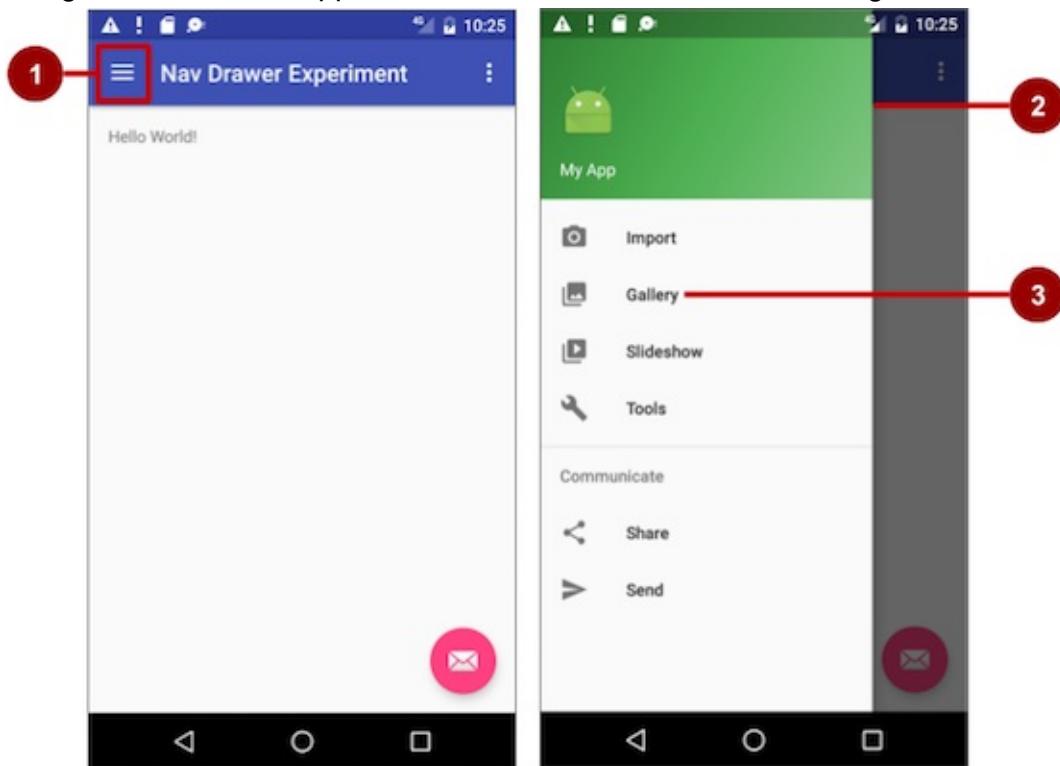
To start, you will want to create the `menu_main.xml` file, and add menu items for the options menu. You must add at least one menu item, such as **Settings**.

You can inflate the options menu in the `Toolbar` by adding the `onCreateOptionsMenu()` method, as you did in a previous lesson on using the options menu.

Finally you can detect which options menu item is checked by using the [onOptionsItemSelected\(\)](#) method.

Challenge 2: Create a new app with a navigation drawer. When the user taps a navigation drawer choice, close the drawer and display a toast message showing which choice was selected.

A *navigation drawer* is a panel that usually displays navigation options on the left edge of the screen, as shown on the right side of the figure below. It is hidden most of the time, but is revealed when the user swipes a finger from the left edge of the screen or touches the navigation icon in the app bar, as shown on the left side of the figure below.



In the above figure:

1. Navigation icon in the app bar
2. Navigation drawer
3. Navigation drawer menu item

To make a navigation drawer in your app, you need to do the following:

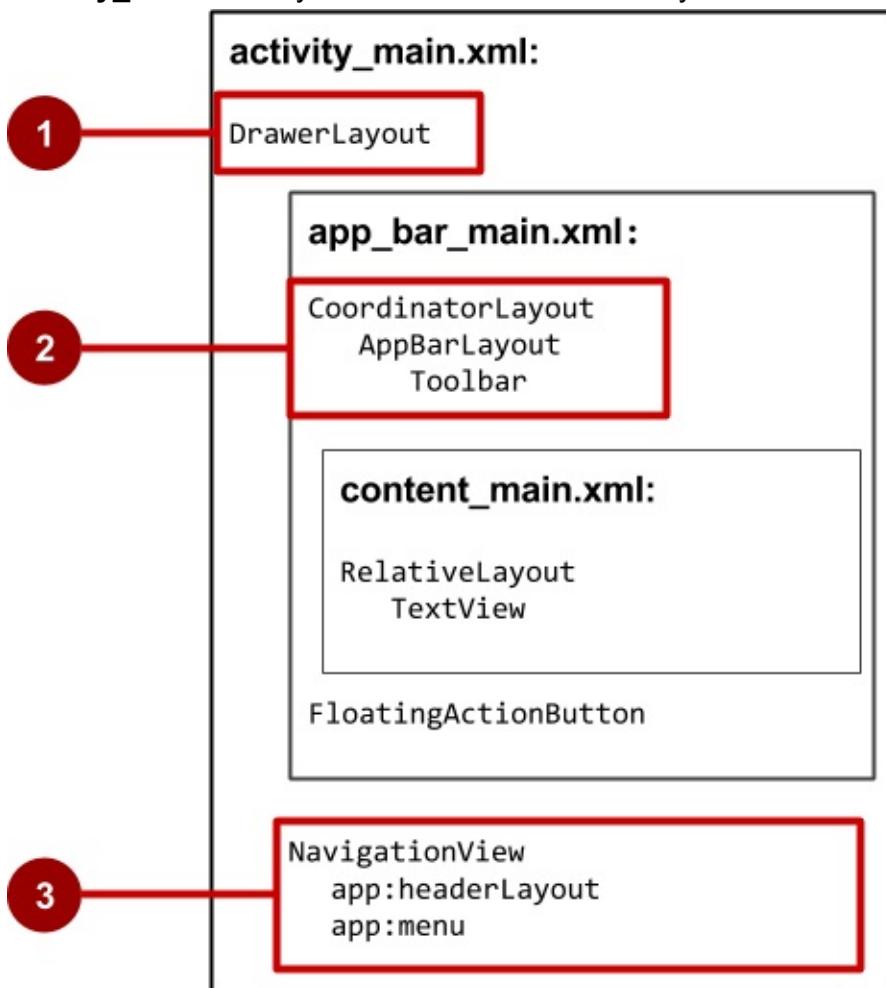
1. Create the following layouts:
 - A navigation drawer as the activity layout's root view.
 - A navigation view for the drawer itself.
 - An app bar layout that will include a navigation icon button.
 - A content layout for the activity that displays the navigation drawer.
 - A layout for the navigation drawer header.
2. Populate the navigation drawer menu with item titles and icons.

3. Set up the navigation drawer and item listeners in the activity code.
4. Handle the navigation menu item selections.

To create a navigation drawer layout, use the [DrawerLayout](#) APIs available in the [Support Library](#). For design specifications, follow the design principles for navigation drawers in the [Navigation Drawer](#) design guide.

To add a navigation drawer, use a `DrawerLayout` as the root view of your activity's layout. Inside the `DrawerLayout`, add one view that contains the main content for the screen (your primary layout when the drawer is hidden) and another view, typically a [NavigationView](#), that contains the contents of the navigation drawer.

Tip: To make your layouts simpler to understand, use the `include` tag to include an XML layout within another XML layout. The figure below is a visual representation of the `activity_main.xml` layout and its included XML layouts:



In the above figure:

1. `DrawerLayout` is the root view of the activity's layout.
2. The included `app_bar_main` uses a `CoordinatorLayout` as its root, and defines the app bar layout with a `Toolbar` which will include the navigation icon to open the drawer.
3. The `NavigationView` defines the navigation drawer layout and its header, and adds

menu items to it.

Summary

- Add Up-button navigation to a child activity by declaring the activity's parent in the `AndroidManifest.xml` file.
- Set up tab navigation:
 - Tabs are a good solution for "lateral navigation" between sibling views.
 - The primary class used for tabs is `TabLayout` in the design support library.
 - You should use the adapter pattern when populating tabs (pages) with data.
 - A `ViewPager` is a layout manager that allows the user to flip left and right through pages of data.
 - `ViewPager` is most often used in conjunction with fragments.
 - There are two standard adapters for using `ViewPager`: `FragmentPagerAdapter` and `FragmentStatePagerAdapter`.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Screen Navigation](#)

Learn more

- Android Developer Reference:
 - [Providing Up Navigation](#)
 - [TabLayout](#)
 - [Creating Swipe Views with Tabs](#)
 - [Navigation Drawer](#)
 - [DrawerLayout](#)
 - [Support Library](#)
- Android Developers Blog: [Android Design Support Library](#)
- Other
 - [AndroidHive: Android Material Design working with Tabs](#)
 - [Truiton: Android Tabs Example – With Fragments and ViewPager](#)

4.4: 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](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

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 of scrollable lists.

Earlier in this class, you used ScrollView to perform scrolling of other Views. ScrollView is easy to use, but it is not recommended for production use, especially for long lists of scrollable items.

RecyclerView is a subclass of ViewGroup and is a more resource-efficient way to display scrollable lists. Instead of creating a view for each item, whether or not it's visible, RecyclerView creates a limited number of list items and reuses them for visible content.

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 \(FAB\)](#), the pink button in the screenshot below. A floating action buttons can be used for common actions, or 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 how to:

- Create a Hello World app with Android Studio.
- Implement different layouts for apps.
- Create and using string resources.
- Add an onClick handler to a view.

What you will LEARN

In this practical, you will learn to:

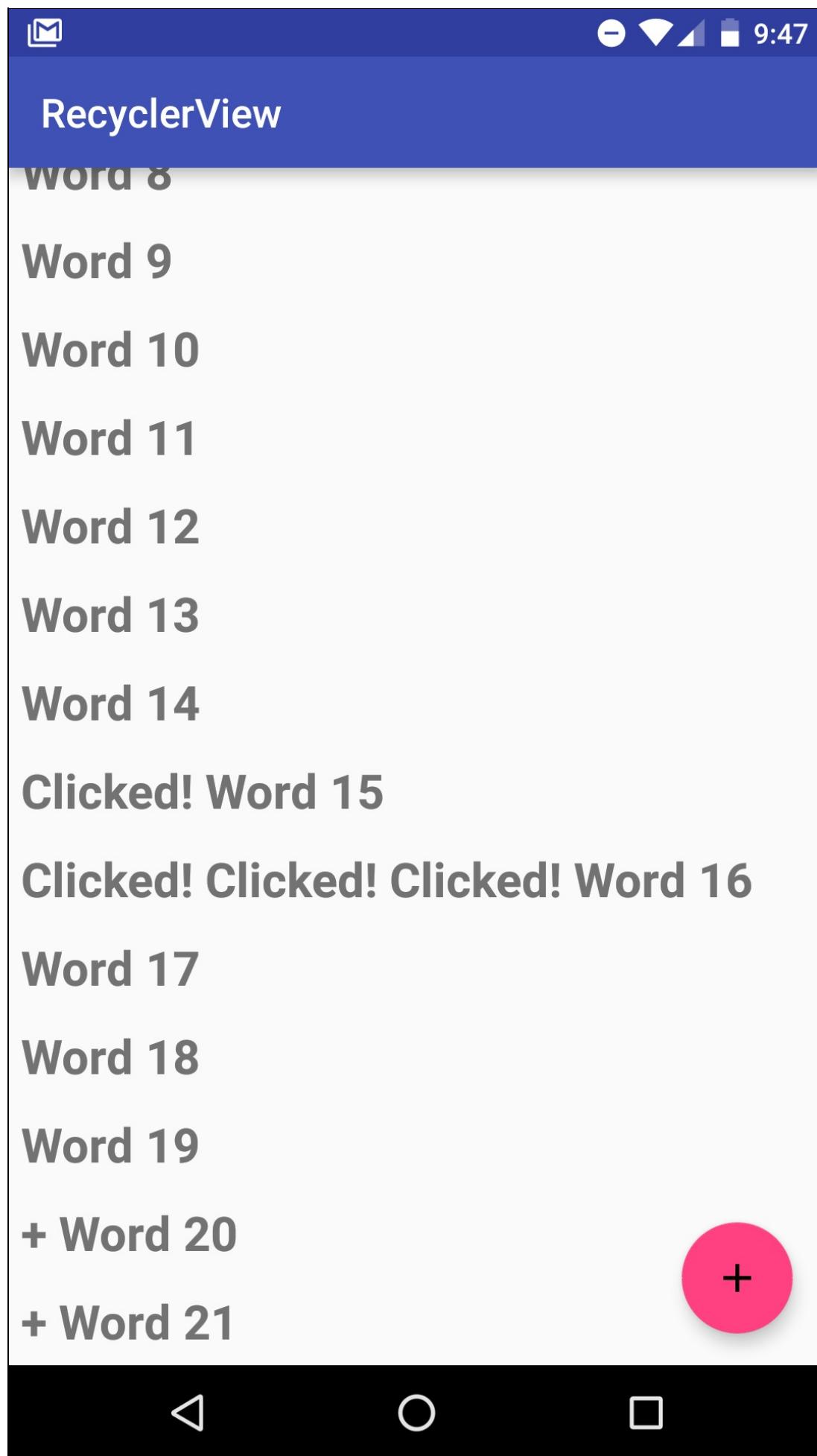
- Use the RecyclerView class to display items in a scrollable list.
- Dynamically add items to the RecyclerView as they become visible through scrolling.
- Perform an action when the user taps a specific item.
- Show a floating action button and perform an action when the user taps it.

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 with the list items. Use a floating action button to let the user 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.
- There is no user input of words for this app.

Task 1. Create and configure a new project

In this task, 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

2. Run your app on an emulator or hardware device. You should see the "RecyclerView" title and "Hello World" in a blank view.

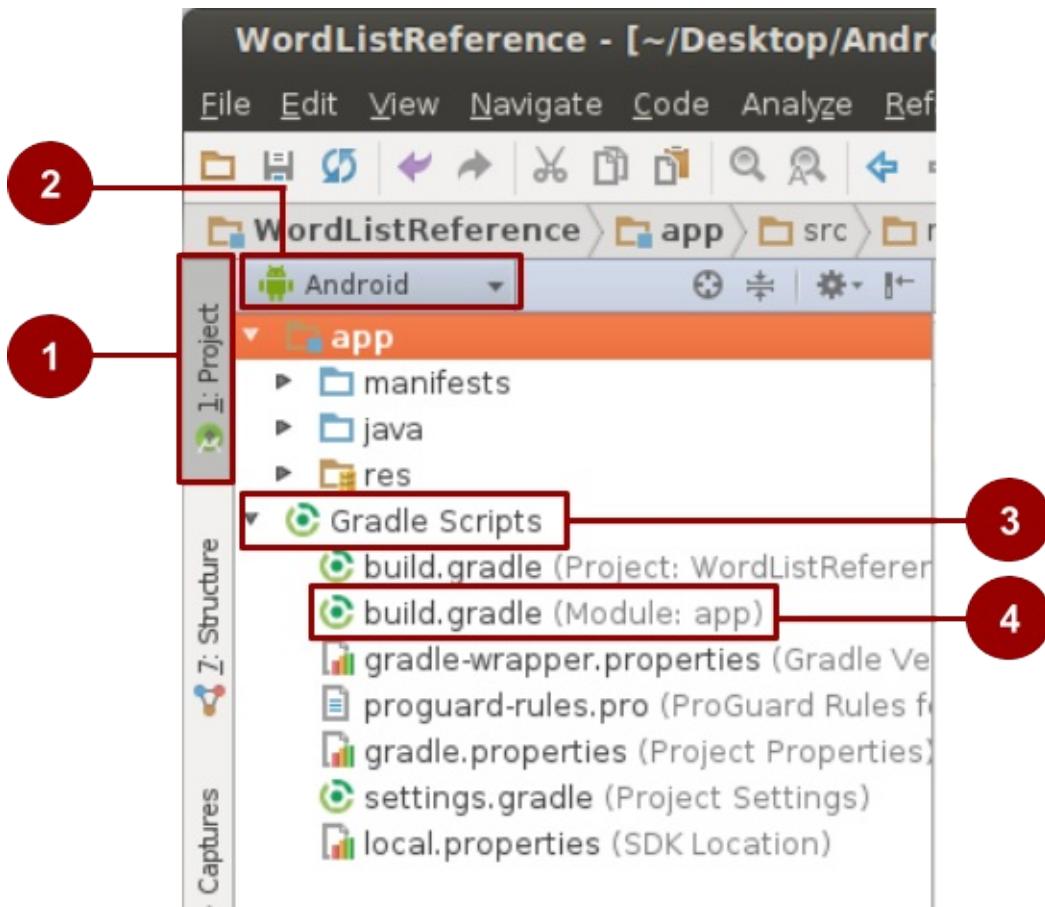
1.2. Add support libraries to the build file

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.

Follow these steps and refer to the screenshot:

1. In Android Studio, in your new project, make sure you are in the **Project** pane (1) and in the **Android** view (2).
2. In the hierarchy of files, find the **Gradle Scripts** folder (3).
3. Expand **Gradle Scripts**, if necessary, and open the **build.gradle (Module: app)** file (4).



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 dependencies section:

```
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. Run your app. You should see the same "RecyclerView" 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. Your file may be slightly different and your entries may have a different 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 main activities `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 your app to use real data in that later lesson.

2.1. Add code to create data

In this task you will dynamically create a linked list of twenty word strings that end in increasing numbers, such that ["Word 1", "Word 2", "Word 3",].

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 private member variable for the `mWordList` linked list.
3. Add an integer counter `mCount` 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 linked list.
6. Run your app to make sure there are no errors.

The app UI has not changed, but you should see a list of log messages in logcat, 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());
}
```

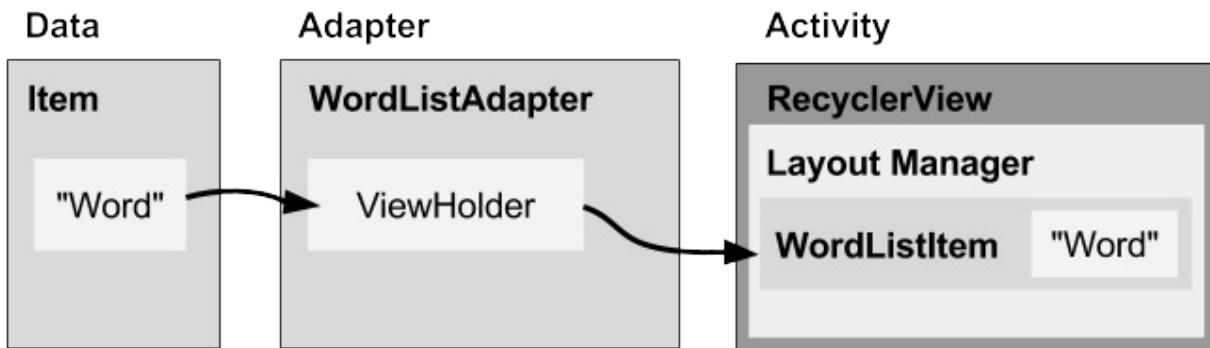
Task 3. Create a RecyclerView

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 parts:

- **Data.** You will use the mWordList.
- **A RecyclerView.** The scrolling list that contains the list items.
- **Layout for one item of data.** All list items look the same.
- **A layout manager.** The layout manager handles the organization (layout) of user interface components in a view. You have already used the LinearLayout in a previous practical where the Android system handles the layout for you. RecyclerView requires an explicit layout manager to manage the arrangement of list items contained within it. This layout could be vertical, horizontal, or a grid. You will use a vertical linear layout manager provided by Android.
- **An adapter.** The adapter connects your data to the RecyclerView. It prepares the data in a view holder. You will create an adapter that inserts into and updates your generated words in your views.
- **A view holder.** Inside your adapter, you will create a ViewHolder class that contains the view information for displaying one item from the item's layout.

The diagram below shows the relationship between the data, the adapter, the view holder, and the layout manager.



Implementation steps overview

To implement these pieces, you will need to:

1. Create the XML layout for the "RecyclerView" app (activity_main.xml).
2. Create the XML layout used to lay out one list item, which is WordListItem (wordlist_item.xml).
3. Create an adapter (WordListAdapter) with a view holder (WordViewHolder). Implement the method that takes the data, places it in a view holder, and let's the layout manager know to display it.
4. In the onCreate method of 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 build.gradle (Module:app) and verify that the recycler view dependency exists.

```
compile 'com.android.support:recyclerview-v7:24.1.1'
```

2. Open activity_main.xml.
3. 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>

```

4. 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.

5. 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>

```

6. Give your RecyclerView the following properties:

Attribute	Value
android:id	"@+id/recyclerview"
android:layout_width	match_parent
android:layout_height	match_parent

7. 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>

```

3.2. Create the layout for one list item

The adapter needs the layout for one item in the list. All the items use the same layout. You need to specify that list item layout in a separate layout resource file, because it is used 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` and click **OK**.
3. In Text mode, change the LinearLayout that was created with the file to match with the following attributes. Extract resources as you go.

Attribute	Value
<code>android:layout_width</code>	"match_parent"
<code>android:layout_height</code>	"wrap_content"
<code>android:orientation</code>	"vertical"
<code>android:padding</code>	"6dp"

4. Add a TextView for the word to the LinearLayout:

Attribute	Value
<code>android:id</code>	"@+id/word"
<code>android:layout_width</code>	"match_parent"
<code>android:layout_height</code>	"wrap_content"
<code>android:textSize</code>	"24sp"
<code>android:textStyle</code>	"bold"

3.3 Create a style from the TextView attributes

You can use styles to allow elements to share groups of 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:

1. While you have `wordlist_item.xml` open, hover the mouse over the `TextView` section you just created and **Right-click > Refactor > Extract > Style**.
2. 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**.
3. When prompted, apply the style to the **Whole Project**.
4. Find and examine the `word_title` style in `values/styles.xml`.
5. Go back to `wordlist_item.xml`. The text view now references the style instead of using individual styling properties.
6. Run your app. Since you have removed the default "Hello World" text view, you should see the "RecyclerView" 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>
```

3.4. 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. Therefore, the adapter contains a view holder (from the [ViewHolder](#) class) that describes an item view and its position within 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.recyclerview` and select **New > Java Class**.
2. Name the class `WordListAdapter`.
3. Give `WordListAdapter` the following signature:

```
public class WordListAdapter extends  
    RecyclerView.Adapter<WordListAdapter.WordViewHolder> {}
```

`WordListAdapter` extends a generic adapter for `RecyclerView` to use a view holder that is specific for your app and defined inside `WordListAdapter`. `WordViewHolder` shows an error, because you have not defined it yet.

4. Click on the class declaration (`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 choose which methods 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 `WordViewHolder`, which hasn't been implemented yet.

3.5 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 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;
}
```

- Run your app to make sure you have no errors. You will still see only a blank view. Take note of the **E/RecyclerView: No adapter attached; skipping layout** warning in logcat.

3.6 Storing your data in the adapter

- To hold your data in the adapter, create a private linked list of strings in WordListAdapter and call it mWordList.

```
private final LinkedList<String> mWordList;
```

- 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 the 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.

- Create a member variable for the inflator in WordListAdapter.

```
private LayoutInflator mInflater;
```

- Implement the constructor for WordListAdapter. The constructor needs to have a context parameter, and a linked list of words with the app's data. The method needs to instantiate a layout inflator for `mInflater` and set `mWordList` to the passed in data.

```
public WordListAdapter(Context context, LinkedList<String> wordList) {
    mInflater = LayoutInflater.from(context);
    this.mWordList = wordList;
}
```

- Fill out the onCreateViewHolder() method with the code below. The onCreateViewHolder method is similar to the onCreate method. It 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);
}

```

- 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);
}

```

- Run your app to make sure there are no errors. You will still see the "E/RecyclerView: No adapter attached; skipping layout" warning. You will fix that in the next task.

3.7. 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.

- Open MainActivity.java
- Add member variables to MainActivity for the RecyclerView and the adapter.

```

private RecyclerView mRecyclerView;
private WordListAdapter mAdapter;

```

- 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 that 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));

```

- Run your app. 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 can  
// update the RecyclerView to display the data.  
mAdapter.notifyDataSetChanged();
```

5. Connect the onClickListener with the view. Add this code to the WordViewHolder constructor (below the "this.mAdapter = adapter" line):

```
itemView.setOnClickListener(this);
```

6. Run your app. Click on items to see their text change.

Solution code: [WordListAdapter.java](#) and [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.

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. You will learn more in the chapter about Material Design. 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:

- **<code>@+id/fab </code>** 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 the **Asset Type is Material Icon**.
4. Click the icon button next to **Icon**.
5. In the Content section find the + sign. The resource name is **ic_add_black_24dp**.
6. Leave everything else unchecked and click **Next**.
7. Click **Finish**.

- 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 to the FAB an onClick listener 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 end of the wordList.  
        mWordList.addLast("Word " + wordListSize);  
        // Notify the adapter, that the data has changed so it can  
        // update the RecyclerView to display the data.  
        mRecyclerView.getAdapter().notifyItemInserted(wordListSize);  
        // Scroll to the bottom.  
        mRecyclerView.smoothScrollToPosition(wordListSize);  
    }  
});
```

- Run your app. To test your app do the following:
 - Scroll the list of words.
 - Click on items.
 - Add items by clicking on the FAB.
 - 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

Android Studio project: [RecyclerView](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: Creating a click listener for each item in the list is easy, but it 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.

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.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [RecyclerView](#)

Learn more

- [CoordinatorLayout](#)
- [RecyclerView](#)
- [Android Support Library](#)
- [Creating Lists and Cards](#)
- [RecyclerView Animations and Behind the Scenes \(Android Dev Summit 2015\)](#)

5.1: 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](#)
- [Summary](#)
- [Related concepts](#)
- [Learn more](#)

In this chapter, you will learn how to reduce your code, increase its 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 basic concepts of the Activity lifecycle, and how to:

- Create and run apps in Android Studio.
- Create and edit UI elements using the Layout Editor, XML, and code.
- Add onClick functionality to a button.
- Update views at runtime.
- Add menu items with onClick functionality.
- Pass data between activities using Intents.

What you will LEARN

In this practical, you will learn to:

- Define a style.
- Apply a style to a view.

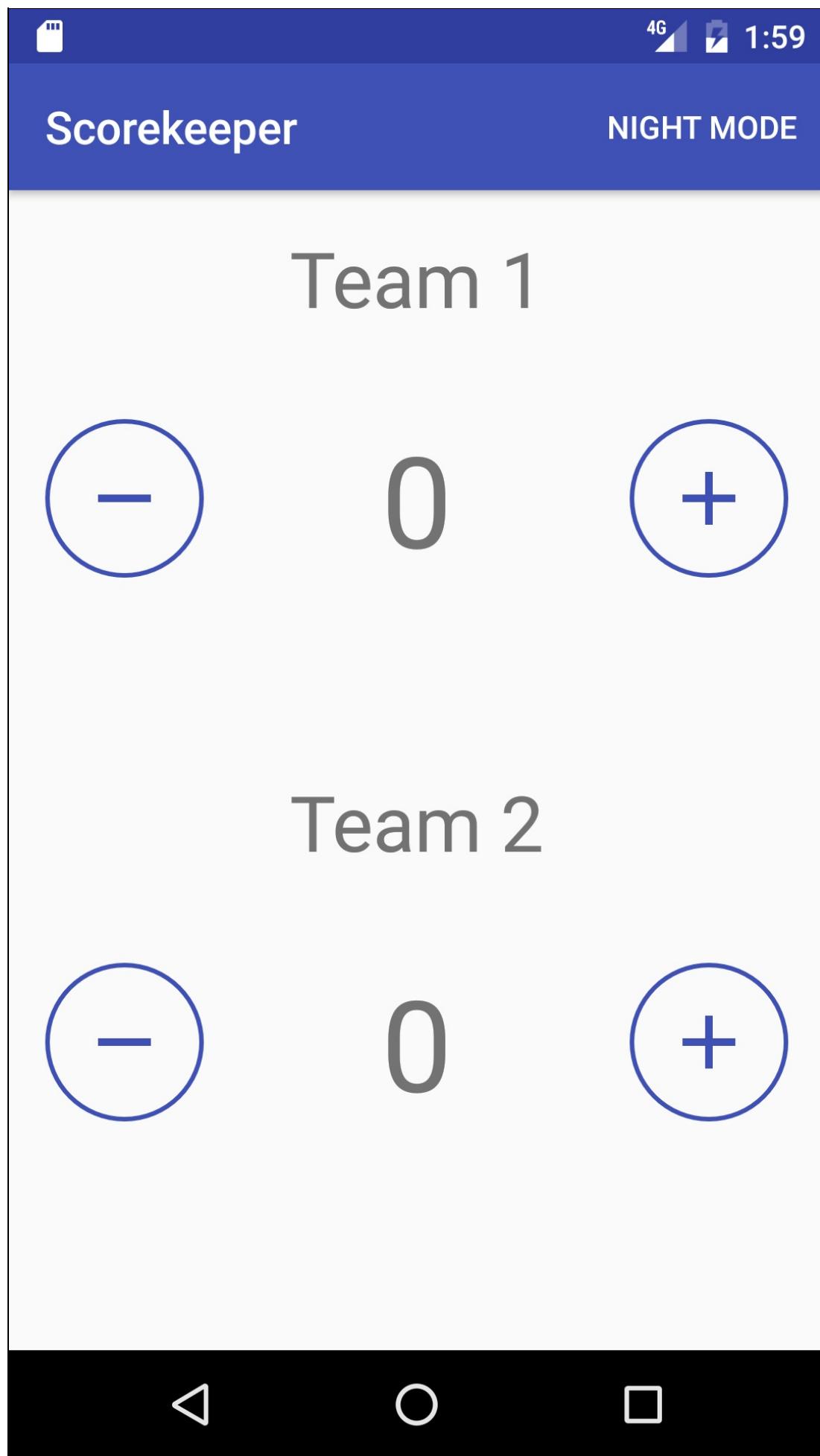
- Apply a theme to an activity or application both in XML and programmatically.
- Use drawable resources.

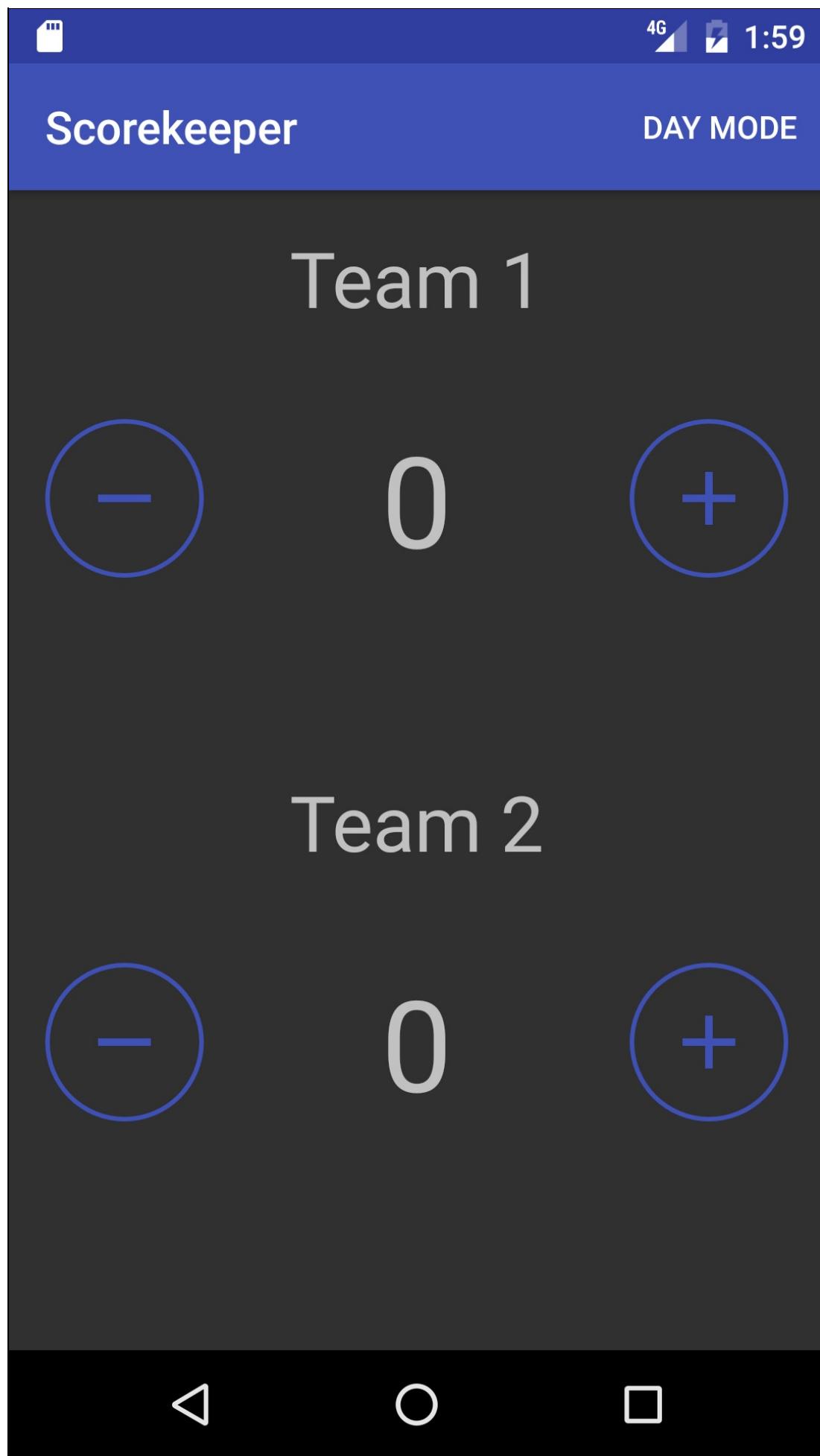
What you will DO

- Create a new Android app and add buttons and TextView views 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 text views 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 root view to a LinearLayout and add the following attributes (without removing the existing attributes):

Attribute	Value
android:orientation	"vertical"

Define the score containers:

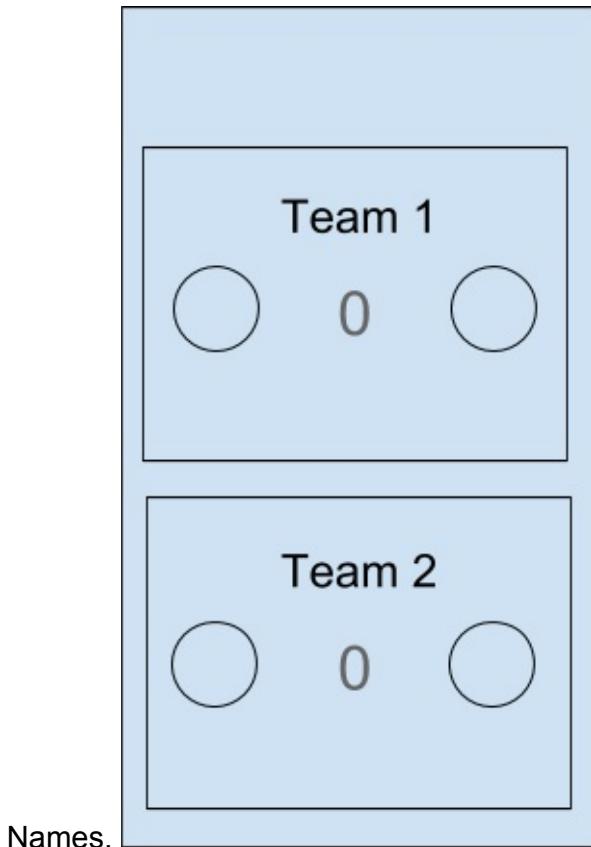
1. Inside the LinearLayout, add two RelativeLayout view groups (one to contain the score for each team) with the following attributes:

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" attribute 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 to a list of material icon files. 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 [text views](#) to read "0" and the team [text views](#) to read "Team 1" and "Team 2".
2. Add the following attributes to your left [image buttons](#):

```
    android:src="@drawable/ic_minus"  
    android:contentDescription="Minus Button"
```

3. Add the following attributes to your right image buttons:

```
    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 the string.xml file. This allows for your app to be easily localized into different languages. To learn how to extract string resources, see the [Extracting Resources](#) section in 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"
    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"/>
    </RelativeLayout>
</LinearLayout>
```

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.
- A reference to your score TextViews in MainActivity so you can update the scores.
- In the `onCreate()` method of MainActivity, find your score TextViews by id and assign them to member variables.
- 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 identify 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. It defines a vector graphic, which can scale up and down without losing any definition.

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 creates 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 the shape 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" . Note that the background  
    automatically scales to fit the size of the view.
```

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. It is not best practice to use hard-coded dimensions, but using weights with nested linear layouts to achieve the desired size is too much detail for this practical.

```
    android:layout_width="70dp"  
    android:layout_height="70dp"
```

4. Extract the dimension resource so you can access it in 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 large and repetitive, especially when you apply the same attributes to many similar elements. A style can specify common properties such as padding, font color, font size, and background color. 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 and apply them 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

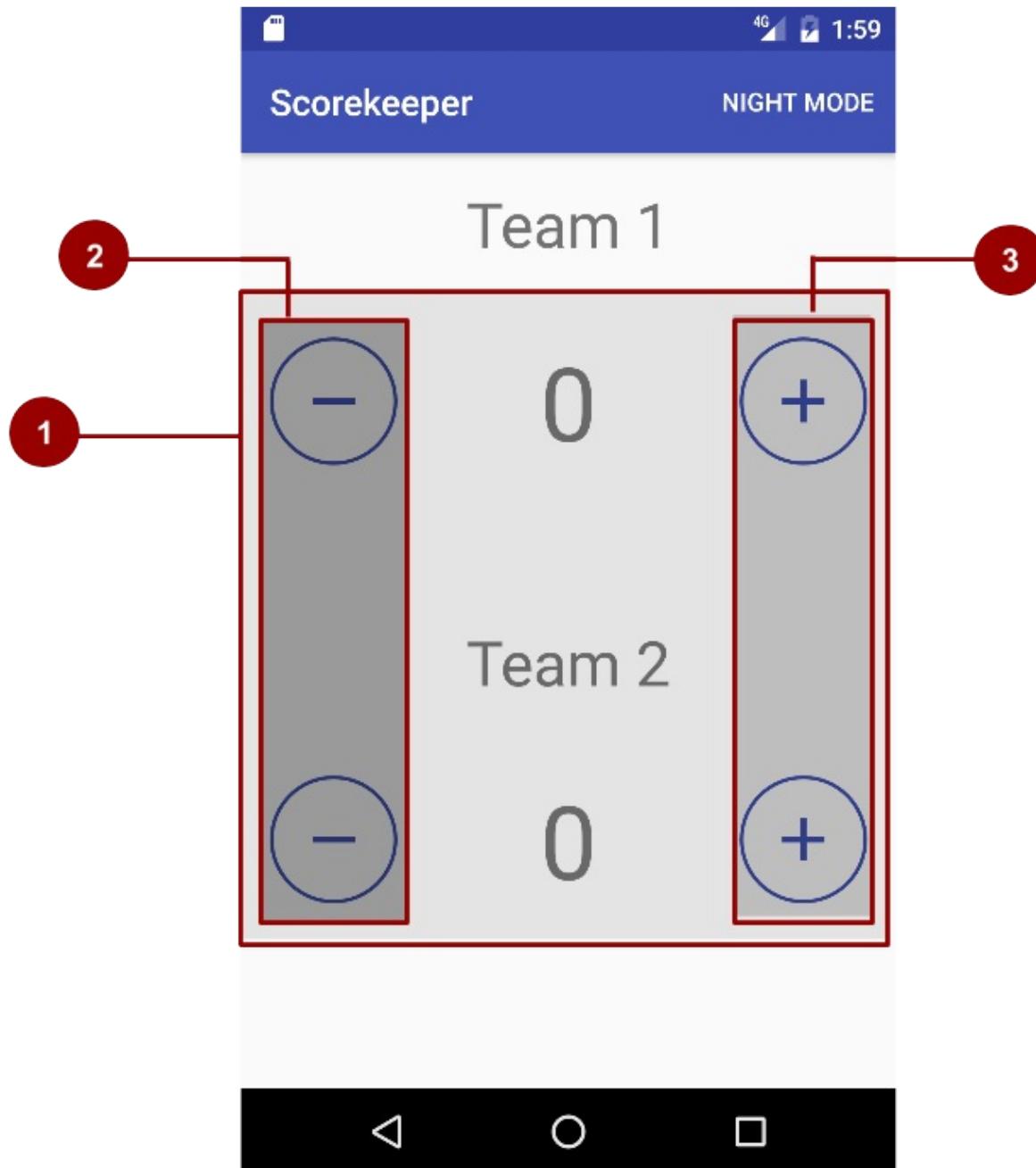
In Android, styles can inherit properties from other styles. You can declare a parent for your style using an optional "parent" parameter and has the following properties:

- Any style attributes defined by the parent style are automatically included in the child style.
- A style attribute defined in both the parent and child style uses the child style's definition (the child overrides the parent).
- A child style can include additional attributes that the parent does not define.

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 styles:

1. A score button style for all of the buttons, which includes the default properties of an `ImageButton` view and also the drawable background.
2. A minus button style for the decrement buttons, which inherits the attributes of the previous style and also includes the minus icon.
3. A plus button style for the increment buttons, again inheriting from the score button style and also includes the plus icon.

These styles are represented in the figure below.



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, which is how you specify your parent style using XML. The

name attribute, in this case "AppTheme", defines the name of the style. The parent attribute, in this case "Theme.AppCompat.Light.DarkActionBar", declares the parent style attributes which "AppTheme" inherits. In this case it is the Android 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. This allows for a consistent style throughout an entire activity or application (such as a consistent look and feel for the App Bar in every part of your application).

2. In between the `<resources>` tags, add a new 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>
```

Note: The contentDescription attribute is for visually impaired users. It acts as a label that certain accessibility devices use to read out loud to provide some context about the meaning of the UI element.

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 replace them with the appropriate style:

```
style="@style/MinusButtons"
style="@style/PlusButtons"
```

Note: The style attribute does not use the "android:" namespace, because it is part of the default XML attributes.

3.2 Create TextView styles

The team name and score display text views 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. Do not extract the attributes that are related to the layout.
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 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 is shorter.

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</i
tem>
    </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. You can 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. Change the style attribute of the team name TextViews to the newly created TeamText style in activity_main.xml.
4. Run your app. With only 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's 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 the users to use a low contrast version of your app that is easier on the eyes at night time, as well as make a few polishing touches to the User Interface.

4.1 Explore themes

1. In the Android manifest file, find the `<application>` tag and change the `android:theme` attribute to:

```
    android:theme="@style/Theme.AppCompat.Light.NoActionBar"
```

This is a 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`.

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. 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 by setting it 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"/>
```

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. Press **Ctrl - O** to open the Override Method menu in your main activity Java file, and select the **onCreateOptionsMenu** method located under the "android.app.Activity" category. 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, you can verify 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;
}
```

10. 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:

1. Override the `onSaveInstanceState()` method in `MainActivity` to preserve the values of the two score TextViews:

```
@Override  
protected void onSaveInstanceState(Bundle outState) {  
    //Save the scores  
    outState.putInt(STATE_SCORE_1, mScore1);  
    outState.putInt(STATE_SCORE_2, mScore2);  
    super.onSaveInstanceState(outState);  
}
```

2. In the `onCreate()` method of `MainActivity.java`, check if there is a `savedInstanceState`. If there is, restore the scores to the text views:

```
if (savedInstanceState != null) {  
    mScore1 = savedInstanceState.getInt(STATE_SCORE_1);  
    mScore2 = savedInstanceState.getInt(STATE_SCORE_2);  
  
    //Set the score text views  
    mScoreText1.setText(String.valueOf(mScore1));  
    mScoreText2.setText(String.valueOf(mScore2));  
}
```

That's it! Congratulations, you now have a styled Scorekeeper Application.

Solution code

Android Studio project: [Scorekeeper](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: 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

- A [ShapeDrawable](#) is a primitive geometric shape defined in an xml file by a number of attributes including color, shape, padding and more.
 - Drawables enhance the look of an application.
- 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.

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Drawables, Styles and Themes](#)

Learn more

Developer Documentation:

- [LinearLayout Guide](#)
- [Drawable Resource Guide](#)
- [Styles and Themes Guide](#)
- [DayNight Theme Guide](#)

Videos

- [Udacity - Themes and Styles](#)

5.2: Material Design: Lists, Cards and Colors

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](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

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 Views, efficiently use images, as well as employ design best practices to make your user's experience delightful.

What you should already KNOW

From the previous chapters you should be able to:

- Create and run apps in Android Studio.
- Create and edit elements using the Layout Editor, XML, and programmatically.
- Use 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 an 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 fix it up to meet the design guidelines to create a delightful user experience! Below are some screenshots of the app before and after the Material Design improvements.

The screenshot shows a mobile application interface with a blue header bar at the top. In the top right corner of the header, there are icons for signal strength (4G), battery level, and the time (19:44). The header contains the text "Material Me!" in white. Below the header, the main content area has a light gray background. It displays four sections, each consisting of a title in large bold black font followed by the word "News" in a smaller black font, and a text block below it stating "Here is some [Sport] news!". At the bottom of the screen is a black navigation bar with three white icons: a triangle pointing left, a circle, and a square.

Material Me!

Baseball

News

Here is some Baseball news!

Badminton

News

Here is some Badminton news!

Basketball

News

Here is some Basketball news!

Bowling

News

Here is some Rowling news!

4G 19:50

Material Me!

Bowling



News

Here is some Bowling news!

Cycling



News

Here is some Cycling news!



Task 1: Download the starter code

The complete starter app project for this practical is available at [MaterialMe-Starter](#). 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 and unzip the [MaterialMe-Starter file](#).
2. Open the app in Android Studio.
3. Build and run the app.

The app shows a list of sports names with some placeholder news text for each sport. The current layout and style of the app makes it nearly unusable: each row of data is not clearly separated and there is no imagery or color to engage the user.

1.2 Explore the App

1. Before making modifications to the app, explore its current structure. It contains the following elements:

- **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.

- **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.

- **MainActivity.java**

The MainActivity initializes the RecyclerView and adapter, and creates the data from resource files.

- **strings.xml**

This resource file contains all of the data for the application, including the titles and information for each sport.

- **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 is a good start for creating a dynamic and captivating user experience.

When presenting information that has mixed media (like images and text), the Material Design 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. CardView is a UI component found in the Android Support Libraries.

In this section, you will move each list item into a CardView and add an Image to make the app comply with Material guidelines.

2.1 Add the CardView

CardView is not included in the default Android SDK, so you must be add it 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"

Note: You will need to move the schema declaration

(`xmlns:android="http://schemas.android.com/apk/res/android"`) from the LinearLayout to the Cardview which is now the top level View of your layout file.

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. You have is a good opportunity to make this sports information app more exciting by adding banner images to every row!

Using images is resource intensive for your app: the Android framework has to load the entire image into memory at full resolution, even if the app only displays a small thumbnail of the image.

In this section you will learn how to use the Glide library to load large images efficiently, without draining your resources or even crashing your app due to 'Out of Memory' exceptions.

1. Download the [banner images zip file](#).
2. Copy these files into the res > drawable directory of your app.

Note: Copy the files using your file explorer, not Android Studio. Navigate to the directory where all your Android Projects are stored (It's called /AndroidStudioProjects), and go to MaterialMe/app/src/main/res/drawable and paste the images here.

You will need an array with the path to each image so that you can include it in the Sports java object. To do this:

3. Define an array that contains all of the paths to the drawables as items in your string.xml file. 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 correpsonds to the sport. To achieve that:

1. Add an integer member variable to the Sport object that will contain the drawable resource:

```
private final int imageResource;
```

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.title = title;
    this.info = info;
    this.imageResource = imageResource;
}
```

3. Create a getter for the resource integer:

```
public int getImageResource() {
    return imageResource;
}
```

2.4 Fix the `initializeData()` method

In `MainActivity.java`, the `initializeData()` method is now broken, because the constructor for the `Sport` object demands the image resource as the third parameter.

A convenient data structure to use would be a [TypedArray](#). A TypedArray allows you to store an array of other XML resources. By using a TypedArray, you will be able to obtain the image resources as well as the sports title and information by using indexing in the same loop.

1. In the `initializeData()` method, get the TypedArray of resource id's by calling `getResources().obtainTypedArray()`, passing in the name of the array of drawable resources you defined in your `strings.xml` file:

```
TypedArray sportsImageResources =
    getResources().obtainTypedArray(R.array.sports_images);
```

You can access an element at index `i` in the TypedArray by using the appropriate "get" method, depending on the type of resource in the array. In this specific case, it contains resource ID's, so you use the `getResourceId()` method.

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. Clean up the data in the typed array once you have created the Sport data ArrayList:

```
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.
2. 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.

3. Add the following attributes to the existing TextViews:

TextView id: title	Attribute	Value
	android:layout_alignBottom	"@+id/sportsImage"
	android:theme	"@style/ThemeOverlay.AppCompat.Dark"
TextView id: newsTitle	Attribute	Value
	android:layout_below	"@+id/sportsImage"
	android:textColor	"?android:textColorSecondary"
TextView id: subTitle	android:layout_below	"@+id/newsTitle"

Note: The question mark in the above textColor attribute ("?
android:textColorSecondary") means that the framework will apply the value from the currently applied theme. In this case, this attribute is inherited from the

"Theme.AppCompat.Light.DarkActionBar" theme, which defines it as a light gray color, often used for subheadings.

2.6 Load the images using Glide

After downloading the images and setting up the ImageView, the next 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. The Android framework has to load the image into memory each time at full resolution, no matter what the display size of the ImageView is.

There are a number of ways to reduce the memory consumption when loading images, but one of the easiest approaches is to use an Image Loading Library like [Glide](#), which you will do in this step. Glide uses background processing, as well some other complex processing, to reduce the memory requirements of loading images. It also includes some useful features like showing placeholder images while the desired images are loaded.

Note: You can learn more about reducing memory consumption in your app in the [Displaying Bitmaps guide](#).

1. Add the following dependency for Glide, in your app level build.gradle file:

```
compile 'com.github.bumptech.glide:glide:3.5.2'
```

2. Add a variable in the SportsAdapter class, ViewHolder class for the ImageView, 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. Glide 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.

4. Run the app, your list items should now have bold graphics that make the user experience dynamic and exciting!

Task 3: Make your CardView swipeable, movable and clickable

When users see cards in an app, they have expectations about the way the cards behave.

The [Material Design guidelines](#) say that:

- 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 behaviors 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. Some of the common use cases are already implemented in a set of methods in [ItemTouchHelper.SimpleCallback](#).

`ItemTouchHelper.SimpleCallback` lets you define which directions are supported for swiping and moving list items, and implement the swiping and moving behavior.

Do the following:

1. Create a new `ItemTouchHelper` object, in the `onCreate()` method of `MainActivity.java`. 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()`.

Note: If the required methods were not automatically added, click on the red light bulb and select **Implement methods**.

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. After creating the new ItemTouchHelper object in the Main Activity's `onCreate()` method, 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!

3.2 Implement drag and drop

You can also implement drag and drop functionality using the same SimpleCallback. The first argument of the SimpleCallback determines which directions the ItemTouchHelper supports for moving the objects around. 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 the `onMove()` method, 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 pass in the dataset, and the initial and final indexes:

```
Collections.swap(mSportsData, from, to);
```

4. Notify the adapter that the item was moved, passing in the old and new indexes:

```
mAdapter.notifyItemMoved(from, to);
```

- Run your app. You can now delete your list items by swiping them left or right, or reorder them using 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, because that is how you expect cards to behave. In this section, you will add a detail activity that will be launched when any list item is pressed. For this practical, the detail activity will contain the name and image of the list item you clicked, but will contain only 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 > Activity > Empty Activity**.
- Call it DetailActivity, accept all of the defaults.
- In the newly created layout file, remove the padding from the rootview RelativeLayout.
- Copy all of the TextView and ImageView 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()`).
- 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() :
```

```
Sport currentSport = mSportsData.get(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.getTitle());
detailIntent.putExtra("image_resource", currentSport.getImageResource());
```

13. Call `startActivity()` on the `mContext` variable, passing in the new Intent.

14. In `DetailActivity.java`, initialize the `ImageView` and `title TextView` in `onCreate()`:

```
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: Add the FAB and choose a Material Color Palette

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 is very likely to be used 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](#).

- 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:24.2.1'
```

- 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.
- In activity_main.xml, add a Floating Action Button view with the following parameters:

Attribute	Value
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_alignParentBottom	"true"
android:layout_alignParentRight	"true"
android:layout_margin	"16dp"
android:src	"@drawable/ic_reset"
android:onClick	resetSports

- Define the `resetSports()` method in MainActivity.java to simply call `initializeData()` to reset the data.
- 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 resets the data in this implementation. In order for the changes to be persistent (as in the case of reordering or removing data), you would have to implement `onSaveInstanceState()` or write the changes to a persistent source (like a database or Shared Preferences).

4.2 Choose a Material Palette

If you run your app you may notice that the FAB has a color that you didn't define anywhere. Also, the App bar (the bar that contains the title of your app) 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). The AppTheme style defines three colors 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.
4. An accent color. A color that contrasts well with the primary color. This is used for various highlights, but it is also the default color of the FAB. You can use the [Material Color Guide](#) to pick some colors to experiment with.
 5. 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. Modify the colorPrimary hex value in your colors.xml file to match the color you picked.
 6. Pick a darker shade of the same color to use as your primary dark color. Again, modify the colors.xml hex value for colorPrimaryDark to match.
 7. Pick an accent color for your FAB from the colors whose values start with an A, and whose color contrasts well with the primary color (like Orange A200). Change the colorAccent value in colors.xml to match.
 8. Add the android:tint attribute to the FAB and set it equal to #FFFFFF (white) in order to change the icon color to white.
 9. 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.

Solution code

Android Studio project: [MaterialMe](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

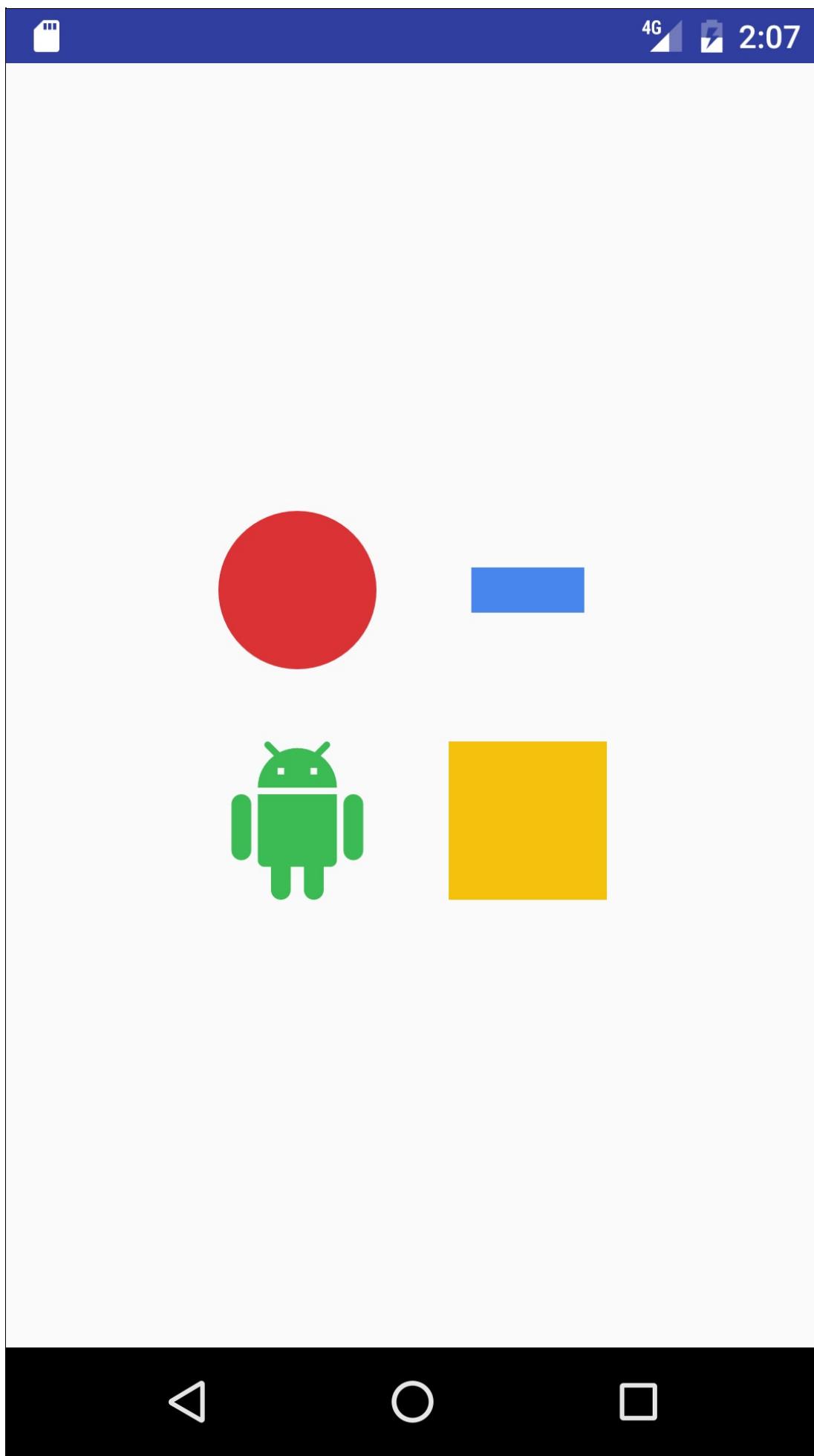
Challenge 1: This challenge consists of several small improvements to your application:

- Add real details to the Sport object and pass the details to the detail view.
- Implement a way to ensure that the state of the app is persistent across orientation changes.

Challenge 2: 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](#). Make each of these images 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.

Summary

- A CardView is a good layout when presenting information that has mixed media (such as images and text)
- CardView is a UI component found in the Android Support Libraries
- CardView was not designed just for text child Views.
- Loading images directly into ImageView is very memory intensive. All images are loaded at full-resolution.
- Use an image loading library, such as Glide, to efficiently load images into your app
- The Android SDK has a class called ItemTouchHelper which assists in obtaining tap, swipe or drag-and-drop information for your UI.
- A Floating Action Button (FAB) focuses the user on a particular action and "floats" about your UI.
- Material Design suggest 3 colors for your app: a primary color, a primary dark color and an accent color.
- The Material Design guidelines are a set of guiding principles that aim to create consistent, intuitive and playful applications.
- Material Design promotes the use of bold imagery and colors to enhance user experience.
- It also promotes consistent elements across platforms (such as CardView and the FAB).
- Material Design should be used for meaningful, intuitive motion like dismissable or rearrangeable cards.

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Material Design](#)

Learn more

- [Material Design Guidelines](#)
- [Material Palette Generator](#)
- [Cards and Lists Guide](#)
- [Floating Action Button Reference](#)

- Defining Custom Animations
- View Animation

5.3: Supporting Landscape, Multiple Screen Sizes and Localization

Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1: Support Landscape Orientation](#)
- [Task 2: Support Tablets](#)
- [Task 3: Localize your App](#)
- [Coding challenges](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

After using the Material Me! application you created in the last practical, you may notice that it is not optimized for use when the orientation for the device is rotated from portrait mode to landscape mode. Likewise, if you are testing on a tablet, the font-sizes are too small and the space is not used efficiently. The Android framework has a way to solve both of these issues. Resource qualifiers allow the Android Runtime to use alternate resource files (.xml) depending on the device configuration, such as, the orientation, the locale and other "qualifiers". For a full list of available qualifiers, visit the [Providing Resources](#) guide. In this practical you will optimize the use of space in the Material sports app so that your app works well in landscape mode, as well as on a tablet.

What you should already KNOW

From the previous chapters you should be able to:

- Locate and edit resource files.
- Extract resources.
- Instantiate a virtual phone or tablet using the emulator.

What you will LEARN

In this practical, you will learn to:

- Provide alternate resources for landscape mode.
- Provide alternate resources for tablets.
- Provide alternate resources for different locales.

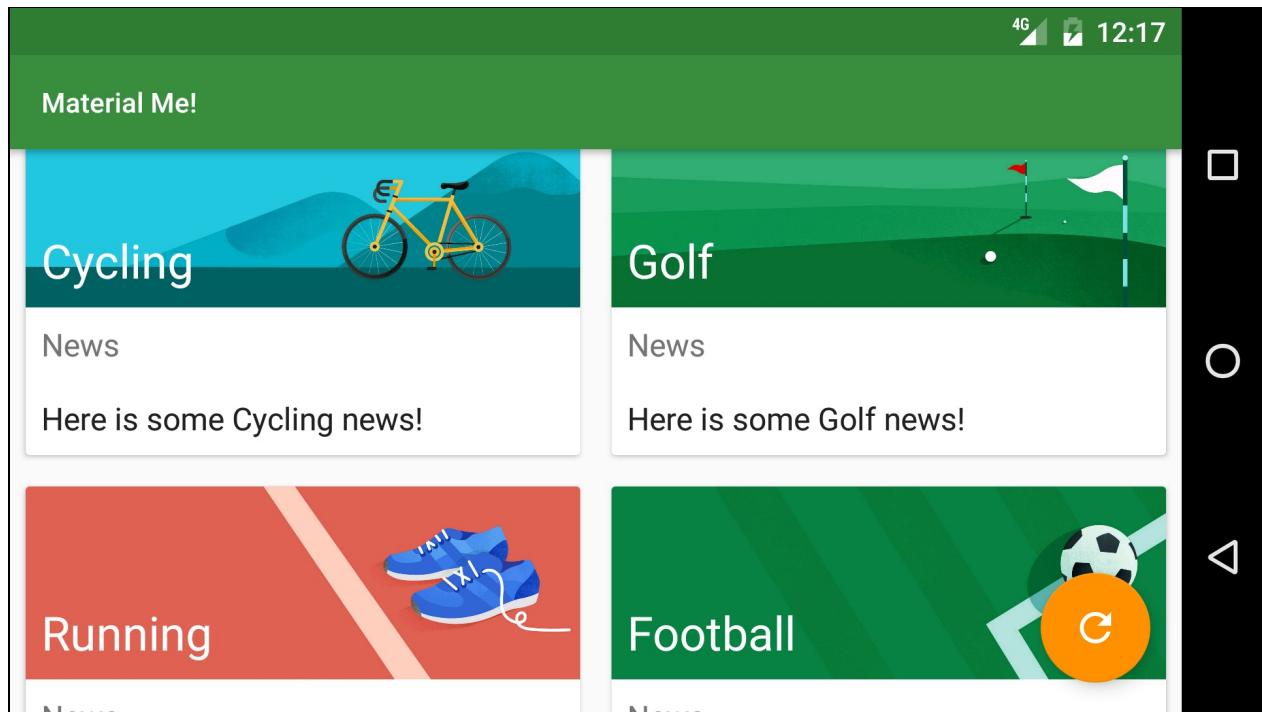
What you will DO

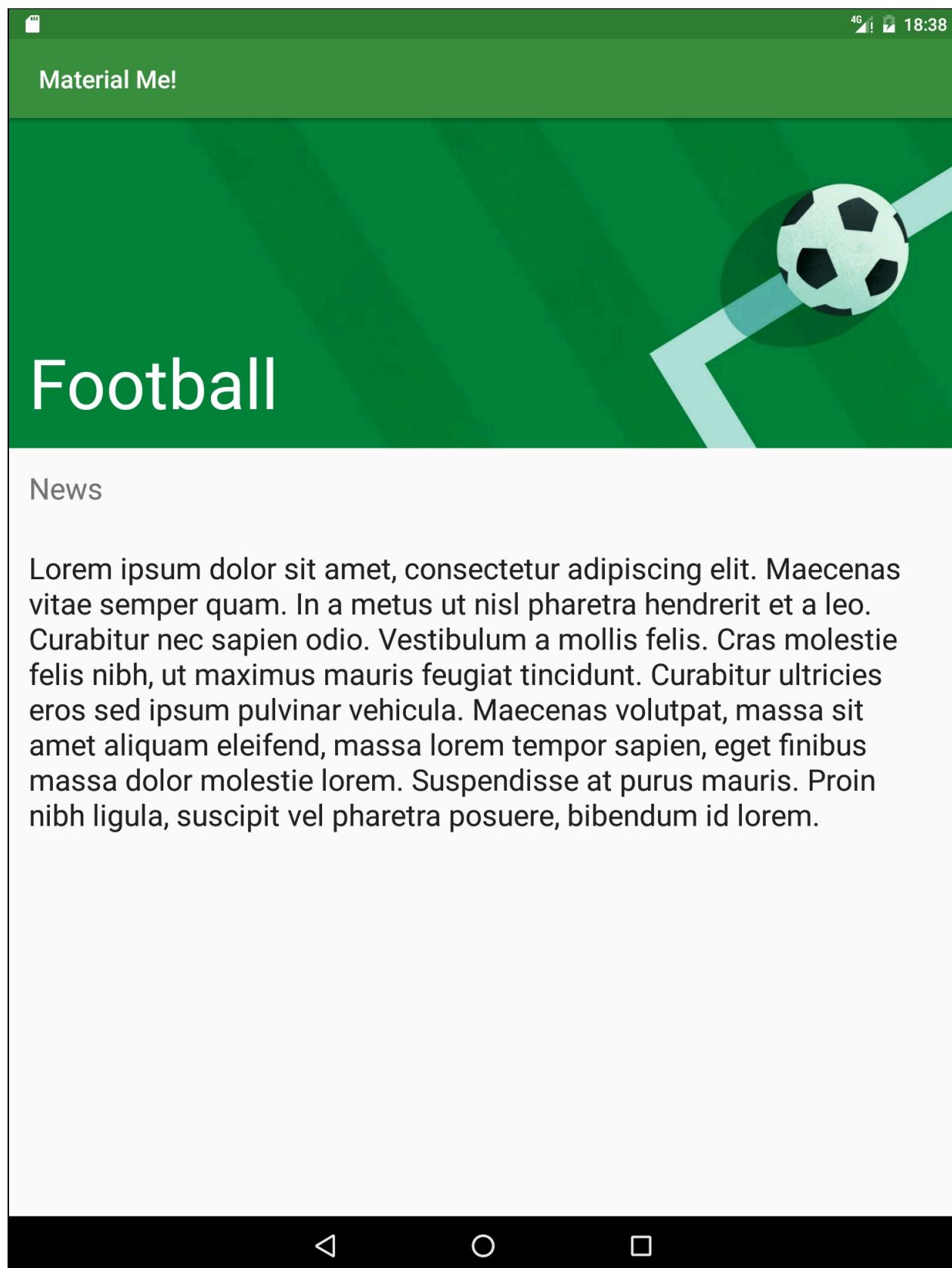
In this practical you will:

- Update the Material Me! application for better use of space in landscape mode.
- Add an alternative layout for tablets.
- Localize the content of your app.

App Overview

The improved Material Me! application will include improved layouts when used in landscape mode, on a tablet, and offer localized content for users outside of the US.



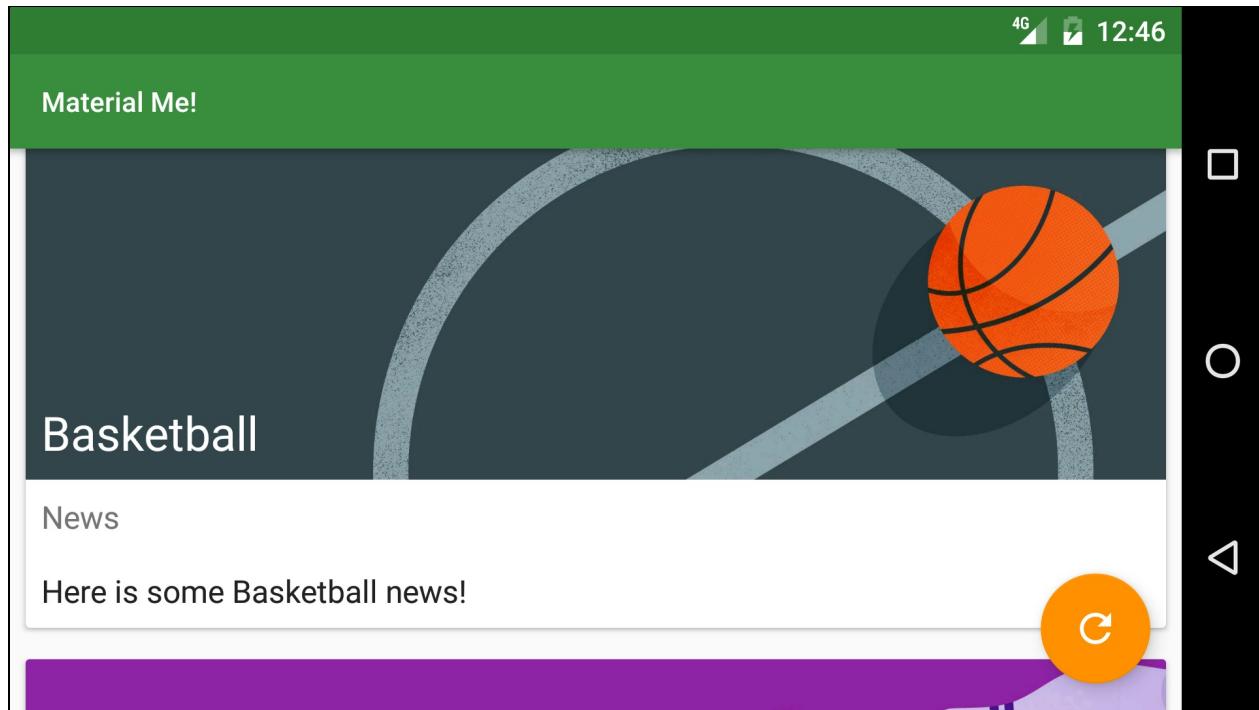


This practical builds on the "Material Me!" app from the previous practical.

1. Continue on your version of the "Material Me!" application, or [download it here](#).

Task 1: Support Landscape Orientation

You may recall that when the user changes the orientation of the device, the Android framework destroys and recreates the current activity. The new orientation often has different layout requirements than the original one. For example, the Material Me! application looks good in portrait mode, but does not make optimal use of the screen in landscape mode. With the larger width in landscape mode, the image in each list item overwhelms the text providing a poor user experience.



In this task, you will create an alternative resource file that will change the appearance of the app when it is used in landscape orientation.

1.1 Change to a GridLayoutManager

Layouts that contain list items often look unbalanced in landscape mode when the list items include full-width images. One good solution is to use a grid, instead of a linear list when displaying the CardViews in landscape mode. Recall that the items in a RecyclerView list are placed using a LayoutManager; until now, you have been using the [LinearLayoutManager](#) which lays out each item in a vertical or horizontal scrolling list. [GridLayoutManager](#) is another layout manager that displays items in a grid, rather than a list. When you create a GridLayoutManager, you supply two parameters: the application context, and an integer representing the number of columns. You can change the number of columns programmatically, which gives you flexibility in designing responsive layouts. In this case, the

number of columns integer should be 1 in portrait orientation (single column) and 2 when in landscape mode. Notice when the number of columns is 1, a GridLayoutManager behaves similar to a LinearLayoutManager.

1. Create a new resources file called integers.xml. Navigate to your resources directory, right-click on the values directory name and select **New > Values resource file**.
2. Name the file integers.xml and click **OK**.
3. Create an integer constant between the `<resources>` tags called "grid_column_count" and set it equal to 1:

```
<integer name="grid_column_count">1</integer>
```

4. Create another values resource file, again called integers.xml but with different characteristics.

Note the "Available qualifiers" option in the dialog for creating the resource file. These characteristics are called "resource qualifiers" and are used to label resource configurations for various situations.

5. Select **Orientation**, and press the **>>** symbol in the middle of the dialog to access this qualifier.
6. Change the Screen orientation selector to Landscape, and notice how the directory name "values-land" is automatically changed. This is the essence of resource qualifiers: the directory name tells Android when to use that specific layout file. In this case, that is when the phone is rotated to landscape mode.
7. Click **OK** to generate the new layout file.
8. Copy the integer constant you created into this new resource file, but change the value to 2.

You should now have two individual integers.xml files. In the "Android" project view in Android Studio, these should be grouped into an "integers.xml" folder, with each file inside labeled with the qualifier you selected ("land" in this case).

1.2 Modify MainActivity

1. In `onCreate()` in MainActivity, get the integer from the integers.xml resource file:

```
int gridColumnCount = getResources().getInteger(R.integer.grid_column_count);
```

The Android Runtime will take care of deciding which integers.xml file to use, depending on the state of the device.

2. Change the LinearLayoutManager to a GridLayoutManager, passing in the context and

the newly created integer:

```
mRecyclerView.setLayoutManager(new GridLayoutManager(this, gridColumnCount));
```

3. Run the app and rotate the device. The number of columns changes automatically with the orientation of the device.

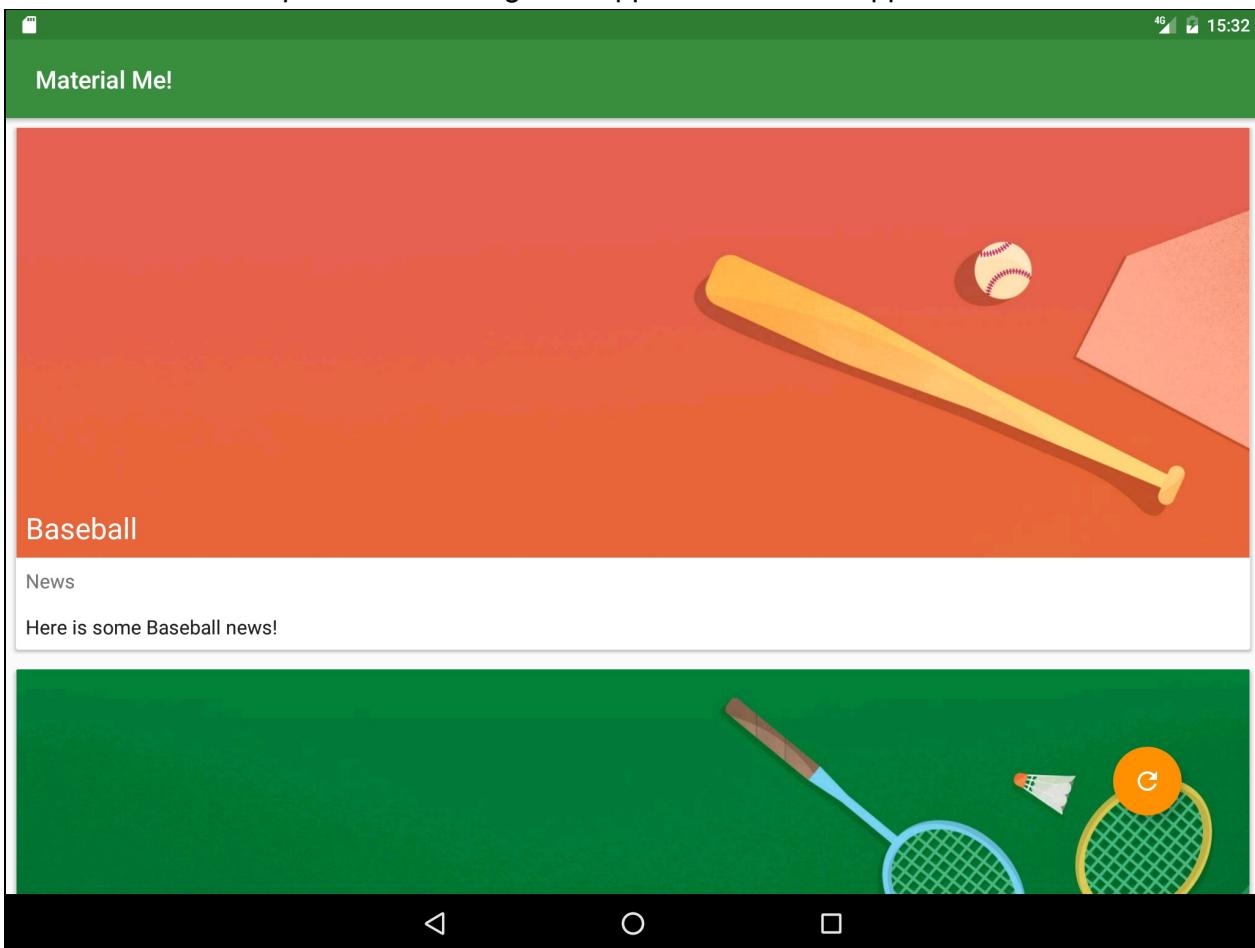
When using the application in landscape mode, you will notice that the swipe to dismiss functionality is no longer intuitive, since the items are now in a grid and not a list. You can use the `gridColumnCount` variable to disable the swipe action when there is more than one column:

```
int swipeDirs;  
if(gridColumnCount > 1){  
    swipeDirs = 0;  
} else {  
    swipeDirs = ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT;  
}  
ItemTouchHelper helper = new ItemTouchHelper(new ItemTouchHelper.SimpleCallback  
    (ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT | ItemTouchHelper.DOWN  
     | ItemTouchHelper.UP, swipeDirs)
```

Task 2 : Support Tablets

Although you have modified the app to look better in landscape mode, running it on a tablet with physically larger dimensions results in all the text appearing too small. Also when the device is in landscape orientation, the screen is not used efficiently; 3 columns would be

more appropriate for a tablet-sized screen in landscape mode. In this task, you will add additional resource qualifiers to change the appearance of the app when used on tablets.



2.1 Make the Layout Adapt to Tablets

In this step, you will create different resource qualifiers to maximize screen use for tablet sized devices, increasing the column count to 2 for portrait orientation and 3 for landscape orientation. The resource qualifier you need depends on your specific requirements. There are several qualifiers that you can use to select the correct conditions:

- "smallest width" - This qualifier is used most frequently to select for tablets. It is defined by the **smallest** width of the device (regardless of orientation), which removes the ambiguity when talking about "height" and "width" since some devices are traditionally held in landscape mode, and others in portrait. Anything with a smallest width of at least 600dp is considered a tablet.
- "available width" - The available width is the **effective** width of the device, regardless of the orientation. The available width changes when the device is rotated, since the effective height and width of the device are switched.
- "available height" - Same as "available width", except it uses the effective height instead of the effective width.

To start this task:

1. Create an integers.xml file which uses the "smallest width" qualifier with the value set to 600. Android uses this file whenever the app runs on a tablet.
 2. Copy the code from the integers.xml file with the landscape resource qualifier (it has a grid count of 2) and paste it in the new integers.xml file.
 3. Create a third integers.xml file that includes both the smallest screen width of 600dp qualifier, and the landscape orientation qualifier. Android uses this file when the app runs on a tablet in landscape mode.
- Note:** Android will look for the resource file with the most specific resource qualifier first, then move on to a more generic one. For example, if a value is defined in the integers.xml file with both the landscape and smallest width qualifier, it will override the value in the integers.xml file with just the landscape qualifier. For more information about resource qualifiers, visit the [Providing Resources Guide](#).
4. Change the grid_column_count variable to 3 in the landscape, tablet integers.xml file.
 5. Create a virtual tablet emulator. Run the app on a tablet emulator as well as a phone emulator and rotate both of the devices into landscape mode. With these resource qualifier files, the app uses the screen real estate much more effectively.

2.2 Update the tablet list item styles

At this point, your app changes the number of columns in a GridLayoutManager to fit the orientation of the device and maximize the use of the screen. However, all the TextViews that appeared correctly-sized on a phone's screen now appear too small for the larger screen of a tablet. To fix this, you will extract the TextAppearance styles from the layout resource files into the styles resource file. You will also create additional styles.xml files for tablets using resource qualifiers.

Note: You could also create alternative layout files with the proper resource qualifiers, and change the styles of the TextViews in those. However, this would require more code duplication, since most of the layout information is the same no matter what device you use, so you will only extract the attributes that will change.

Create the Styles

1. In the styles.xml file, create the following styles:

Name	Parent
SportsTitle	TextAppearance.AppCompat.Headline
SportsDetailText	TextAppearance.AppCompat.Subhead

Create a styles.xml file for tablets

Now you will create the file where you will define styles for tablets.

1. Create a new styles.xml resource file that uses the **Smallest Screen Width** qualifier with a value of 600.
2. Copy the "SportsTitle" and "SportsDetailText" styles from the original styles.xml file into the new, qualified styles.xml file.
3. Change the parent of the "SportsTitle" style to `"TextAppearance.AppCompat.Display1"`
4. The Android pre-defined `Display1` style uses the `textColorSecondary` value from the current theme (`ThemeOverlay.AppCompat.Dark`), which in this case is a light gray color. The light gray color does not show up well on the banner images in your app. To correct this add an `"android:textColor"` attribute to the "SportsTitle" style and set it to `"? android:textColorPrimary"`.

Note: The question mark tells Android Runtime to find the value in the theme applied to the View. In this example the theme is `ThemeOverlay.AppCompat.Dark` in which the `textColorPrimary` attribute is white.

5. Change the parent of "SportsDetailText" style to `"TextAppearance.AppCompat.Headline"`.

Update the style of the text views in list_item.xml

1. Back in the list_item.xml file, change the style attribute of the "title" TextView to `"@style/SportsDetailTitle"`
2. Change the style attribute of the "newsTitle" and "subTitle" TextViews to `"@style/SportsDetailText"`.
3. Run your app. Each list item now has a larger text size on the tablet.

2.3 Update the tablet sports detail styles

You have now fixed the display for the MainActivity, which lists all the Sports CardViews. The DetailActivity still has the same font sizes on tablets and phones.

1. Create the following style in each styles.xml file:

styles.xml (sw600dp)	
Name	Parent
SportsDetailTitle	TextAppearance.AppCompat.Display3
styles.xml (not qualified)	
Name	Parent
SportsDetailTitle	TextAppearance.AppCompat.Headline

2. Change the style of both the "newsTitleDetail" and "subTitleDetail" TextViews in the activity_detail.xml layout file to the "SportsDetailText" style you created in the previous step.
3. Run your app. All of the text is now larger on the tablet, which greatly improves the user

experience of your application.

Task 3: Localize your App

A "locale" represents a specific geographic, political or cultural region of the world. Resource qualifiers can be used to provide alternate resources based on the users' locale. Just as for orientation and screen width, Android provides the ability to include separate resource files for different locales. In this step, you will modify your strings.xml file to be a little more international.

3.1 Add a localized strings.xml file

You may have noticed that the sports information contained in this app is designed for users from the US. It uses the term "soccer" to represent a sport known as "football" everywhere else in the world. In order to make your app more internationalized, you can provide a locale specific strings.xml file to the US users which uses "soccer", while all other locales will use "football".

1. Create a new values resource file.
2. Call the file strings.xml and select **Locale** from the list of available qualifiers.
3. In the "Language:" section, select **en: English**.
4. In the "Specific Region Only:" section, select **US: United States**. This will create a specific values directory for the US locale, called "values-en-rUS".
5. Copy the entirety of the generic strings.xml file to the new locale specific strings.xml file.
6. In the generic strings.xml file, change the "Soccer" item in the sports_titles array to "Football", as well as changing the relevant item in "sports_info" array.
7. Run the app. Depending on the language setting on your device, you will see either "Soccer" or "Football".

Note: To change the locale setting on your device, go to the device settings, then choose **Language & input** and change the **Language** setting. If you pick **English (United States)** the app will have "Soccer" as the string, otherwise it will say "Football".

Solution code

Android Studio project: [MaterialMe-Resource](#)

Coding challenges

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge 1: It turns out that several countries other than the US use "soccer" instead of "football". Research these countries and add localized strings resources for them.

Challenge 2: Use the localization techniques you learned in Task 3 in combination with Google translate to translate all of the strings in your app into a different language.

Summary

- GridLayoutManager is a layout manager that handles 2-dimensional scrolling lists.
- You can dynamically change the number of columns in a GridLayoutManager.
- The Android Runtime uses alternative configuration files for various runtime environments such as device layout, screen dimensions, locale, country, keyboard, etc.
- Alternative resources are located in files named with resource qualifiers as part of their names.
- The format for these directories is `<resources_name>-<qualifier>`.
- Any file in your "res" directory can be qualified this way.
- Some common qualifiers are:
 - orientation: land, portrait
 - smallest width: sw600dp
 - locale: en-rGB, fr
 - screen density: ldpi, mpdi, xhdpi, xxhdpi, xxxhdpi
 - mobile country code: mcc310 (US), mcc208 (France)
 - and more!

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Providing Resources for Adaptive Layouts](#)

Learn more

Developer Documentation:

- [Supporting Multiple Screens](#)
- [Providing Resources](#)

6.1: 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](#)
- [Related concept](#)
- [Learn more](#)

When you, as a developer, test user interactions within your app, it helps to ensure that your app's users don't encounter unexpected results or have a poor experience when interacting with your app.

You can test the 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 hardware devices.

When you automate tests of UI interactions, you free yourself up 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. It is a good idea to get into the habit of creating user interface (UI) tests as you code 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 able to:

- Create and run apps in Android Studio.
- Create and edit UI elements using the Layout Editor, entering XML code directly, and accessing UI elements from your Java code.
- Add onClick functionality to a button.
- Build the TwoActivities app from a previous lesson.
- Build the PhoneNumberSpinner app from a previous lesson.
- Build the RecyclerView app from a previous lesson.

What you will LEARN

During this practical, you will learn to:

- Set up Espresso in your app project.
- Write an Espresso test that tests for user input and checks for the correct output.
- Write an Espresso test to find a spinner, click one of its items, and check for the correct output.
- Use the Record Espresso Test function in Android Studio.

What you will DO

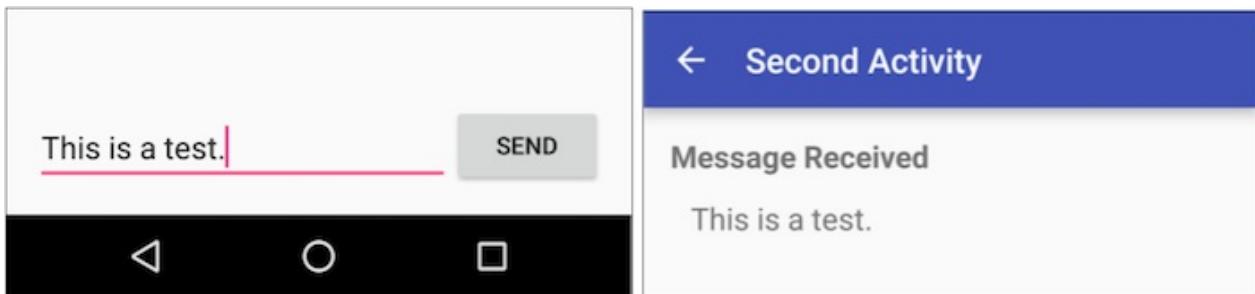
In this practical application you will:

- Modify a project to create Espresso tests.
- Test the app's text input and output.
- Test clicking a spinner item and check its output.
- Record an Espresso test of a RecyclerView.

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 and click the **Send** button, as shown on the left side of the figure below, and view that text in a second activity, as shown on the right side of the figure below.

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 and click the **Send** button, as shown on the left side of the figure below, and view that text in a second activity, as shown on the right side of the figure below.



Tip: For an introduction to testing Android apps, see [Test Your App](#).

Android Studio project: [TwoActivities](#)

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. If it is not, you will install it. You then will configure Espresso in the TwoActivities project 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 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 for the project

and widely-used unit testing framework for Java. Your test class using Espresso or UI Automator should be written as a JUnit 4 test class. If you do not have JUnit, you can get it at <http://junit.org/junit4/>.

Note: The most current JUnit revision is JUnit 5. However for the purposes of using Espresso or UI Automator, version 4.12 is recommended.

If you have created your project in a previous version of Android Studio, you may have to add the dependencies and instrumentation yourself. To add the dependencies yourself, follow these steps:

1. Open the TwoActivities project, or if you prefer, make a copy of the project first and then open the copy. See [Copy and rename a project](#) in the [Appendix](#) for instructions.
2. Open your **build.gradle (Module: app)** file.

Note: Do *not* make changes to the **build.gradle (Project: yourappname)** file.

3. Check if the following is included (along with other dependencies) in the `dependencies` section of the project's **build.gradle (Module: app)** file:

```
androidTestCompile  
    ('com.android.support.test.espresso:espresso-core:2.2.2', {  
        exclude group: 'com.android.support', module: 'support-annotations'  
    })  
testCompile 'junit:junit:4.12'
```

Note: If the file doesn't include the above dependency statements, enter them into the `dependencies` section.

4. Android Studio 2.2 also adds the following instrumentation statement to the end of the `defaultConfig` section of a new project:

```
testInstrumentationRunner  
    "android.support.test.runner.AndroidJUnitRunner"
```

Note: If the file doesn't include the above instrumentation statement, enter it at the end of the `defaultConfig` section.

[Instrumentation](#) is a set of control methods, or hooks, in the Android system. These hooks control an Android component independently of the component's 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.

5. If you modified the **build.gradle (Module: app)** file, click the **Sync Now** link in the notification about Gradle files in top right corner of the window.

5. If you modified the **build.gradle (Module: app)** file, 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

To let Android Studio communicate with your device, you must first turn on USB Debugging on your device, as described in an earlier chapter.

Android phones and tablets display animations when moving between apps and screens. The animations are attractive when using the device, but they slow down performance, and may also cause unexpected results or may lead your test to fail. So it's a good idea to turn off animations on your physical device. To turn off animations on your test device, tap on the Settings icon on your physical device. Look for **Developer Options**. Now look for the **Drawing** section. In this section, turn off the following options:

- Window animation scale
- Transition animation scale
- Animator duration scale

Tip: You should also keep in mind that instrumenting a system, such as in executing unit tests, can alter the timing of specific functions. For this reason, it is useful to keep unit testing and actual debugging separate. Unit testing uses an API based Espresso Framework with hooks for instrumentation. Debugging uses breakpoints and other methods in the actual coding statements within your app's code, as described in a previous lesson. </div>

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 Espresso tests are classes that are separate from your app's code. You can create as many tests as you need, in order to interact with the views in your UI that you want to test.

The Espresso test is like a robot that must be told what to do. It must *find* the view you want it to find on the screen, and it must *interact* with it, such as clicking the view, and checking the contents of the view. If it fails to do any of these things properly, or if the result is not what you expected, the test fails.

With Espresso, you create what is essentially a script of actions to take on each view and check against expected results. The key concepts are *locating* and then *interacting* with UI elements. These are the basic steps:

1. **Match a view:** Find a view.

Hamcrest (an anagram of "matchers") is a framework that assists writing software tests in Java. To create a test, you create a method within the test class that uses Hamcrest expressions.

Tip: For more information about the Hamcrest matchers, see [The Hamcrest Tutorial](#).

With Espresso you use the following types of Hamcrest expressions to help find views and interact with them:

- *ViewMatchers*: Hamcrest matcher expressions in the [ViewMatchers](#) class that lets you find a view in the current view hierarchy so that you can examine something or perform some action.
- *ViewActions*: Hamcrest action expressions in the [ViewActions](#) class that lets you perform an action on a view found by a ViewMatcher.
- *ViewAssertions*: Hamcrest assertion expressions in the [ViewAssertions](#) class that lets you assert or check the state of a view found by a ViewMatcher.

The following shows how all three expressions work together:

1. Use a ViewMatcher to find a view:

```
onView(withId(R.id.my_view))
```

2. Use a ViewAction to perform an action:

```
.perform(click())
```

3. Use a ViewAssertion to check if the result of the action matches an assertion:

```
.check(matches(isDisplayed()));
```

The following shows how the above expressions are used together in a statement:

```
onView(withId(R.id.my_view))
    .perform(click())
    .check(matches(isDisplayed()));
```

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 `ExampleInstrumentedTest`.

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 `ExampleInstrumentedTest`.
2. To make the test more understandable and describe what it does, rename the class from `ExampleInstrumentedTest` to the following:

```
public class ActivityInputOutputTest
```

3. Change the class definition to the following:

```
@RunWith(AndroidJUnit4.class)
public class ActivityInputOutputTest {
    @Rule
    public ActivityTestRule mActivityRule = new ActivityTestRule<>(
        MainActivity.class);
}
```

The class definition now includes several annotations:

- `@RunWith` : To create an instrumented JUnit 4 test class, add the `@RunWith(AndroidJUnit4.class)` annotation at the beginning of your test class definition.
- `@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`.

In the above statement, `ActivityTestRule` may turn red at first, but then Android Studio adds the following import statement automatically:

```
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.

But what happens if no text is entered? Will the `Second` activity still appear?

The `ActivityInputOutputTest` class of tests will show that the views appear regardless of whether text is entered. Follow these steps to add your tests to `ActivityInputOutputTest`:

1. Add an `activityLaunch()` method to `ActivityInputOutputTest` 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` may turn red at first, but then Android Studio adds import statements for `onView`, `withID`, and `click`.
4. Add another ViewMatcher expression to the `activityLaunch()` method 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, the `check()` method may turn red at first, but then Android Studio adds an `import` statement for it.
6. Add similar statements to test whether clicking the `button_second` button in the `Second`

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, the `check()` method may turn red at first, but then Android Studio adds an `import` statement for it.
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 `activityLaunch()` method you just created in the `ActivityInputOutputTest` class. 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:

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. 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.")));
```

3. 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:

Android Studio Project: [TwoActivitiesEspressoTest](#)

See `ActivityInputOutputTest.java`.

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$AssertionFailedWithCauseException: 'with text: is "This is a failing test."' doesn't match the selected view.  
Expected: with text: is "This is a failing test."  
Got: "AppCompatTextView{id=2131427417, res-name=text_message ...}
```

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$AssertionFailedWithCauseE  
rror: 'with text: is "This is a failing test."' doesn't match the selected view.  
Expected: with text: is "This is a failing test."  
Got: "AppCompatTextView{id=2131427417, res-name=text_message ...
```

Other fatal error messages appear after the above, due to the cascading effect of a failure leading to other failures. You can safely ignore them and fix the test itself.

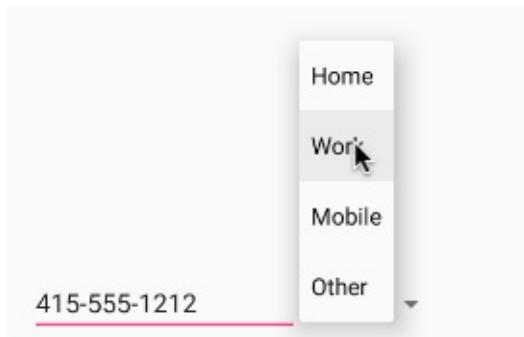
Task 3: Test the display of spinner selections

The Espresso `onView()` method finds a view that you can test. This method will find a view in the current view hierarchy. But you need to be careful—in an AdapterView such as a spinner, the view is typically dynamically populated with child views at runtime. This means there is a possibility the view that you want to test may not be in the view hierarchy at that time.

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 locating and performing actions on 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.



2. Configure Espresso in your project as described previously.
3. Expand **com.example.android.phonenumberspinner (androidTest)**, and open **ExampleInstrumentedTest**.
4. Rename `ExampleInstrumentedTest` to `SpinnerSelectionTest` in the class definition, and add the following:

```
@RunWith(AndroidJUnit4.class)
public class SpinnerSelectionTest {
    @Rule
    public ActivityTestRule mActivityRule = new ActivityTestRule<>(
        MainActivity.class);
}
```

5. Create the `iterateSpinnerItems()` method as `public void` returning `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()`.

2. Assign the length of the array to `size`, 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

1. Add an `onView()` statement within the `for` loop to find the spinner and click on it:

```

int size = myArray.length;
for (int i=0; i<size; i++) {
}

```

3.3 Locate spinner items and click on them

1. 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());

```

A user must click the spinner itself in order click any item in the spinner, so your test must also click the spinner first before clicking the item.

2. 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());

```

The above statement matches if the object is a specific item in the spinner, as specified by the `myArray[i]` array element.

If `onData` appears in red, click the word, and then click the red light bulb icon that appears in the left margin. Choose the following in the pop-up menu:

Static import method 'android.support.test.espresso.Espresso.onData'

If `is` appears in red, click the word, and then click the red light bulb icon that appears in the left margin. Choose the following in the pop-up menu:

Static import method...> Matchers.is (org.hamcrest)

3. Add two more `onView()` statements to the for loop:

```

// Find the Submit 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]`.

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:

Android Studio project: [PhoneNumberSpinnerEspressoTest](#)

See `SpinnerSelectionTest.java`.

Task 4: Record a test of a RecyclerView

You learned how to create a [RecyclerView](#) in a previous chapter. 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.

Recording Espresso tests, rather than coding the tests by hand, ensures that your app gets UI test coverage on areas that might take too much time or be too difficult to code by hand.

Solution code:

Android Studio project: [RecyclerView](#)

4.1 Open and run the app

1. Open the RecyclerView project, or if you prefer, make a copy of the project first and then open the copy. See [Copy and rename a project](#) in the [Appendix](#) for instructions.
2. Configure Espresso in your project as described previously.

Android Studio project: RecyclerView

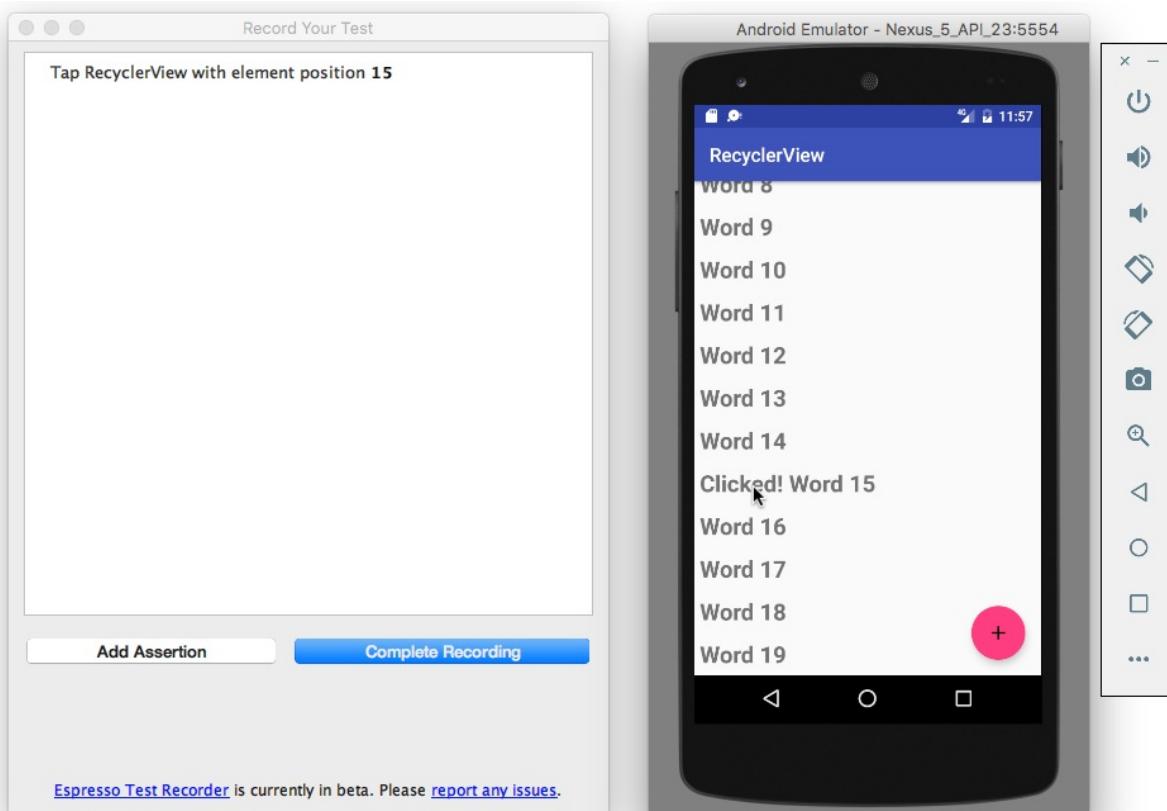
4.1 Open and run the app

1. Open the RecyclerView project, or if you prefer, make a copy of the project first and then open the copy. See [Copy and rename a project](#) in the [Appendix](#) for instructions.
2. Configure Espresso in your project as described previously.
3. Run the app to ensure that it runs properly. You can use the emulator or an Android device.

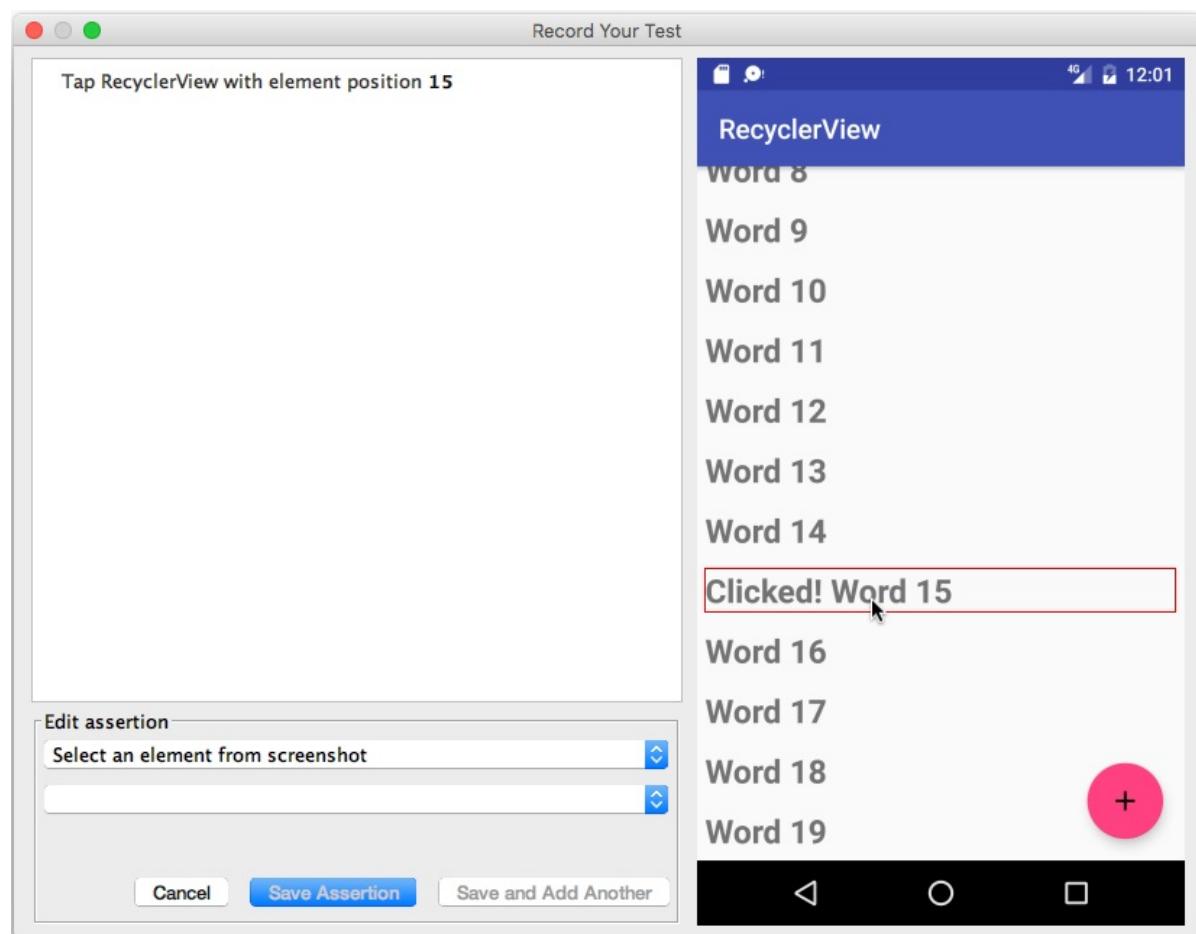
The app lets you scroll a list of words. When you click on a word such as **Word 15** the word in the list changes to "Clicked! Word 15".

4.2 Record the test

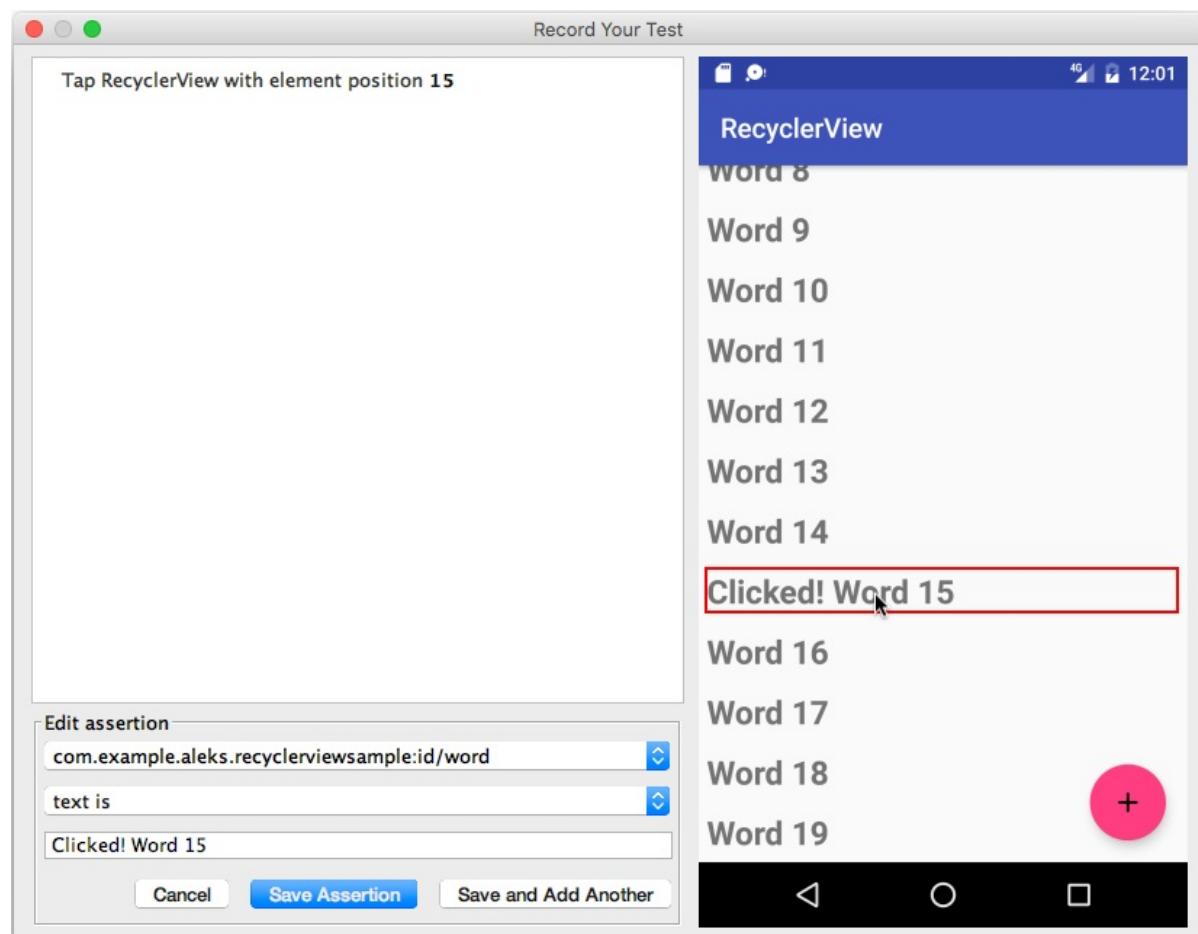
1. Choose **Run > Record Espresso Test**, select your deployment target (an emulator or a device), and click **OK**.
2. Scroll the word list in the app on the emulator or device, and tap on **Word 15**. The Record Your Test window shows the action that was recorded ("Tap RecyclerView with element position 15").



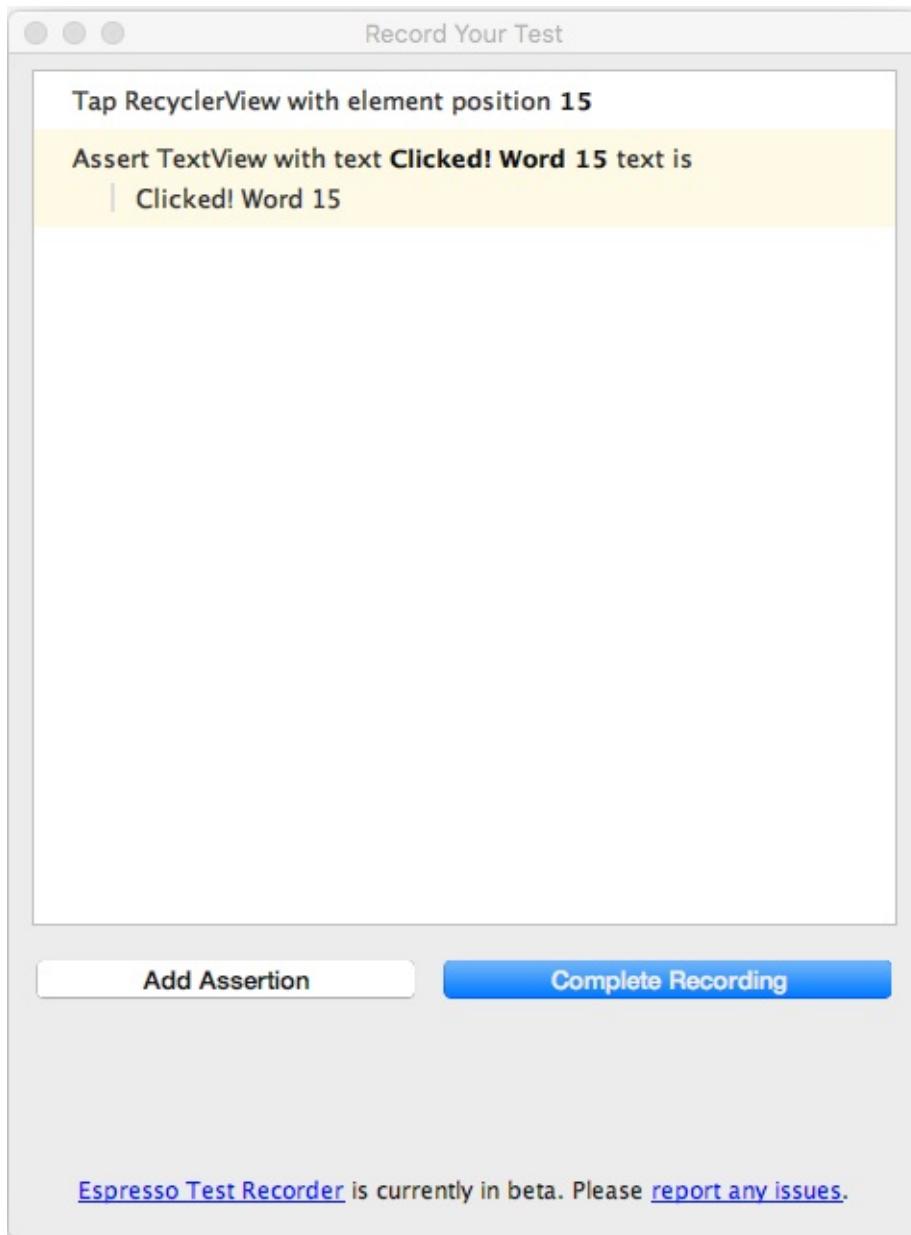
3. Click **Add Assertion** in the Record Your Test window. A screenshot of the app's UI appears in a pane on the right side of the window. Select **Clicked! Word 15** in the



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. Android Studio may display a request to add more 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 in the RecyclerViewTest.java file:

```

@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"),
                childAtPosition(
                    childAtPosition(
                       (withId(R.id.recyclerview),
                            11),
                        0),
                isDisplayed())));
        textView.check(matches(withText("Clicked! Word 15")));
    }

    private static Matcher<View> childAtPosition(
        final Matcher<View> parentMatcher, final int position) {

        return new TypeSafeMatcher<View>() {
            @Override
            public void describeTo(Description description) {
                description.appendText("Child at position " + position + " in parent ");
                parentMatcher.describeTo(description);
            }

            @Override
            public boolean matchesSafely(View view) {
                ViewParent parent = view.getParent();
                return parent instanceof ViewGroup && parentMatcher.matches(parent)
                    && view.equals(((ViewGroup) parent).getChildAt(position));
            }
        };
    }
}

```

The test uses a RecyclerView object of the [ViewInteraction](#) class, which is the primary interface for performing actions or assertions on views, providing both `check()` and `perform()` methods. Examine the test code to see how it works:

- **Perform:** The code below uses the `perform()` method and 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()));
```

- **Check whether it matches the assertion:** The code below checks to see if the clicked item matches the assertion that it should be `"Clicked! Word 15"` :

```
ViewInteraction textView = onView(
    allOf(withId(R.id.word), withText("Clicked! Word 15"),
        childAtPosition(
            childAtPosition(
               (withId(R.id.recyclerview),
                11),
            0),
        isDisplayed())));
textView.check(matches(withText("Clicked! Word 15")));
```

The code above uses a method called `childAtPosition()` , which is defined as a custom `Matcher` :

```
private static Matcher<View> childAtPosition(
    final Matcher<View> parentMatcher, final int position) {
    // TypeSafeMatcher() returned
    ...
}
```

- **Implement a custom matcher:** The custom matcher extends the abstract `TypeSafeMatcher` class and requires that you implement the following:
 - The `matchesSafely()` method, shown below, to define how to check for a view in a `RecyclerView`.
 - The `describeTo()` method, shown below, to define how Espresso describes the output's matcher in the Run pane at the bottom of Android Studio if a failure occurs.

```
...
// TypeSafeMatcher() returned
return new TypeSafeMatcher<View>() {
    @Override
    public void describeTo(Description description) {
        description.appendText("Child at position "
            + position + " in parent ");
        parentMatcher.describeTo(description);
    }

    @Override
    public boolean matchesSafely(View view) {
        ViewParent parent = view.getParent();
        return parent instanceof ViewGroup &&
            parentMatcher.matches(parent)
            && view.equals(((ViewGroup)
                parent).getChildAt(position));
    }
};
}
```

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.

Solution code

Android Studio project: [RecyclerViewEspressoTest](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: Write an Espresso test 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**.

Summary

In this practical, you learned how to do the following:

- Set 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.
- Test 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.
- Test a spinner and its selections:
 - 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.
- Record 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.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Testing Your User Interface](#)

Learn more

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: Create an AsyncTask

Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [Task 1: Setup the SimpleAsyncTask project](#)
- [Task 2: Create the AsyncTask subclass](#)
- [Task 3: Implement the Final Steps](#)
- [Coding challenge](#)
- [Summary](#)
- [Related Concept](#)
- [Learn more](#)

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 of code 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 needs to perform 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 if all of the code executes serially on a single thread. When the app is performing resources intensive work, the app does not respond to the user or draw on the screen because it is waiting for that work to be done. This can yield poor performance, which negatively affects the user experience. Users may get frustrated and uninstall your Android app if the performance of the app is slow.

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, thereby allowing the UI thread to remain very responsive. Since the separate thread is not synchronized with the calling thread, it is called an asynchronous thread. An `AsyncTask` also contains a callback that allows you to display the results of the computation 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

You should be able to:

- Create an Activity.
- Add a TextView to the layout for the activity.
- Programmatically get the id for the TextView and set its content.
- Use Button views and their onClick functionality.

What you will LEARN

During this practical, you will learn to:

- Add an AsyncTask to your app in order to run a task in the background, off of the UI thread.
- Identify and understand the benefits and drawbacks of using AsyncTask for background tasks.

What you will DO

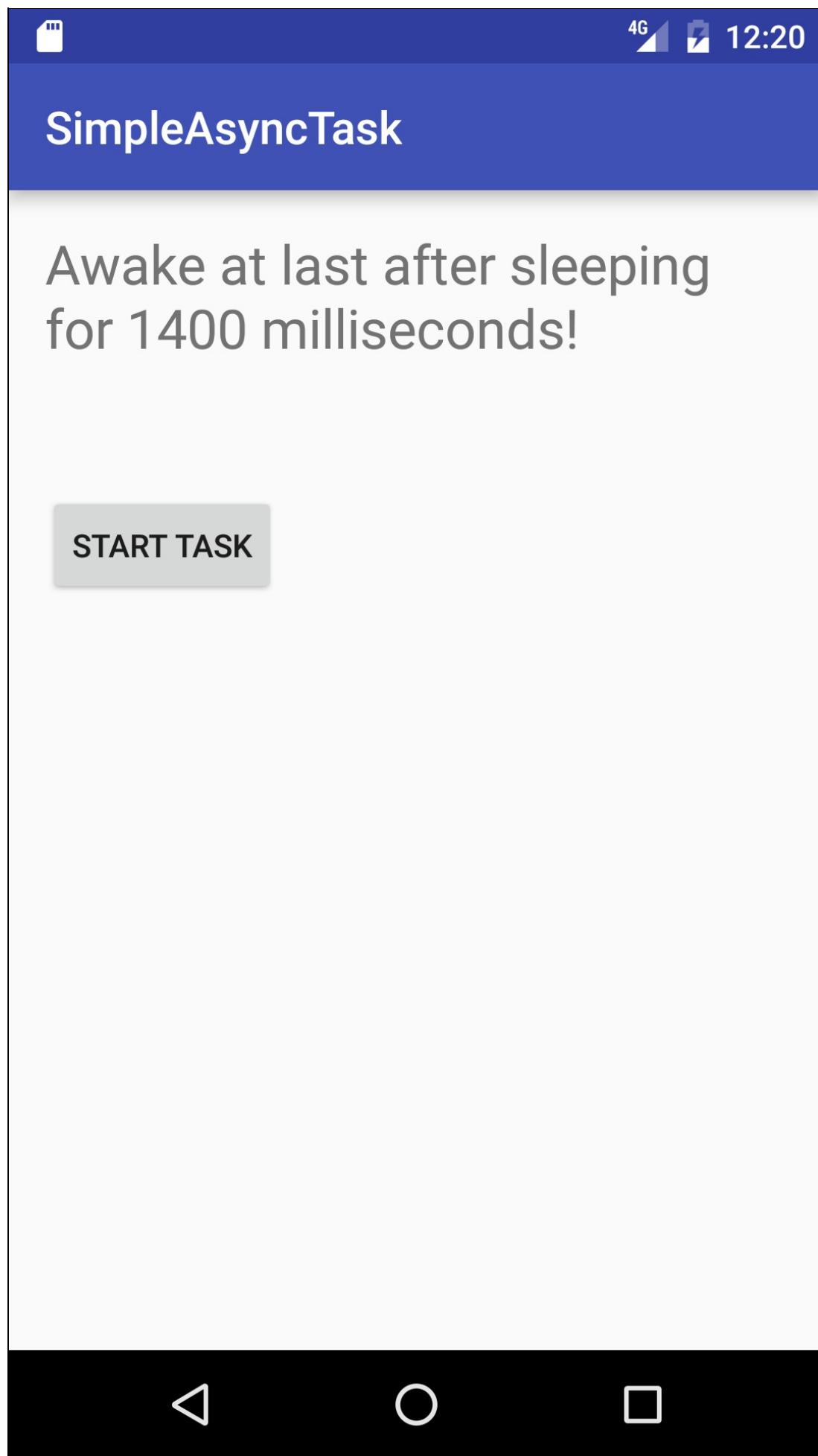
During this practical, you will:

- 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. Setup the SimpleAsyncTask Project

The SimpleAsyncTask UI is straightforward. It contains a button that launches the AsyncTask, and a TextView that displays the status of the application.

1.1 Create the layout

1. Create a new project called **SimpleAsyncTask** using the Empty Activity template (accept the defaults for the other options).
2. Change the root view `RelativeLayout` to a `LinearLayout`.
3. Add the following essential UI elements to the layout for the MainActivity:

View	Attributes	Values
LinearLayout	<code>android:orientation</code>	vertical
TextView	<code>android:text</code> <code>android:id</code>	I am ready to start work! <code>@+id/textView1</code>
Button	<code>android:text</code> <code>android:onClick</code>	Start Task <code>startTask</code>

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 (using `wrap_content` ensures that this is the case).

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"/>

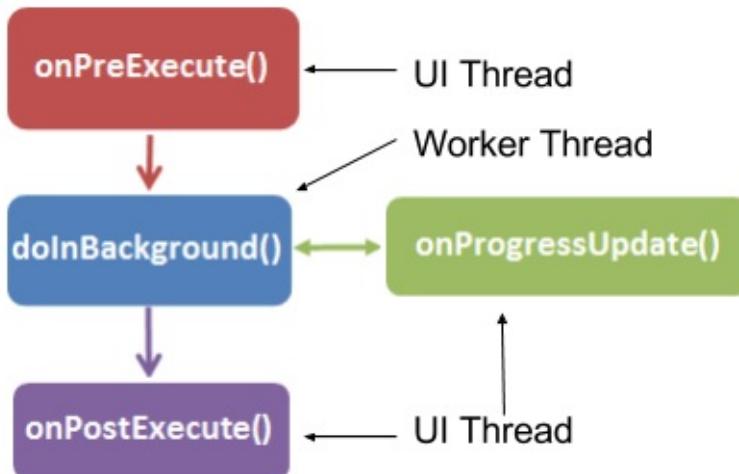
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/start_task"
        android:id="@+id/button"
        android:layout_marginTop="56dp"
        android:onClick="startTask" />
</LinearLayout>
```

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 so that you can beat the current Go champion.

An AsyncTask has the following methods for performing work off of the main thread:

- `onPreExecute()` : This method runs on the UI thread, and is used for setting up your task (like showing a progress bar).
- `doInBackground()` : This is where you implement the code to execute the work that is to be performed on the separate thread.
- `onProgressUpdate()` : This is 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, this is used for updating the results to the UI once the AsyncTask has finished loading.



Note: A background or worker thread is any thread which is not the main or UI thread.

When you create an `AsyncTask`, you may need to give it information about the work which it is to perform, whether and how to report its progress, and in what form to return the result.

In this exercise you will use an `AsyncTask` subclass to define work that will run in a different thread than the UI thread, which will avoid any 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 then 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 one 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
- Put the current thread to sleep. (Use `Thread.sleep()`) in a try/catch block.
- 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 (UI) thread. You cannot call `mTextView.setText()` in `doInBackground()`, because that method is executed on a separate thread.

Task 3. Implement the 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 you didn't call `sleep()` as the 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 the `TextView` in the UI.
3. Add code to the `startTask()` method to create an instance of `SimpleAsyncTask`, passing the `TextView` `mTextView` to the constructor.
4. Call `execute()` on that `SimpleAsyncTask` instance.

Note: The `execute()` method is where you pass in the parameters (separated by commas) that are then 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 Implement onSaveInstanceState()

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. If the background task completes before you can rotate the phone, try again. Alternatively, you can update the code and make it sleep for a longer time period.

Note: 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 several things going on here:

- o When you rotate the device, the system restarts the app, calling `onDestroy()` and then `onCreate()`, which restarts the activity lifecycle. Since the AsyncTasks are no longer connected to the lifecycle of your app, and cannot reconnect to the activity.
- o The AsyncTasks will continue running to completion in the background, consuming system resources, but never showing the results in the UI, which gets reset in

`onCreate()`. It will never be able to update the `TextView` that was passed to it, since that particular `TextView` has also been destroyed. Eventually, the system runs out of resources, and will fail.

- Even without the `AsyncTask`, the rotation of the device 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.

In order to prevent the `TextView` from resetting to the initial string, you need to save its state. You've already learned how to maintain the state of views in a previous practical, using the `SavedInstanceState` class.

You will now implement `onSaveInstanceState()` to preserve the content of your `TextView` when the activity is spontaneously destroyed.

Note: Not all uses of `AsyncTask` require you to handle the state of the views on rotation. This app uses a `TextView` to display the results of the app, so preserving the state is useful. In other cases, such as uploading a file, you may not need any persistent information in the UI, so retaining the state is not critical.

- Override the `onSaveInstanceState()` method in `MainActivity` to preserve the text inside the `TextView` when the activity is destroyed:

```
outState.putString(TEXT_STATE, mTextView.getText().toString());
```

- 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 in 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());
    }
}

```

```
}
```

Solution code

Android Studio project: [SimpleAsyncTask](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: 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

- 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.
- An AsyncTask is an abstract Java class that moves intensive processing onto a separate thread.
 - AsyncTask must be subclassed to be used.
 - 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 your AsyncTask 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.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [AsyncTask and AsyncTaskLoader](#)

Learn more

Android Developer Documentation

- [Processes and threads](#)
- [Processing Bitmaps off the UI thread](#) using [AsyncTask](#)
- [AsyncTask](#)

Other 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: Connect to the Internet with AsyncTask and AsyncTaskLoader

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. Implement UI best practices](#)
- [Task 4. Migrate to AsyncTaskLoader](#)
- [Coding challenges](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

In this practical you will use an `AsyncTask` to start a background task which gets 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. Then you will reimplement the same background task using `AsyncTaskLoader`, which will be more efficient in updating your UI, handling performance issues, and improving the overall UX.

What you should already KNOW

From the previous practicals you should be able to:

- Create an activity.
- Add a `TextView` to the layout for the activity.
- Implement `onClick` functionality to a button in your layout.
- Implement an `AsyncTask` and display the result in your UI.
- Pass information between activities as extras.

What you will LEARN

In this practical, you will learn to:

- Use the Google API Explorer to investigate Google APIs and to view JSON responses to http requests.
- Use the Books API as an example API retrieving 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](#).
- Parse the JSON results returned from your API query.
- Implement an AsyncTaskLoader that preserves data upon configuration changes.
- Update your UI using the Loader callbacks.

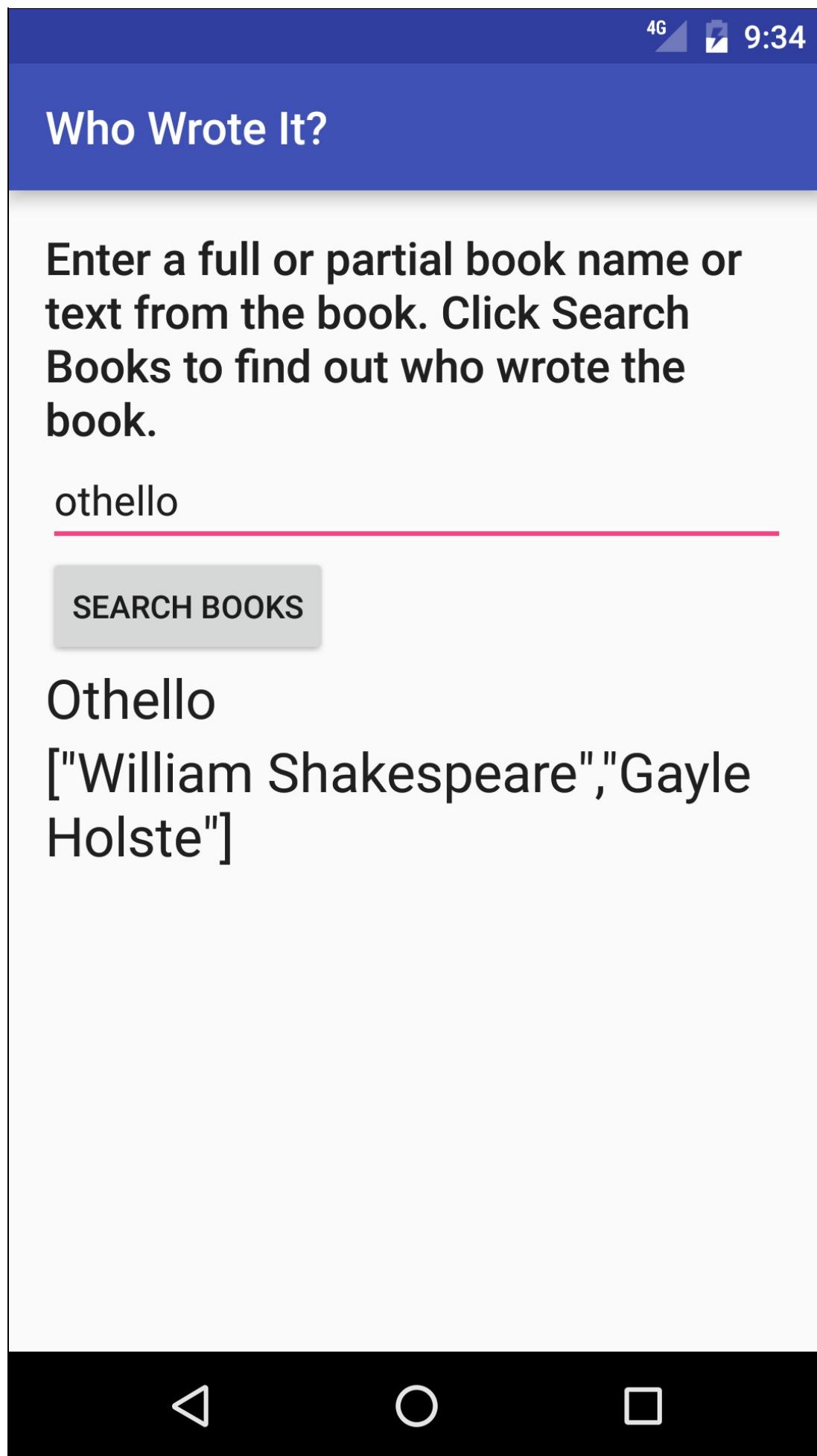
What you will DO

In this practical, you will:

- Use the Google API Explorer to learn about the simple search feature of the Books API.
- Create the "Who Wrote It?" application that queries the Books API using a worker thread and displays the result in the UI.
- Modify the "Who Wrote it?" app to use an AsyncTaskLoader instead of an AsyncTask.

App Overview

You will build an app that contains an EditText field and a Button. The user enters the name of the book in the EditText field and clicks the button. The button executes an AsyncTask which queries the Google Book Search API to find the author and title of the book the user is looking for. The results are retrieved and displayed in a TextView field below the button. Once the app is working, you will then modify the app to use [AsyncTaskLoader](#) instead of the [AsyncTask](#) class.



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 title. 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](#) (found at <https://developers.google.com/apis-explorer/>).
2. Click **Books API**.
3. Find (**Ctrl-F** or **Cmd-F**) **books.volumes.list** and click that function name. You should see a webpage that lists the various parameters of the Books API function that performs the book searches.
4. In the `q` field enter a book name, or partial book name. The `q` parameter is the only required field.
5. 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`, which returns only books in print; and `magazines` which returns only magazines.
6. 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.
7. 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. 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. For security reasons, when accessing a public API, you usually need to get an API key and include it in your Request. However, this specific API does not require a key, so you can leave out that portion of the Request URI in your app.

1.2 Analyze the Books API Response

Towards the bottom of the page you can see the Response to the query. The response uses the [JSON format](#), which is a common format for API query responses. In the API Explorer web page, the JSON code is nicely formatted so that it is human readable. In your application, the JSON response will be returned from the API service as a single string, and you will need to parse that string to extract the information you need.

1. In the Response section, 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.

Task 2. Create the "Who Wrote It?" App

Now that you are familiar with the Books API method that you will be using, it's 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 root view—the view that contains all the other views inside a layout XML file. Make sure the LinearLayout uses `android:orientation="vertical"` :

View	Attributes	Values
TextView	android:layout_width android:layout_height android:id android:text android:textAppearance	wrap_content wrap_content @+id/instructions @string/instructions @style/TextAppearance.AppCompat.Title
EditText	android:layout_width android:layout_height android:id android:inputType android:hint	match_parent wrap_content @+id/bookInput text @string/input_hint
Button	android:layout_width android:layout_height android:id android:text android:onClick	wrap_content wrap_content @+id/searchButton @string/button_text searchBooks
TextView	android:layout_width android:layout_height android:id android:textAppearance	wrap_content wrap_content @+id/titleText @style/TextAppearance.AppCompat.Headline
TextView	android:layout_width android:layout_height android:id android:textAppearance	wrap_content wrap_content @+id/authorText @style/TextAppearance.AppCompat.Headline

3. In the strings.xml file, add these string resources:

```
<string name="instructions">Enter a book name, or part of a  
book name, or just some 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 with all `onClick` methods, this one takes 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 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 use 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 that it's a bad idea.

Use an `AsyncTask` to make network connections:

1. Create a new Java class called `FetchBook` in `app/java` that extends `AsyncTask`. An `AsyncTask` requires three arguments:
 - The input parameters.
 - The progress indicator.
 - 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, `doInBackground()`, by placing your cursor on the red underlined text, pressing **Alt + Enter** (**Opt + Enter** on a Mac) and selecting **Implement methods**. Choose `doInBackground()` and click **OK**. Make sure the parameters and return types are the correct type (It takes a `String` array and returns a `String`).
 - i. Click the **Code** menu and choose **Override methods** (or press **Ctrl + O**). Select

the `onPostExecute()` method. The `onPostExecute()` method takes a `String` as a parameter and returns `void`.

- 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 initialize them in a constructor. You will use this constructor in your `MainActivity` to pass along the `TextViews` to your `AsyncTask`.

Solution code for `FetchBook`:

```
public class FetchBook extends AsyncTask<String,Void,String>{
    private TextView mTitleText;
    private TextView mAuthorText;

    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 Create the `NetworkUtils` class and 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. You will write the code for connecting to the internet in a helper class called `NetworkUtils`.

- Create a new Java class called `NetworkUtils` by clicking **File > New > Java Class** and only filling in the "Name" field.
- Create a unique `LOG_TAG` variable to use throughout `NetworkUtils` class for logging:

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

- Create a new static method called `getBookInfo()` that takes a `String` as a parameter (which will be the search term) and returns a `String` (the JSON String response from the API you examined earlier).

```
static String getBookInfo(String queryString){}
```

4. Create the following two local variables in `getBookInfo()` that will be needed later to help connect and read the incoming data.

```
HttpURLConnection urlConnection = null;  
BufferedReader reader = null;
```

5. Create another local variable at the end of `getBookInfo()` to contain the raw response from the query and return it:

```
String bookJSONString = null;  
return bookJSONString;
```

If you remember the request from the Books API webpage, you will notice that all the requests begin with the same URI. To specify the type of resource, you append query parameters to the base URI. It is common practice to separate all of these query parameters into constants, and combine them using a `Uri.Builder` so 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, you will limit the number and type of results returned to increase the query speed. To restrict the query, you will only look for books that are printed.

6. Create the following member constants in the `NetworkUtils` class:

```
private static final String BOOK_BASE_URL = "https://www.googleapis.com/books/v1/  
volumes?"; // Base URI for the Books API  
private static final String QUERY_PARAM = "q"; // Parameter for the search string  
private static final String MAX_RESULTS = "maxResults"; // Parameter that limits s  
earch results  
private static final String PRINT_TYPE = "printType"; // Parameter to filter by  
print type
```

7. Create a skeleton try/catch/finally block in `getBookInfo()`. This is where you will make your HTTP request. The code to build the URI and issue the query 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;
}

```

- Build up your request URI in the try block:

```

//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();

```

- Convert your URI to a URL:

```
URL requestURL = new URL(builtURI.toString());
```

2.5 Make the Request

It is fairly common to make an API request via the internet. Since you will probably use this functionality again, you may want to create a utility class with this functionality or develop a useful subclass for your own convenience. This API request 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.

- In the try block of the `getBookInfo()` method, open the URL connection and make the request:

```

urlConnection = (HttpURLConnection) requestURL.openConnection();
urlConnection.setRequestMethod("GET");
urlConnection.connect();

```

- 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 its input parameter for a null string and let the user know the connection failed. This error handling strategy is simplistic, as the user has no idea why the connection failed. A better solution for a production application 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. You are now done with the `getBookInfo()` method.

```
Log.d(LOG_TAG, bookJSONString);
```

6. In your `AsyncTask` `doInBackground()` method, call the `getBookInfo()` method, passing in the search term which you obtained from the `params` argument passed in by the system (it is the first value in the `params` array). Return the result of this method in the `doInBackground()` method:

```
return NetworkUtils.getBookInfo(params[0]);
```

7. 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);
```

8. Run your app. Execute a search. Your app will crash. Look at your Logs to see 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 your apps do not have connectivity by default. You must add permissions manually in the form of a `<uses-permission>` ; tag in the `AndroidManifest.xml`.

2.6 Add the Internet permissions

1. Open the `AndroidManifest.xml` file.
2. All permissions of your app need to be put in the `AndroidManifest.xml` file outside of the `<application>` ; tag. You should be sure to follow the order in which tags are defined in `AndroidManifest.xml`.
3. Add the following xml tags outside of the `<application>` tag:

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

4. 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 aids in the parsing and handling of JSON type data. This process, as well as updating the UI, will happen in the `onPostExecute()` method.

There is chance that the `doInBackground()` method might not return the expected JSON string. For example, the try catch might fail and throw an exception, the network might time out or other unhandled errors might occur. In those cases, the Java JSON methods will fail to parse the data and will throw exceptions. This is why you have to do the parsing in the try block, and the catch block must handle the case where incorrect or incomplete data is returned.

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);
JSONArray itemsArray = jsonObject.getJSONArray("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 be 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 {
        title = volumeInfo.getString("title");
        authors = volumeInfo.getString("authors");
    } catch (Exception e){
        e.printStackTrace();
    }

    //If both a title and author exist, update the TextViews and return
    if (title != null && authors != null){
        mTitleText.setText(title);
        mAuthorText.setText(authors);
        return;
    }
}
```

4. If there are no results which meet the criteria of having both a valid author and a title, and the loop has stopped, 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 `authors` 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);
        JSONArray itemsArray = jsonObject.getJSONArray("items");
        for(int i = 0; i<itemsArray.length(); i++){
            JSONObject book = itemsArray.getJSONObject(i);
            String title=null;
            String authors=null;
            JSONObject volumeInfo = book.getJSONObject("volumeInfo");

            try {
                title = volumeInfo.getString("title");
                authors = volumeInfo.getString("authors");
            } catch (Exception e){
                e.printStackTrace();
            }

            if (title != null && authors != null){
                mTitleText.setText(title);
                mAuthorText.setText(authors);
                return;
            }
        }

        mTitleText.setText("No Results Found");
        mAuthorText.setText("");
    } catch (Exception e){
        mTitleText.setText("No Results Found");
        mAuthorText.setText("");
        e.printStackTrace();
    }
}
```

Task 3. Implement UI Best Practices

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 actually 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. To use this solution, you can:

1. Add the following code to the `searchBooks()` method to hide the keyboard when the button is pressed:

```
InputMethodManager inputManager = (InputMethodManager)
    getSystemService(Context.INPUT_METHOD_SERVICE);
inputManager.hideSoftInputFromWindowgetCurrentFocus().getWindowToken(),
    InputMethodManager.HIDE_NOT_ALWAYS);
```

2. Add a line of code beneath 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 Manage the network state and the empty search field case

Whenever your application uses the network, it needs to handle the possibility that a network connection is unavailable. Before attempting to connect to the network in your AsyncTask or AsyncTaskLoader, your app should check the state of the network connection.

1. Modify your `searchBooks()` method to check both the network connection and if there is any text in the search field before executing the FetchBook task.
2. Update the UI in the case that there is no internet connection or no text in the search field. Display the cause of the error in the TextView.

Solution code:

```
public void searchBooks(View, view) {  
  
    String queryString = mBookInput.getText().toString();  
  
    InputMethodManager inputManager = (InputMethodManager)  
        getSystemService(Context.INPUT_METHOD_SERVICE);  
    inputManager.hideSoftInputFromWindowgetCurrentFocus().getWindowToken(),  
        InputMethodManager.HIDE_NOT_ALWAYS);  
  
    ConnectivityManager connMgr = (ConnectivityManager)  
        getSystemService(Context.CONNECTIVITY_SERVICE);  
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();  
  
    if (networkInfo != null && networkInfo.isConnected() && queryString.length()!=0) {  
        new FetchBook(mTitleText, mAuthorText).execute(queryString);  
        mAuthorText.setText("");  
        mTitleText.setText(R.string.loading);  
    }  
  
    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.");  
        }  
    }  
}
```

Task 4. Migrate to AsyncTaskLoader

When using an `AsyncTask`, it cannot update the UI if a configuration change occurs while the background task is running. To address this situation, the Android SDK provides a set of classes called loaders designed specifically for loading data into the UI 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 created it. The Loader framework does the work for you by reassociating the loader with the appropriate Activity when the device changes its configuration. This means that if you rotate the device while the task is still running, the results will be displayed correctly in the Activity once the data is returned.

In this task you will use a specific loader called an `AsyncTaskLoader`. An `AsyncTaskLoader` is an abstract subclass of `Loader` and uses an `AsyncTask` to efficiently load data in the background.

Note: When you used an `AsyncTask`, you implemented 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.

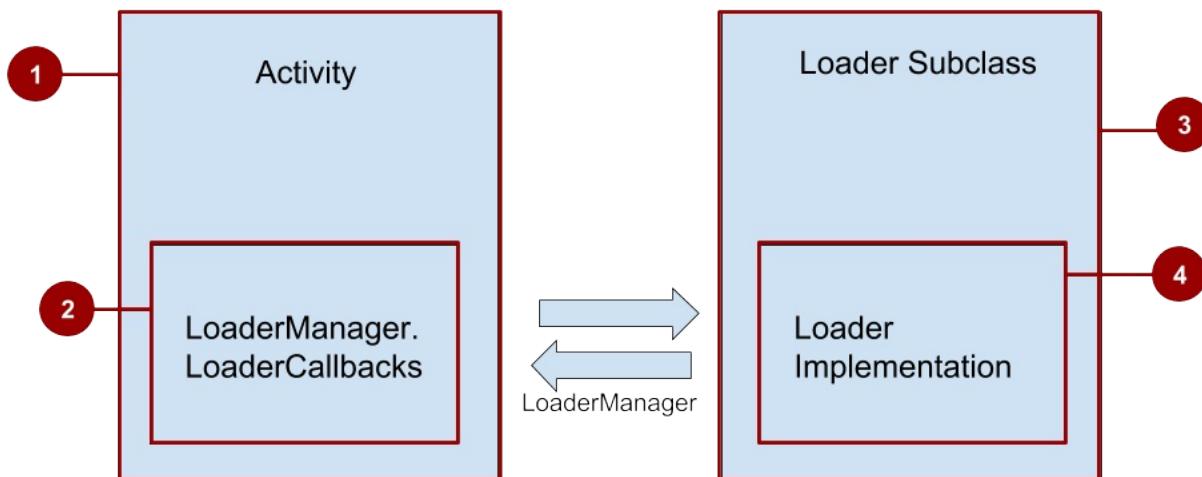
So why should you use an `AsyncTask` if an `AsyncTaskLoader` is so much more useful? The answer is that it depends on the situation. If the background task is likely to finish before any configuration changes occur, and it is not crucial that it updates the UI, an `AsyncTask` may be sufficient. The Loader framework actually uses an `AsyncTask` behind the scenes to work its magic.

A good rule of thumb is to use an `AsyncTaskLoader` instead of an `AsyncTask` if the user might rotate the screen while the job is running, or when it's critical to update the UI when the job finishes.

In this exercise you will learn how to use a `AsyncTaskLoader` instead of an `AsyncTask` to run your Books API query. You will learn more about the uses of other loaders in a later lesson.

Implementing a Loader requires the following components:

- A class that extends a Loader class (in this case, `AsyncTaskLoader`).
- An Activity that implements the `LoaderManager.LoaderCallbacks` class.
- An instance of the `LoaderManager`.



1. The Activity.
2. The `LoaderManager.LoaderCallbacks`.
3. The Loader subclass.
4. The Loader Implementation.

The LoaderManager automatically moves the loader through its lifecycle depending on the state of the data and the Activity. For example, the LoaderManager calls `onStartLoading()` when the loader is initialized and destroys the loader when the Activity is destroyed.

The LoaderManager.LoaderCallbacks are a set of methods in the Activity that are called by the LoaderManager when loader is being created, when the data has finished loading, and when the loader is reset. The LoaderCallbacks can take the results of the task and pass them back to the Activity's UI.

The Loader subclass contains the details of loading the data, usually overriding at least `onStartLoading()`. It can also contain additional features such as observing the data source for changes and caching data locally.

Your Loader subclass implements Loader lifecycle callback methods such as `onStartLoading()`, `onStopLoading()` and `onReset()`. The loader subclass also contains the `forceLoad()` method which initiates the loading of the data. This method is not called automatically when the loader is started because some setup is usually required before a load is performed. The simplest implementation would call `forceLoad()` in `onStartLoading()` which results in a load every time the LoaderManager starts the loader.

4.1 Create an AsyncTaskLoader

1. Copy the [WhoWroteIt project](#), in order to preserve the results of the previous practical. Rename the copied project **WhoWroteItLoader**.
2. Create a new class in your Java directory called BookLoader.
3. Have your BookLoader class extend AsyncTaskLoader with parameterized type `<String>`.
4. Make sure you import the loader from the v4 Support Library.
5. Implement the required method (`loadInBackground()`). Notice the similarity between this method and the initial `doInBackground()` method from AsyncTask.
6. Create the constructor for your new class. In Android Studio, it's likely the class declaration will still be underlined in red because your constructor does not match the superclass implementation. 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.

Define `onStartLoading()`

1. Press **Ctrl + O** to open the Override methods menu, and select **onStartLoading**. This method is called by the system when you start your loader.
2. The loader will not actually start loading the data until you call the `forceLoad()` method. Inside the `onStartLoading()` method stub, call `forceLoad()` to start the `loadInBackground()` method once the Loader is created.

Define loadInBackground()

1. Create a member variable `mQueryString` that will hold the query `String`, and modify the constructor to take a `String` as an argument and assign it to the `mQueryString` variable.
2. In the `loadInBackground()` method, call the `getBookInfo()` method passing in `mQueryString`, and return the result to download the information from the Books API:

```
@Override
public String loadInBackground() {
    return NetworkUtils.getBookInfo(mQueryString);
}
```

4.2 Modify MainActivity

You must now implement the [Loader Callbacks](#) in your `MainActivity` to handle the results of the `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 the required methods: `onCreateLoader()`, `onLoadFinished()`, `onLoaderReset()`. 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.

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 that all imports are from the Android Support Library.

Loaders use the `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:

- `initLoader()` : This method creates a new loader if one does not exist already, and passes in the arguments Bundle. If a loader exists, the calling Activity is re-associated with it without updating the Bundle.
- `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.

Both of these methods are defined in the LoaderManager, which manages all the Loader instances used in an Activity (or Fragment). Each Activity has exactly one LoaderManager instance that is responsible for the lifecycle of the Loaders that it manages.

Currently, the FetchBook AsyncTask is triggered when the user presses the button. You'll want to start your loader with a new Bundle each time the button is pressed. To do this, you need to edit the onClick method for the button.

1. In the `searchBooks()` method, which is the onClick method for the button, replace the call to execute the FetchBook task with a call to `restartLoader()`, 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:

- A loader `id` (useful if you implement more than one loader in your activity).
- An arguments `Bundle` (this is where any data needed by the loader goes).
- 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.

2. Examine the Override methods in the LoaderCallbacks class. These methods are:

- `onCreateLoader()` : Called when you instantiate your Loader.
- `onLoadFinished()` : Called when the loader's task finishes. This is where you add the code to update your UI with the results.
- `onLoaderReset()` : Cleans up any remaining resources.

You will only be defining the first two methods, since your current data model is a simple string that does not need extra care when the loader is reset.

Implement `onCreateLoader()`

1. In `onCreateLoader()`, return an instance of the BookLoader class, passing in the `queryString` obtained from the arguments Bundle:

```
return new BookLoader(this, args.getString("queryString"));
```

Implement `onLoadFinished()`

1. Update `onLoadFinished()` to process your result, which is 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`.
- 2. Run your app.

You should have the same functionality as before, but now in a Loader! One thing still does not work. When the device 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.

- 3. 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: If the loader exists, initialize it. You only want to reassociate 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.

- 4. Run your app again and rotate the device. The `LoaderManager` now preserves your data across device configurations!
- 5. Remove the `FetchBook` class as it is no longer used.

Solution code

Android Studio project: [WhoWroteItLoader](#)

Coding challenges

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge 1: Explore 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.

Challenge 2: 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 a

`maxResults` amount of entries.

Summary

- Tasks that connect to the network, or require extra time processing, 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 numerous 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 a JSON string.
- Use `getText()` to retrieve text from an EditText view. 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 class that allows you to run tasks in the background, asynchronously, instead of on the UI thread.
 - An AsyncTask can be started via `execute()`.
 - An AsyncTask will not be able to update the UI if the Activity it is controlling terminates (such as in a configuration change on the device).
 - An AsyncTask must be subclassed to be used. The subclass will override at least one method `dolnBackground(Params)`, and most often will override a second one `onPostExecute(Result)` as well.
- Whenever an AsyncTask is executed, it goes through the following 4 steps:
 1. `onPreExecute()`. Invoked on the UI thread before the task is executed. This step is normally used to set up the task.
 2. `dolnBackground(Params)`. Invoked on the background thread immediately after `onPreExecute()` finishes executing. This step is used to perform background computation that can take a long time.
 3. `onProgressUpdate(Progress)`. Invoked on the UI thread after you a call in `dolnBackground` to `publishProgress(Progress)`.
 4. `onPostExecute(Result)`. Invoked on the UI thread after the background computation has finished. The result of the background computation gets passed into this method as a parameter.
- `AsyncTaskLoader` is the Loader equivalent of an AsyncTask. It provides a method,

`loadInBackground()`, that runs on a separate thread and whose results are automatically delivered to the UI thread (to the `onLoadFinished()` `LoaderManager` callback).

- You must configure network permissions 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.
- 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 terminated before the task finishes (such as by device rotation).
 - An AsyncTaskLoader is a Loader that uses an AsyncTask helper class behind the scenes to do work in the background off the main thread.
 - Loaders are managed by a LoaderManager; one or more Loaders can be assigned and managed by a single LoaderManager.
 - The LoaderManager allows you to associate a newly created Activity with a Loader using `getSupportLoaderManager().initLoader()`.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Connect to the Internet with AsyncTask and AsyncTaskLoader](#)

Learn more

Android Developer Documentation

Guides

- [Connecting to the Network](#)
- [Managing Network State](#)
- [Loaders](#)

Reference

- [AsyncTask](#)
- [AsyncTaskLoader](#)

7.3: 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
- Summary
- Related concept
- Learn more

Certain events that can happen in the Android system might affect the functionality of applications installed on the device. For example, if the system has finished booting, you might like 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](#). A BroadcastReceiver is the base class for code that will receive Intents sent by `sendBroadcast()`. There are two major classes of broadcasts that can be received:

- Normal broadcasts (sent with [Context.sendBroadcast\(\)](#)) are completely asynchronous. All receivers of the broadcast are run in an undefined order, often at the same time. This is more efficient, but means that receivers cannot use the result or abort APIs included here.
- Ordered broadcasts (sent with [Context.sendOrderedBroadcast](#)) are delivered to one receiver at a time. As each receiver executes in turn, it can propagate a result to the next receiver, or it can completely abort the broadcast so that it won't be passed to other receivers. The order receivers run in can be controlled with the [android:priority](#) attribute of the matching intent-filter; receivers with the same priority will be run in an arbitrary order.

Even in the case of normal broadcasts, the system may in some situations revert to delivering the broadcast one receiver at a time. In particular, for receivers that may require the creation of a process, only one will be run at a time to avoid overloading the system with new processes. In this situation, however, the non-ordered semantics hold: these receivers still cannot return results or abort their broadcast.

Additionally, you can create Intents with custom actions and broadcast them yourself from your application using the `sendBroadcast()` method. The broadcast will be received by all applications with a BroadcastReceiver registered for that action. To learn more about broadcast Intents and Broadcast receivers, visit the [Intent documentation](#).

It is useful to note that while the Intent class is used for sending and receiving broadcasts, the Intent broadcast mechanism is completely separate from Intents that are used to start Activities.

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

Prior to this practical, you should be able to:

- Identify key parts of the `AndroidManifest.xml` file.
- Create Implicit Intents.

What you will LEARN

During this practical, you will learn to:

- Subclass and implement a BroadcastReceiver.
- Register for system Broadcast intents.
- Create and send custom Broadcast intents.

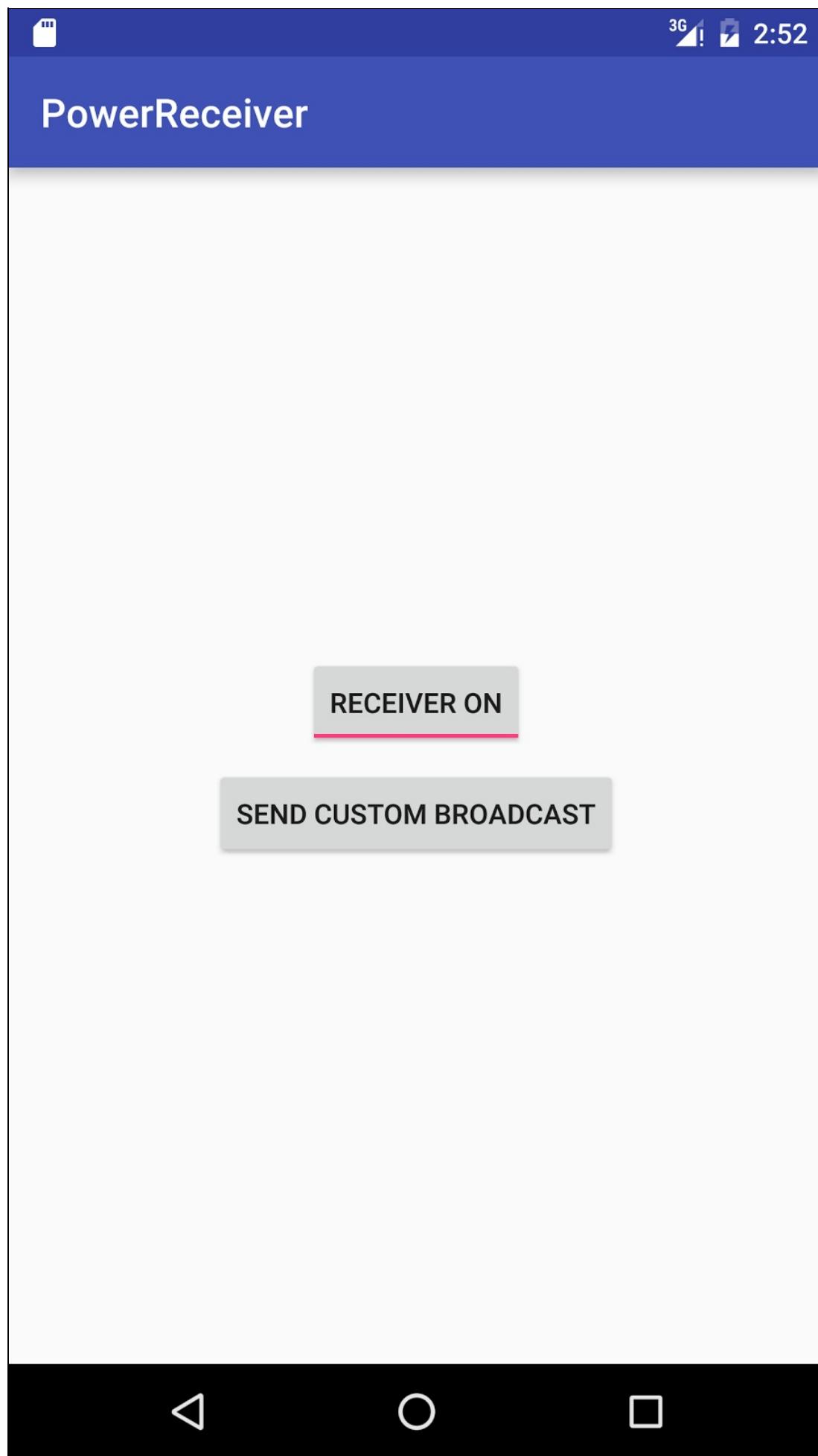
What you will DO

In this practical, you will:

- 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 message 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. Set up the PowerReceiver Project

1.1 Create the Project

1. Create a new project called **PowerReceiver**, accept the default options and use the Empty template.
2. Create a new Broadcast Receiver. Select the package name in the Android Project View and navigate to **File > New > Other > Broadcast Receiver**.
3. Name the class **CustomReceiver** 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.

4. Navigate to your Android manifest file. Note that Android Studio automatically generates a `<receiver>` tag with your chosen options as attributes. BroadcastReceivers can also be registered programmatically, but it is easiest to define them 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. You learned about implicit intents in an earlier practical.

1. In the **AndroidManifest.xml** file, add the following code between the `<receiver>` tags 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 with the `intentAction` string, so that your app 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 message 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 Restrict your Broadcast Receiver

Broadcast Receivers are always active, and therefore your app does not even need to be running for its `onReceive()` method to be called.

1. Go ahead, try it out: close your app, and plug or unplug your device.

The toast message is still displayed!

There is a lot of responsibility on you, as the developer, to not overwhelm your user with notifications or unwanted functionality every time a broadcast occurs. In this example, having a Toast message pop up every time the power state changes could quickly annoy the user. To limit this, you will add some code to ensure that the broadcast receiver is only active when the app is showing.

The [PackageManager](#) class is responsible for enabling and disabling a particular android component (such as a service, activity or broadcast receiver). This is accomplished using the `setComponentEnabledSetting()` method which takes three arguments:

- The ComponentName (an identifier for the component you want to enable or disable).
- One of the PackageManager class constants that represent the enabled state of a component. In this app we will use

`PackageManager.COMPONENT_ENABLED_STATE_ENABLED` and
`PackageManager.COMPONENT_ENABLED_STATE_DISABLED`. See the [PackageManager](#) reference for the other constants.

- An optional flag constant that tells the system not to kill the app when changing the state of the component: `PackageManager.DONT_KILL_APP`.

2. For the broadcast receiver to only be active when the app is showing, enable it in `onStart()` and disable it in `onStop()`.
3. Create two member variables: a PackageManager and a ComponentName.
4. Initialize both of them in `onCreate()`.

Instantiate the PackageManager with `getPackageManager()`. The constructor for ComponentName takes the application context and the class name of the component:

```
mReceiverComponentName = new ComponentName(this, CustomReceiver.class);  
mPackageManager = getPackageManager();
```

5. Override both `onStart()` and `onStop()`:

```
@Override  
protected void onStart() {  
    super.onStart();  
}  
@Override  
protected void onStop() {  
    super.onStop();  
}
```

6. Call `setComponentEnabledSetting()` on the PackageManager in `onStart()`. Pass in the

Component name, the `PackageManager.COMPONENT_ENABLED_STATE_ENABLED` constant, and the `DONT_KILL_APP` flag:

```
mPackageManager.setComponentEnabledSetting
(mReceiverComponentName, PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
PackageManager.DONT_KILL_APP);
```

7. In `onStop()`, use the `PackageManager` to disable the `CustomReceiver`, using the `PackageManager.COMPONENT_ENABLED_STATE_DISABLED` constant:

```
mPackageManager.setComponentEnabledSetting
(mReceiverComponentName, PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
PackageManager.DONT_KILL_APP);
```

Task 2. Send and Receive a Custom Broadcast

In addition to responding to system broadcasts, your application can also send and receive custom Broadcast Intents. A custom broadcast intent is exactly the same as a system one except you must define your own Intent action for it (a unique string) and it's delivered using the `sendBroadcast()` method. 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 message.

2.1 Define your custom Broadcast Action string

Both the sender and receiver of a custom broadcast must agree on a unique action string for the Broadcast Intent. It is a common practice to create unique action strings by prepending your Action Name with your package name.

1. Create a constant `String` variable in both your `MainActivity` and your `CustomReceiver` class to be used as the Broadcast Intent Action (this is your custom action string):

```
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
android:id	"@+id/sendBroadcast"
android:layout_width	wrap_content
android:layout_height	wrap_content
android:text	"Send Custom Broadcast"
android:layout_margin	"8dp"
android:onClick	"sendCustomBroadcast"

2. Extract your string resources.
3. Create the stub for the `sendCustomBroadcast()` method: Click in the yellow highlighted `onClick` method name. **Press Alt (Option for Mac users) + Enter** and choose '**Create 'sendCustomBroadcast(View)**' in 'MainActivity'.

2.3 Implement sendCustomBroadcast()

Because this broadcast is meant to be used solely by your application, you should use `LocalBroadcastManager` to manage the broadcasts in your application.

`LocalBroadcastManager` is a class that allows you to register for and send broadcasts of `Intents` to local objects within your app. By keeping broadcasts local, your application data will not be shared with other Android applications, keeping your information more secure and maintaining system efficiency.

1. In the `sendCustomBroadcast()` method in `MainActivity`, create a new `Intent`, with your custom action string as the argument.

```
Intent customBroadcastIntent = new Intent(ACTION_CUSTOM_BROADCAST);
```

2. Send the broadcast using the `LocalBroadcastManager` class:

```
LocalBroadcastManager.getInstance(this).sendBroadcast(customBroadcastIntent);
```

2.4 Register your Custom Broadcast

For system broadcasts, you registered your receiver in the `AndroidManifest.xml` file. It is also possible to register your receiver for specific actions programmatically. For broadcasts sent using `LocalBroadcastManager`, static registrations in the manifest is not allowed.

If you programmatically register the broadcast receiver, you must also unregister the receiver when it is no longer needed. In your application, the receiver will only need to respond to the custom broadcast when it is running, so we can therefore register the action in `onCreate()` and unregister it in `onDestroy()`.

1. Create a member variable in `MainActivity` for your Receiver and initialize it:

```
private CustomReceiver mReceiver = new CustomReceiver();
```

2. In `onCreate()`, 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

1. In `onReceive()` in your `CustomReceiver` class, add a case statement for the custom Intent Action.
2. Modify the toast message to "Custom Broadcast Received", extract it into your `strings.xml` and call it `custom_broadcast_toast` (press **Alt + Enter** or **Option + Enter** on a Mac and choose **extract string resource**):

```
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 toast messages, but not the Custom Broadcast Intent Toast messages.

That's it! Your app now delivers custom Broadcast intents and is able to receive both system and custom Broadcasts.

Solution code

Android Studio project: [PowerReceiver](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

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.

Summary

- Broadcast Receivers are one of the fundamental components of an android application.
- Broadcast Receivers can receive Intents broadcasted by both the system and application.
- The Intent broadcast mechanism is completely separate from Intents that are used to start Activities.
- You need to subclass the BroadcastReceiver class and implement onReceive() to process the incoming Intent associated with the broadcast.
- A broadcast receiver can be registered in the Android manifest file or programmatically.
- Use LocalBroadcastManager to register and send for Broadcasts that are private to your application.
- LocalBroadcastManager is more efficient and secure than system broadcasts.
- For broadcasts sent using LocalBroadcastManager, you can only register interest for specific actions programmatically.
- A common practice to create unique Intent action names for broadcasts is to prepend your Action Name with your package name.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Broadcast Receivers](#)

Learn more

Android Developer Documentation

Guides

- [Intents and Intent Filters](#)
- [Manipulating Broadcast Receivers On Demand](#)

Reference

- [BroadcastReceiver](#)

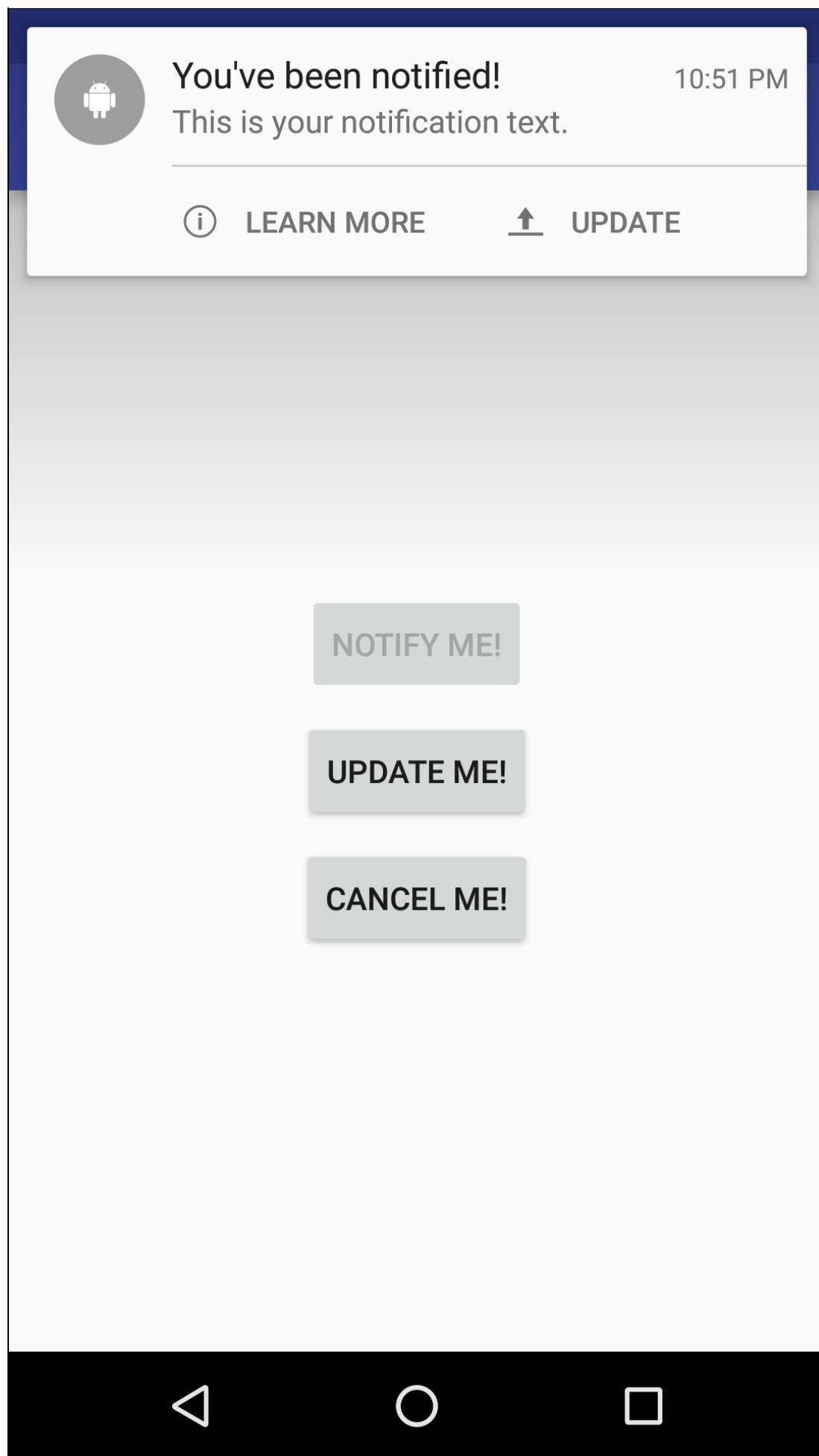
8.1: Notifications

Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Create a basic notification](#)
- [Task 2. Update and cancel your notification](#)
- [Task 3. Add notification actions](#)
- [Coding challenge](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

Until now, the apps you have built used 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 message when the device was connected or disconnected from power. There are many times when you want to show your user 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. The Android framework provides a mechanism for your app to notify users even when the app is not in the foreground: the [Notification](#) framework.

A Notification is a message you can display to the user outside of your application's normal UI. When Android issues a notification, it will first appear as an icon in the **notification area of the device**. To see the specific details of the notification, the user opens the **notification drawer**. Both the notification area and the notification drawer are system-controlled areas that the user can view at any time.



In this practical you'll create an app that triggers a notification when a button is pressed and provides the ability to update the notification or cancel it.

What you should already KNOW

For this practical, you should be able to:

- Implement the `onclick()` method for buttons.
- Create Implicit Intents.
- Send Custom Broadcast Intents.
- Use Broadcast Receivers.

What you will LEARN

During this practical, you will learn to:

- Create a Notification using the Notification Builder.
- Use Pending Intents to respond to Notification actions.
- Update or cancel existing Notifications.

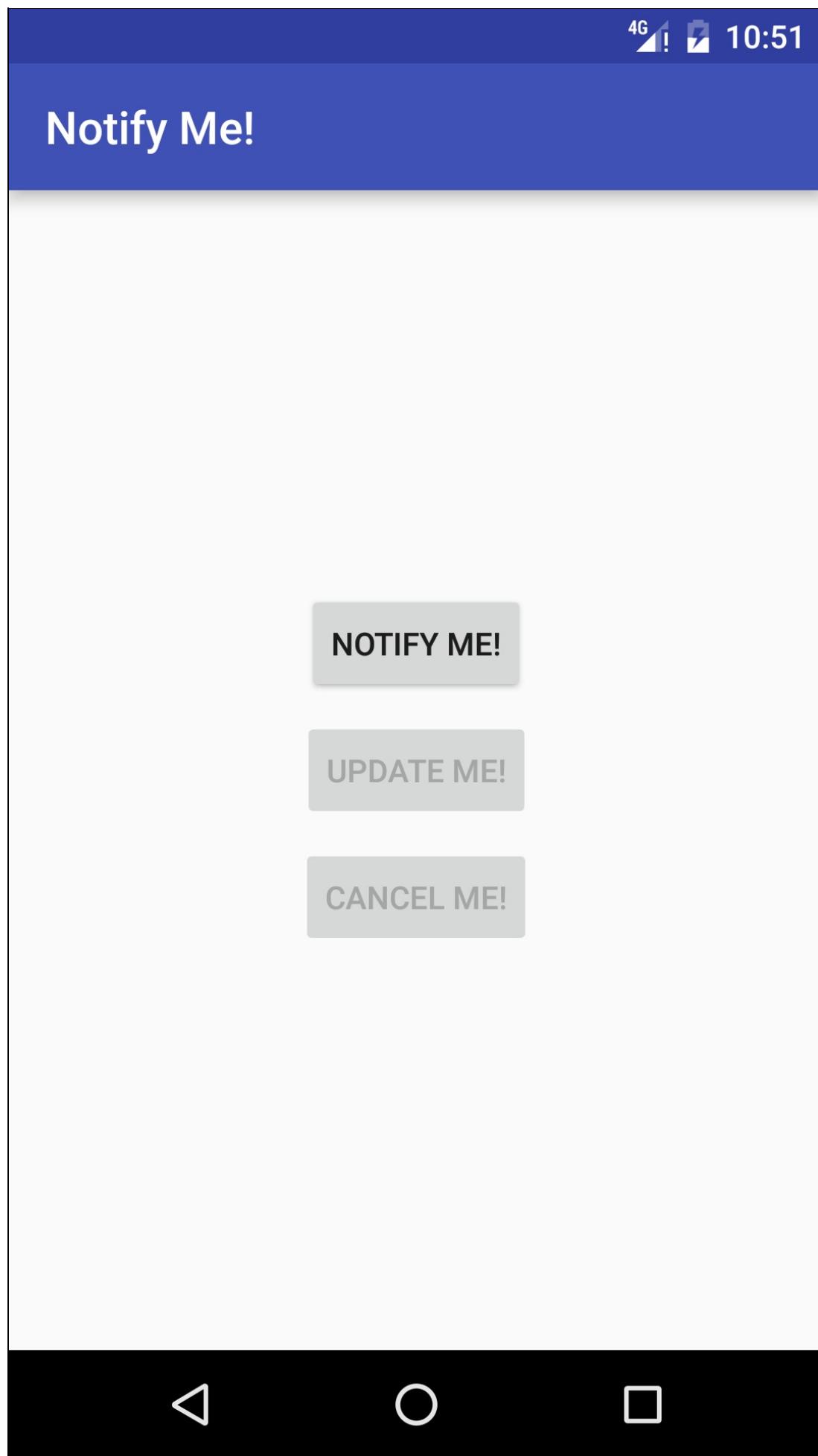
What you will DO

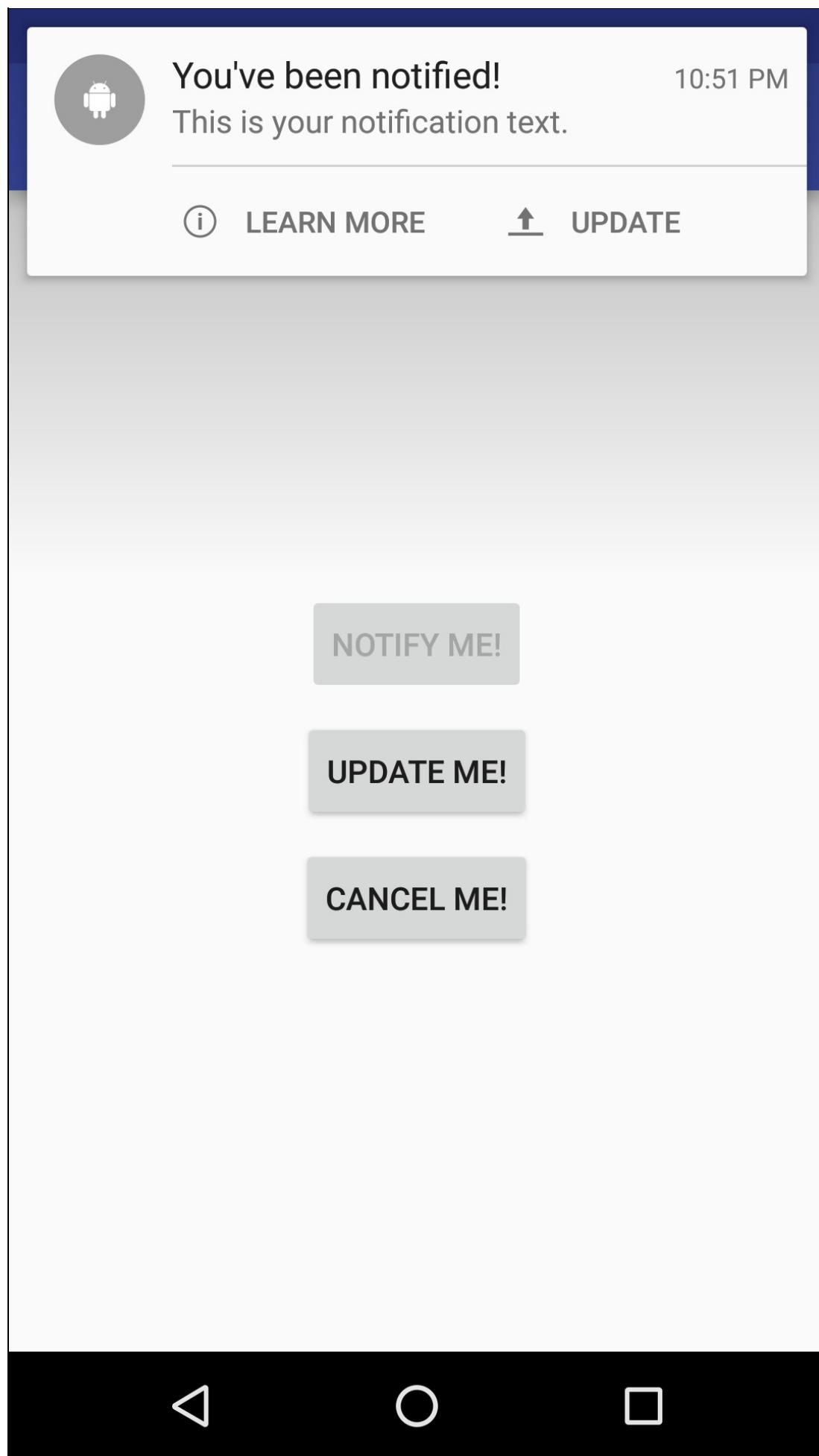
In this practical, you will:

- 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. Create a basic notification

1.1 Create the project

1. Create a new project called "Notify Me!", accept the default options, and use the empty template.
2. In your `activity_main.xml` file, change the rootview element to a vertical `LinearLayout` with its `gravity` attribute set to "center".
3. Add a button with the following attributes to replace the default `TextView`:

Attribute	Value
<code>android:id</code>	"@+id/notify"
<code>android:layout_width</code>	"wrap_content"
<code>android:layout_height</code>	"wrap_content"
<code>android:text</code>	"Notify Me!"
<code>android:layout_margin</code>	"4dp"

4. Create a method stub for the `sendNotification()` method. The method should take no arguments and return `void`:

```
public void sendNotification() {}
```

5. Create a member variable for the Notify Button.
6. Initialize the button in `onCreate()` and create an `onClickListener` for it:

```
mNotifyButton = (Button) findViewById(R.id.notify);
mNotifyButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
    }
});
```

7. Call `sendNotification()` from the `onClick` method.

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()`.

An Android Notification is deployed by the [NotificationManager](#). If you need to update or cancel the notification in the future, you should associate a notification ID with your Notification.

Create the Notification Icon

1. Go to **File > New > Image Asset**.
2. From the **Icon Type** dropdown, select **Notification Icons**.
3. Click on the icon next to the **Clip Art** item to select a material icon that you will use as the icon for your notification. In this example, you can use the Android icon.
4. Rename the resource `ic_android` and click **Next and Finish**. This will create a number of drawable files with different resolutions for different API levels.
5. Create a member variable in `MainActivity` to store the `NotificationManager`:

```
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. Create and instantiate the `Notification` Builder in the `sendNotification()` method:

```
NotificationCompat.Builder notifyBuilder = new NotificationCompat.Builder(this)
```

Note: Make sure the `NotificationCompat` class is imported from the v4 support library.

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.

```
NotificationCompat.Builder notifyBuilder = 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:

```
Notification myNotification = notifyBuilder.build();
mNotifyManager.notify(NOTIFICATION_ID, myNotification);
```

13. Run your app. The "Notify Me!" button now issues a notification (look for the icon in the status bar), but it's missing some essential features: there is no notification sound or vibration, clicking on the notification doesn't do anything. Let's add some additional 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, and is set by `setContentIntent()`.
- A priority, which determines how the system displays the notification with respect to other notifications, and is set by `setPriority()`.
- The default options, such as sounds, vibration and LED lights (if available), and is set by `setDefaults()`.

Tapping a notification launches an Intent. Content 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 a system or custom event. The major difference with an Intent in a notification is that it must be wrapped in a `PendingIntent`, which allows the notification to perform the action even if your application is not running. A `PendingIntent` is given to an external component (e.g. `NotificationManager`) which allows the external application to use your application's permissions to execute a predefined piece of code. In effect, it authorizes the notification to send the intent on the application's behalf.

For this example, the content intent of the notification (that is, the intent that is launched when the notification is pressed) will launch the `MainActivity` of the application (if you are already in the application this will have no effect).

1. Create an explicit intent in the `sendNotification()` method to launch the `MainActivity`

class:

```
Intent notificationIntent = new Intent(this, MainActivity.class);
```

- 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);
```

- Add the PendingIntent to the Notification using `setContentIntent()` in the `NotificationCompat.Builder`:

```
.setContentIntent(notificationPendingIntent)
```

- Run the app. Click the Notify Me! button to send the notification. Quit the app. Now view the notification and click it. Notice the app will open back up at the `MainActivity`.

1.4 Add priority and defaults to your notification

When your user clicks the "Notify Me!" button, the notification is issued but the only visual 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 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. 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 an LED indicator). In this example, you will use the default options by adding the following line to your `Builder`:

```
.setDefaults(NotificationCompat.DEFAULT_ALL)
```

- You need to quit the application and start it again to see the changes.

Note: The high priority notification will not drop down in front of the active screen unless both the priority and the defaults are set. The priority alone is not enough.

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.

In this task, you will learn how to update and cancel your notification.

2.1 Add update and cancel buttons

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. Add a member variable for each of the new buttons and initialize them in `onCreate()`.
5. Create two methods in the MainActivity that take no parameters and return void:

```
public void updateNotification() {}  
public void cancelNotification() {}
```

6. Create `onClick` Listeners for the new buttons and call `updateNotification()` in "update" button `onClick` method and `cancelNotification()` in the "cancel" button `onClick` method.

2.2 Implement the cancel and update notification methods

Cancel the Notification

Canceling a notification is straightforward: call `cancel()` on the `NotificationManager`, passing in the notification ID:

```
mNotifyManager.cancel(NOTIFICATION_ID);
```

Update the Notification

Updating a notification is more complex. Android notifications come with alternative styles that can help condense information or represent it more efficiently. 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.

1. Download [this image](#) to use in your notification.
2. Put it in the `res/drawables` folder.
3. In your `updateNotification()` method, convert your drawable into a bitmap:

```
Bitmap androidImage = BitmapFactory
    .decodeResource(getResources(), R.drawable.mascot_1);
```

4. Copy the Intent and PendingIntent you create in `sendNotification()` to `updateNotification()`, as you will use the same PendingIntent as a Content Intent.
5. Copy the `NotificationCompat.Builder` code from `sendNotification()` to `updateNotification()`, to have the same basic notification options in your updated notification.
6. Change the style of your notification in the same `NotificationCompat.Builder`, setting the image and the "Big Content Title":

```
.setStyle(new NotificationCompat.BigPictureStyle()
    .bigPicture(androidImage)
    .setBigContentTitle("Notification Updated!"));
```

Note: The `BigPictureStyle` is a subclass of `NotificationCompat.Style` which provides alternative layouts for notifications. See the documentation for other defined subclasses.

7. Change the priority of the Builder 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)
```

8. Call `notify()` on the `NotificationManager`, passing in the same notification ID as before.

```
mNotifyManager.notify(NOTIFICATION_ID, notifyBuilder.build());
```

9. Run your app. After clicking update, check the notification again. It now has the image and updated title! You can shrink back to the regular notification style by pinching on the extended one.

2.3 Toggle the button state

In this application, the user can get confused because the state of the notification is not tracked inside the activity. For example, the user may tap "Cancel Me!" when no notification is showing. You can fix this by enabling and disabling the various buttons depending on the state of the 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 the notification 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);
```

Task 3. Add notification actions

Sometimes, a notification requires immediate interaction: snoozing an alarm, replying to a text message, and so on. When these types of notifications occur, the user might tap your notification to respond to the event. Android then loads the proper Activity in your application for the user to respond. To avoid opening your application, the notification framework lets you embed a notification action directly in the notification itself. This allows 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 notification action is clicked.

For this example, you will add two actions to your notification. First you'll add a "Learn More" action with an implicit intent that launches a web page, then an "Update" action with a broadcast intent that updates your notification without launching the application.

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.

1. Create a member String variable that contains the URL to the Material Design guide for notifications: <https://developer.android.com/design/patterns/notifications.html>.
2. Create an implicit Intent that opens the saved URL in the `sendNotification()` method before you build the notification.
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. Add this icon using the Image Asset Studio, and call it `ic_learn_more`:
5. Add the following line of code to your builder in both `sendNotification()` and `updateNotification()` to add the action to both the original and updated notification:
`.addAction(R.drawable.ic_learn_more, "Learn More", learnMorePendingIntent);`

6. Run your app. Your notification will now have a clickable icon that takes you to the web!

3.2 Implement the "Update" action

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 a PendingIntent created using the `getActivity()` method. You can also create a PendingIntent which delivers a broadcast intent by calling `getBroadcast()` on the PendingIntent class. Broadcast Intents are very useful in notifications, since a broadcast receiver can register its interest in the intent and respond accordingly, entirely without launching a specific activity.

You will now implement a Broadcast Receiver that will call the `updateNotification()` method when the "Update" action in the notification is pressed. It is a common pattern to add functionality to a notification that already exists in the app, so the user does not need to launch any app to perform the action.

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()`.
3. Create a constant member variable in MainActivity to represent the update notification action for your BroadcastIntent. Make sure it begins with your package name to insure it's uniqueness:

```
private static final String ACTION_UPDATE_NOTIFICATION =
    "com.example.android.notifyme.ACTION_UPDATE_NOTIFICATION";
```

4. Create a member variable for your receiver and initialize it using the default constructor.
5. In the `onCreate()` method, register your Broadcast Receiver to receive the ACTION_UPDATE_NOTIFICATION intent:

```
registerReceiver(mReceiver, new IntentFilter(ACTION_UPDATE_NOTIFICATION));
```

6. Override the `onDestroy()` method of your Activity to 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 an inner class. When receivers are defined this way, they cannot be registered in the Android Manifest since they are dynamic and have the possibility of changing during the life of the application.

It may seem the broadcast sent by the notification only concerns your app and should be delivered with a LocalBroadcastManager. However, the use of PendingIntents delegates the responsibility of delivering the notification to the Android Framework. Because the Android runtime is handling the broadcast, LocalBroadcastManager can not be used.

Create the Update Action

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. Create this icon using the Image Asset Studio, call it `ic_update`.
4. Add the action to the builder in the `sendNotification()` method, giving it the title "Update":

```
.addAction(R.drawable.ic_update, "Update", updatePendingIntent)
```

5. Run your app. You can now update your notification without opening the app!

Solution code

Android Studio project: [NotifyMe](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: Enabling and disabling the various buttons is a common way to ensure the user does not perform any actions that are not supported in the current state of the app (think of disabling a "Sync" button when there is no network"). In this application, however, there is one use case in which the state of your buttons does not match the state of the application: when a user dismisses a 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.

Summary

- A Notification is a message you can display to the user outside of your application's normal UI.
- Notifications provide a way for your app to interact with the user even when the app is not running.
- When Android issues a notification, it will first appear as an icon in the notification area of the device.
- The UI and actions for a notification are specified using `NotificationCompat.Builder`.
- To create a notification use `NotificationCompat.Builder.build()` .
- To issue the notification, pass the Notification object to the Android runtime system with `NotificationManager.notify()` .
- To update or cancel a notification, you need to associate a notification ID with your Notification.
- An Intent can be part of a notification (Explicit, Implicit or Broadcast).
- Intents in a notification must be "wrapped" in a PendingIntent, which really isn't an Intent. A PendingIntent is an implementation of the decorator pattern.
- The required components of a notification are: small icon (`setSmallIcon()`), title (`setContentTitle()`) and some detailed text (`setContentText()`).
- Some optional components of a notification are: intent, expanded styles, priority, et al. See [NotificationCompat.Builder](#) for more info.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Notifications](#)

Learn more

Guides

- [Notifications](#)

- [Notification Design Guide](#)

Reference

- [NotificationCompat.Builder](#)
- [NotificationCompat.Style](#)

8.2: Alarm Manager

Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- Task 1. Set up the Parking Alarm Project and Views
- Task 2. Set up the Notification
- Task 3. Create the Repeating Alarm
- Coding challenge
- Summary
- Related concept
- Learn more

In your previous practicals, you've learned how to make your app respond to user interaction by pushing a button or tapping a notification. You've also learned how to make your app respond to system events using BroadcastReceivers. But what if your app needs to take action at a specific time, such as is the case with a calendar notification? In that case, you would use [AlarmManager](#), a class that allows you to launch and repeat a [PendingIntent](#) at a specific time and interval.

In this practical, you will create a timer that will remind you to stand up if you have been sitting for too long.

What you should already KNOW

From previous practicals, you should be able to:

- Implement onCheckChanged listeners for toggle buttons.
- Set up custom broadcast intents.
- Use broadcast receivers.
- Send notifications.

What you will LEARN

You will learn to:

- Schedule repeating alarms with AlarmManager.

- Check if an Alarm is already set up.
- Cancel a repeating alarm.

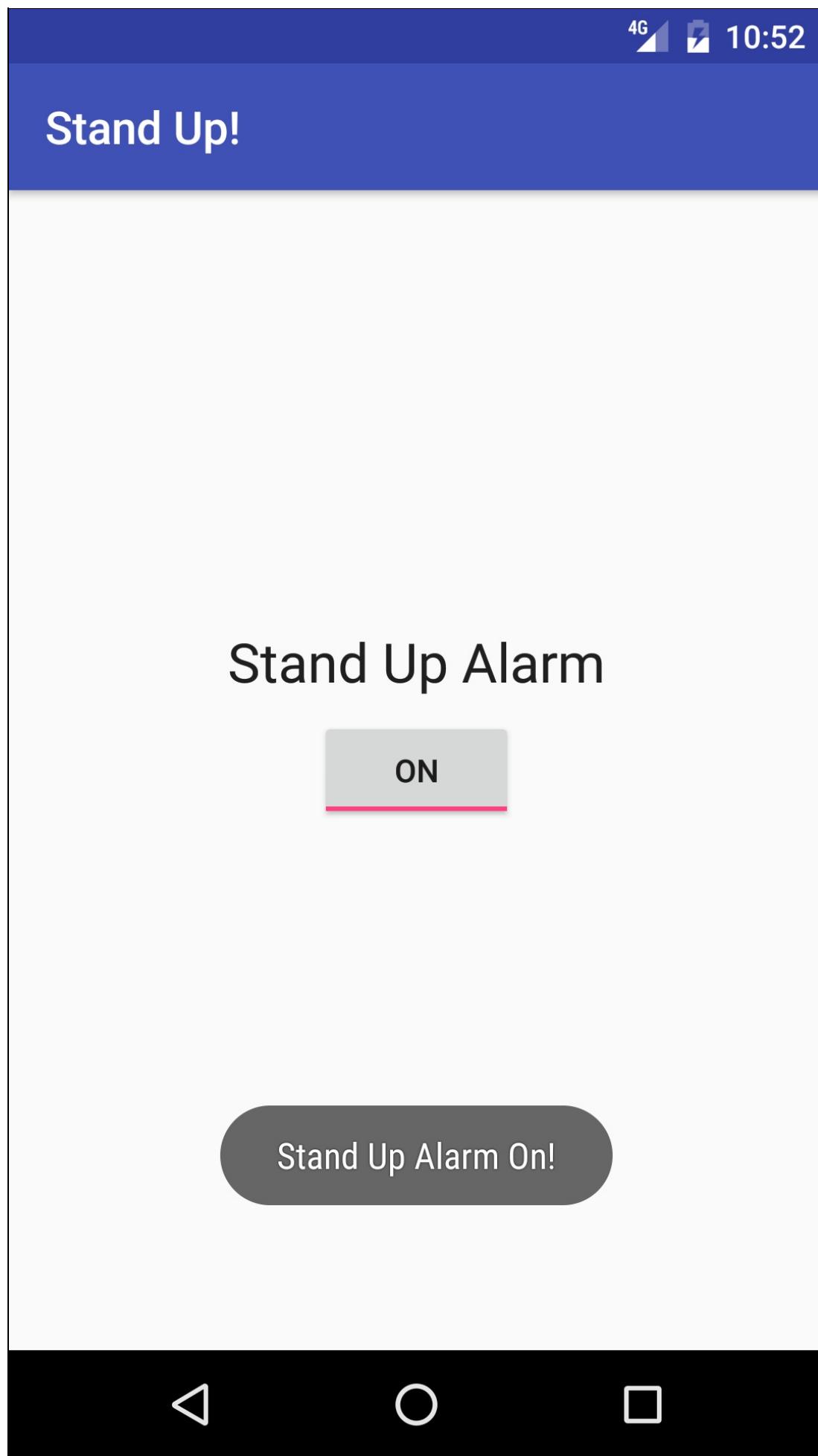
What you will DO

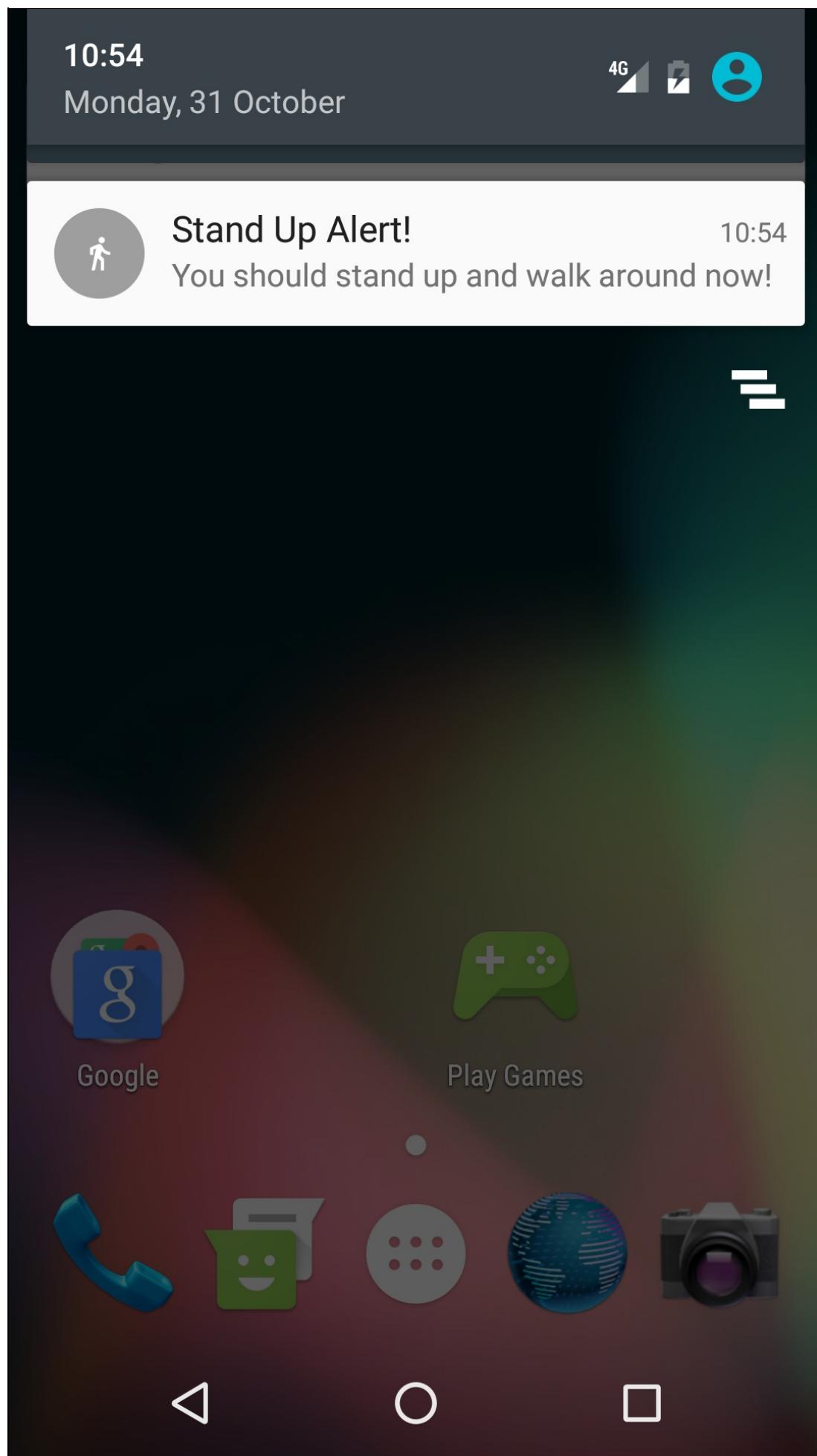
In this practical, you will:

- Set a repeating alarm to notify you every fifteen minutes.
- Use a ToggleButton to set and keep track of the alarm.
- Use Toast messages to notify the user when the Alarm is turned on or off.

App Overview

Stand Up! is an app that helps you stay healthy by reminding you to stand up and walk around every fifteen minutes. It uses a notification to let you know when fifteen minutes have passed. The app includes a toggle button that can turn the Alarm on and off.





Task 1. Setup the Stand Up! Project and Views

1.1 Create the Stand Up! Project layout

1. Create a new project called "Stand Up!", accept the default options and use the empty activity template .
2. Remove the default TextView and add the following elements:

TextView	Attribute	Value
	android:layout_width	"wrap_content"
	android:layout_height	"wrap_content"
	android:layout_above	"@+id/alarmToggle"
	android:layout_centerHorizontal	"true"
	android:layout_margin	"8dp"
	android:text	"Stand Up Alarm"
	android:textAppearance	"@style/TextAppearance.AppCo
ToggleButton	Attribute	Value
	android:id	"@+id/alarmToggle"
	android:layout_width	"wrap_content"
	android:layout_height	"wrap_content"
	android:layout_centerHorizontal	"true"
	android:layout_centerVertical	"true"

1.2 Setup the `setOnCheckedChangeListener()` method

The Stand Up! app includes a toggle button that is used to set and cancel the alarm, as well as visibly represent the alarm's current status. To set the alarm when the toggle is turned on, you will use the `onCheckedChangeListner()` method:

1. In your `MainActivity onCreate()` method, find the Alarm Toggle by id.
2. Call `setOnCheckedChangeListener()` on the toggle button instance, and begin typing "`new OnCheckedChangeListener`". Android Studio will autocomplete the method for you, including the required `onCheckedChanged()` override method. This method has two parameters: the `CompoundButton` that was clicked (in this case it's the Alarm Toggle button), and a boolean representing the current state of the Toggle Button (i.e., whether the toggle is now set on or off).

```
alarmToggle.setOnCheckedChangeListener(  
    new CompoundButton.OnCheckedChangeListener() {  
  
        @Override  
        public void onCheckedChanged(CompoundButton compoundButton,  
            boolean isChecked) {  
        }  
    });
```

3. It is useful for the user to have some feedback other than the toggle button being turned on and off to indicate the alarm was indeed set (you haven't implemented the alarm yet, you will do that in a further section). Set up an if/else block using the boolean parameter in the `onCheckedChanged()` method that delivers a toast message to tell the user if the Alarm was turned on or off. Don't forget to extract your string resources.

```
String toastMessage;  
if(isChecked){  
    //Set the toast message for the "on" case  
    toastMessage = getString(R.string.alarm_on_toast);  
} else {  
    //Set the toast message for the "off" case  
    toastMessage = getString(R.string.alarm_off_toast);  
}  
  
//Show a toast to say the alarm is turned on or off  
Toast.makeText(MainActivity.this, toastMessage, Toast.LENGTH_SHORT)  
.show();
```

Task 2. Set up the Notification

The next step is to create the notification that will remind the user to stand up every fifteen minutes. For now, the notification will be delivered immediately when the toggle is set.

2.1 Create the notification

In this step, you will create a `deliverNotification()` method that will post the reminder to stand up and walk around.

1. Create a member variable in `MainActivity` called `mNotificationManager` of type `NotificationManager`.
2. Initialize it in `onCreate()` by calling `getSystemService()` :

```
mNotificationManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
```

3. Create a method in `MainActivity` called `deliverNotification()` that takes the Context as an argument and does not return anything.

```
private void deliverNotification(Context context) {}
```

4. Create a member constant in `MainActivity` called `NOTIFICATION_ID` and set it to 0. Your app will only have one notification at a time, so you will use the same notification id for all notifications.

Note: Notification ID's are used to distinguish notifications within your application. The `NotificationManager` will only be able to cancel notifications delivered from your app so you can use the same ID in different applications.

Notification Content Intent

1. Create an Intent in `onCreate()` that you will use for the notification Content Intent:

```
Intent contentIntent = new Intent(context, MainActivity.class);
```

2. Create a `PendingIntent` from content Intent right below the definition of `contentIntent` using the `getActivity()` method, passing in the notification ID and using the `FLAG_UPDATE_CURRENT` flag:

```
PendingIntent contentPendingIntent = PendingIntent.getActivity
    (context, NOTIFICATION_ID, contentIntent, PendingIntent.FLAG_UPDATE_CURRENT);
```

Note: `PendingIntent` flags tell the system how to handle the situation when multiple instances of the same `PendingIntent` are created (meaning they contain the same intent). The `FLAG_UPDATE_CURRENT` flag tells the system to use the old Intent but replace the extras data. Since you don't have any extras in this Intent, you reuse the same `PendingIntent` over and over.

Notification Title and Text

1. Create a string resource in your `strings.xml` file called `notification_title`. Set it equal to "Stand Up Alert".

2. Create a string resource in your strings.xml file called notification_text. Set it equal to "You should stand up and walk around now!".

Notification Icon

1. Add an image asset to use as the notification icon (use the [Image Asset Studio](#)).



Choose any icon you find appropriate for this alarm:

Build the notification

1. Use the NotificationCompat.Builder to build a notification in the

```
deliverNotification() method using the above notification title, text, icon and content intent.
```

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(context)
    .setSmallIcon(R.drawable.ic_stand_up)
    .setContentTitle(context.getString(R.string.notification_title))
    .setContentText(context.getString(R.string.notification_text))
    .setContentIntent(contentPendingIntent)
```

2. Set the Notification priority to PRIORITY_HIGH:

```
.setPriority(NotificationCompat.PRIORITY_HIGH)
```

3. Add an option to the builder to set AutoCancel to true, and another option to use the default light, sound and vibration pattern:

```
.setAutoCancel(true)
.setDefaults(NotificationCompat.DEFAULT_ALL);
```

Deliver the notification

1. Use the NotificationManager to deliver the notification:

```
mNotificationManager.notify(NOTIFICATION_ID, builder.build());
```

2. Call `deliverNotification()` when the alarm toggle button is turned on, passing in the activity context:
3. Call `cancelAll()` on the NotificationManager if the toggle is turned off to remove the notification.

```
if(isChecked){  
    deliverNotification(MainActivity.this);  
    //Set the toast message for the "on" case  
    toastMessage = getString(R.string.alarm_on_toast);  
} else {  
    //Cancel notification if the alarm is turned off  
    mNotificationManager.cancelAll();  
  
    //Set the toast message for the "off" case  
    toastMessage = getString(R.string.alarm_off_toast);  
}
```

4. Run the app, and check that the notification is delivered with all the desired options.

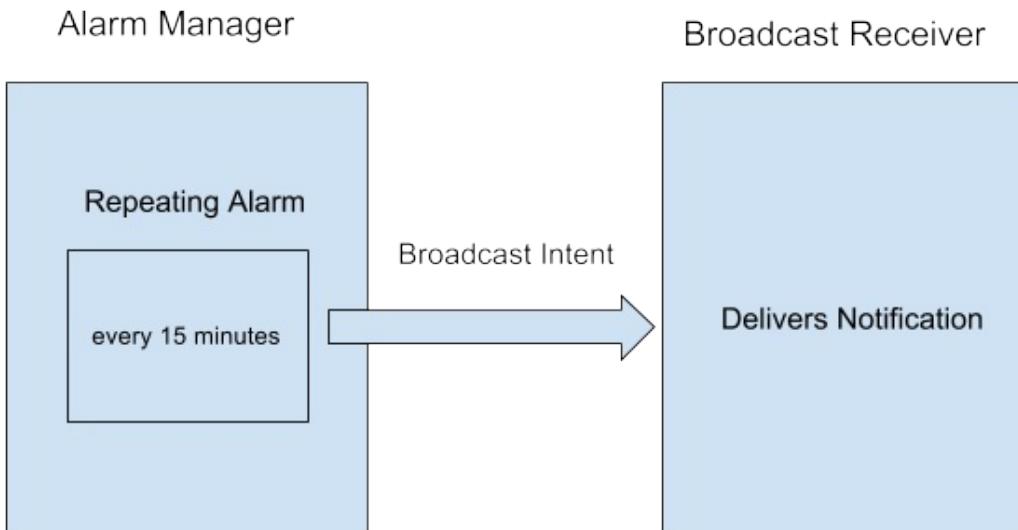
At this point there is no alarm at all: the notification is immediately delivered when the alarm toggle is turned on. In the next section you will implement the `AlarmManager` to schedule and deliver the notification every 15 minutes.

Task 3. Create the Repeating Alarm

Now that your app can send a notification it is time to implement the main component of your application: the [AlarmManager](#). This is the class that will be responsible for periodically delivering the reminder to stand up. `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](#).

`AlarmManager`, like notifications, uses a [PendingIntent](#) that it delivers with the specified options. Because of this, it can deliver the Intent even when the application is no longer running. In this application, your `PendingIntent` will deliver an Intent broadcast with a custom "Notify" action.

The broadcast intent will be received by a broadcast receiver that takes the appropriate action (delivers the notification).



The [AlarmManager](#) can trigger one-time or recurring events which occur even when the device is in deep sleep or your application is not running. Events may be scheduled with your choice of [currentTimeMillis\(\)](#) when using the real time version (RTC) or [elapsedRealtime\(\)](#) when using the elapsed time version (ELAPSED_REALTIME), and deliver a PendingIntent when they occur. For more information on the different clocks available and information on how to control the timing of events, please refer to the [SystemClock Developer Reference](#).

3.1 Set up the broadcast pending intent

The AlarmManager is responsible for delivering your PendingIntent at a specified interval. This PendingIntent will deliver a broadcast intent letting the application know it is time to update the remaining time in the notification.

1. Create a string constant as a member variable in MainActivity to be used as the broadcast intent action which will deliver the notification:

```

private static final String ACTION_NOTIFY =
    "com.example.android.standup.ACTION_NOTIFY";

```

Note: use the fully-qualified package name for the Intent string, to ensure that your Broadcast is unique, and can not accidentally be used by other applications with similar actions.

2. Create an Intent called `notifyIntent` in `onCreate()` with the custom string as its action:

```
Intent notifyIntent = new Intent(ACTION_NOTIFY);
```

3. Create the notify PendingIntent using the context, the `NOTIFICATION_ID` variable, the new `notifyIntent`, and the `PendingIntent` flag `UPDATE_CURRENT`:

```
PendingIntent notifyPendingIntent = PendingIntent.getBroadcast
    (this, NOTIFICATION_ID, notifyIntent, PendingIntent.FLAG_UPDATE_CURRENT);
```

3.2 Set the repeating alarm

You will now use the `AlarmManager` to deliver this broadcast Intent every 15 minutes. For this task, the appropriate type of alarm is an inexact, repeating alarm that uses elapsed time and will wake the device up if it is asleep. The real time clock is not relevant here, since we want to deliver the notification every fifteen minutes.

1. Initialize the `AlarmManager` in `onCreate()` by calling `getSystemService()` :

```
AlarmManager alarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
```

2. In the `onCheckedChanged()` method, call `setInexactRepeating()` on the alarm manager instance when the user clicks the Alarm "ON" (The second parameter is `true`). You will use the `setInexactRepeating()` alarm since it is more resource efficient to use inexact timing (the system can bundle alarms from different apps together) and it is acceptable for your alarm to deviate a little bit from the exact 15 minute repeat interval. The `setInexactRepeating()` method takes 4 arguments:
 - The [alarm type](#). In this case you will use the elapsed time since boot type, since only the relative time is important. You also want to wakeup the device if it's asleep, so the alarm type is `ELAPSED_REALTIME_WAKEUP`.
 - The trigger time in milliseconds. For this, use the current elapsed time, plus 15 minutes. To get the current elapsed time, you can call `SystemClock.elapsedRealtime()`. You can then use a built-in `AlarmManager` constant to add 15 minutes to the elapsed time:
`AlarmManager.INTERVAL_FIFTEEN_MINUTES` .
 - The time interval in milliseconds. You want the notification posted every 15 minutes. You can use the `AlarmManager.INTERVAL_FIFTEEN_MINUTES` constant again.
 - The `PendingIntent` to be delivered. You created the `PendingIntent` in the previous task.

```

long triggerTime = SystemClock.elapsedRealtime()
    + AlarmManager.INTERVAL_FIFTEEN_MINUTES;

long repeatInterval = AlarmManager.INTERVAL_FIFTEEN_MINUTES;

//If the Toggle is turned on, set the repeating alarm with a 15 minute interval
alarmManager.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
    triggerTime, repeatInterval, notifyPendingIntent);

```

Note: Because you are accessing the AlarmManager and notifyPendingIntent instances from an anonymous inner class, Android Studio may make these instances final. If it doesn't, you have to make them final yourself.

7. Remove the call to `deliverNotification()` in the `onCheckedChanged()` method.
8. If the alarm toggle is turned off (by clicking the toggle in the ON state), cancel the alarm by calling `cancel()` on the AlarmManager, passing in the pending intent used to create the alarm.

```
alarmManager.cancel(notifyPendingIntent);
```

Keep the call to `cancelAll()` on the NotificationManager, since turning the toggle off should still remove any existing notification.

The AlarmManager will now start delivering your Broadcast Intent starting fifteen minutes from when the Alarm was set, and every fifteen minutes after that. Your application needs to be able to respond to these intents by delivering the notification. In the next step you will subclass a BroadcastReceiver to receive the broadcast intents and deliver the notification.

3.3 Create the Broadcast Receiver

The Broadcast Receiver is responsible for receiving the broadcast intents from the AlarmManager and reacting appropriately.

1. In Android Studio, click on **File > New > Other > Broadcast Receiver**.
2. Enter AlarmReceiver for the name, make sure the Exported checkbox is unchecked (to ensure that other apps will not be able to invoke this Broadcast Receiver). You can also change this setting in the AndroidManifest by setting the `android:exported` attribute to `false`. Android Studio will create the subclass of BroadcastReceiver with the required method (`onReceive()`), as well as add the receiver to your AndroidManifest. You need to add an Intent Filter to your the `<receiver>` tag in the AndroidManifest to select the proper incoming Broadcast Intents.
3. In the Android Manifest, create an `<intent-filter>` opening and closing tag between the `<receiver>` tags. Create an `<action>` item in the intent filter with `android:name`

attribute set to the custom ACTION_NOTIFY action string you created:

```
<intent-filter>
    <action android:name="com.example.android.standup.ACTION_NOTIFY" />
</intent-filter>
```

4. Cut and paste the `deliverNotification()` method to the `onReceive()` method in the BroadcastReceiver and call it from `onReceive()`. The notification manager and notification id has not been initialized in the BroadcastReceiver class so it will be highlighted in red.
5. Copy the NOTIFICATION_ID variable from the MainActivity into the BroadcastReceiver class.
6. Initialize the NotificationManager at the beginning of the `onReceive()` method. You have to call `getSystemService()` from the passed in Context:

```
NotificationManager notificationManager = (NotificationManager)
    context.getSystemService(Context.NOTIFICATION_SERVICE);
```

7. Remove the line that raises the `UnsupportedOperationException`.
8. Run your app. If you don't want to wait for fifteen minutes to see the notification, you can change the trigger time to `SystemClock.elapsedRealtime()` to see the notification immediately. You can also change the interval to a shorter time to make sure that the repeated alarm is working.

You now have an app that can schedule and perform a repeated operation, even if the application is no longer running. Go ahead, exit the application completely, the notification will still be delivered. There is one final component missing that would ensure a proper user experience: if the application is exited, the toggle button will reset to the off state, even if the alarm has already been set. To fix this, you will need to check the state of the alarm every time the application is launched.

3.5 Check the State of the Alarm

To track the state of the alarm, you will need a boolean variable that is `true` if the Alarm already exists, and `false` otherwise. To set this boolean, you can call

`PendingIntent.getBroadcast()` with the `FLAG_NO_CREATE` PendingIntent flag. In this case, the PendingIntent is returned if it already exists, otherwise the call returns null. This is extremely useful for checking whether the alarm has already been set.

Note: When you create a PendingIntent, the system uses the `Intent.filterEquals()` method to determine if a PendingIntent with the same Intent already exists. This means that to have two distinct PendingIntents, the contained Intents have to differ in one of action, data, type, class, or categories. Intent extras are not included in the comparison. The PendingIntent flag

determines what happens when a PendingIntent whose Intent matches the one you are trying to create already exists. In the case of the NO_CREATE flag, it will return null unless a PendingIntent with a matching Intent already exists.

1. Create a boolean that is true if PendingIntent is not null, and false otherwise, using this strategy. Use this boolean to correctly set the state of the ToggleButton when your app starts. This code has to come before the PendingIntent has been created, otherwise it will always return true:

```
boolean alarmUp = (PendingIntent.getBroadcast(this, NOTIFICATION_ID, notifyIntent,  
PendingIntent.FLAG_NO_CREATE) != null);
```

2. Set the checked state of the toggle right after you define the alarmUp boolean:

```
alarmToggle.setChecked(alarmUp);
```

This ensures that the toggle will always be turned on if the Alarm is set, and off otherwise. That's it, You now have a repeated scheduled alarm to remind you to stand up every fifteen minutes.

3. Run your app. Switch on the alarm. Exit the app. Open the app again. The alarm button will show that the alarm is on.

Solution code

**Android Studio project: [StandUp](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

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.

Summary

- AlarmManager allows you to schedule tasks based on the real time clock or the elapsed

time since boot.

- AlarmManager provides a variety of alarm types, both periodic and one time, with options to wake up your device if it is asleep.
- AlarmManager is meant for situations where precise timing is critical (such as a calendar event). Otherwise, consider the [Job Scheduler](#) framework for more resource-efficient timing and scheduling.
- Use the inexact timing version of the AlarmManager whenever possible, to minimize the load caused by multiple users' devices or multiple applications performing a task at the exact same time.
- AlarmManager uses PendingIntents to perform the operations, so you can schedule broadcasts, services and activities using the appropriate PendingIntent.

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Scheduling Alarms](#)

Learn more

Android Developer Documentation

Guides

- [Scheduling Repeating Alarms](#)

Reference

- [AlarmManager](#)
- [SystemClock](#)

Other Web resources

- [Blog Post on choosing the correct alarm type](#)

8.3: Job Scheduler

Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- Task 1. Implement a Job Service
- Task 2. Implement the job conditions
- Coding challenge
- Summary
- Related concept
- Learn more

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 be scheduled based on a combination of system and user requirements. For example, a news app might like to update the news in the morning, but could wait until the device is charging and connected to wifi to update the news, to preserve the user's data and system resources.

The `JobScheduler` class is meant for this kind of scheduling; it allows you to set the conditions, or parameters of running your task. Given these conditions, the `JobScheduler` calculates the best time to schedule the execution of the job. Some examples of these parameters are: persistency of the job across reboots, the interval that the job should run at, whether or not the device is plugged in, or whether or not the device is idle.

The task to be run is implemented as a `JobService` subclass and executed according to the specified 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 that schedules a notification to be posted when the parameters set by the user are fulfilled, and the system requirements are met.

What you should already KNOW

From the previous practicals, you should be able to:

- Deliver a notification.
- Get an integer value from a Spinner view.
- Use Switch views for user input.
- Create PendingIntents.

What you will LEARN

You will learn to:

- Implement a JobService.
- Construct a JobInfo object with specific constraints.
- Schedule a JobService based on the JobInfo object.

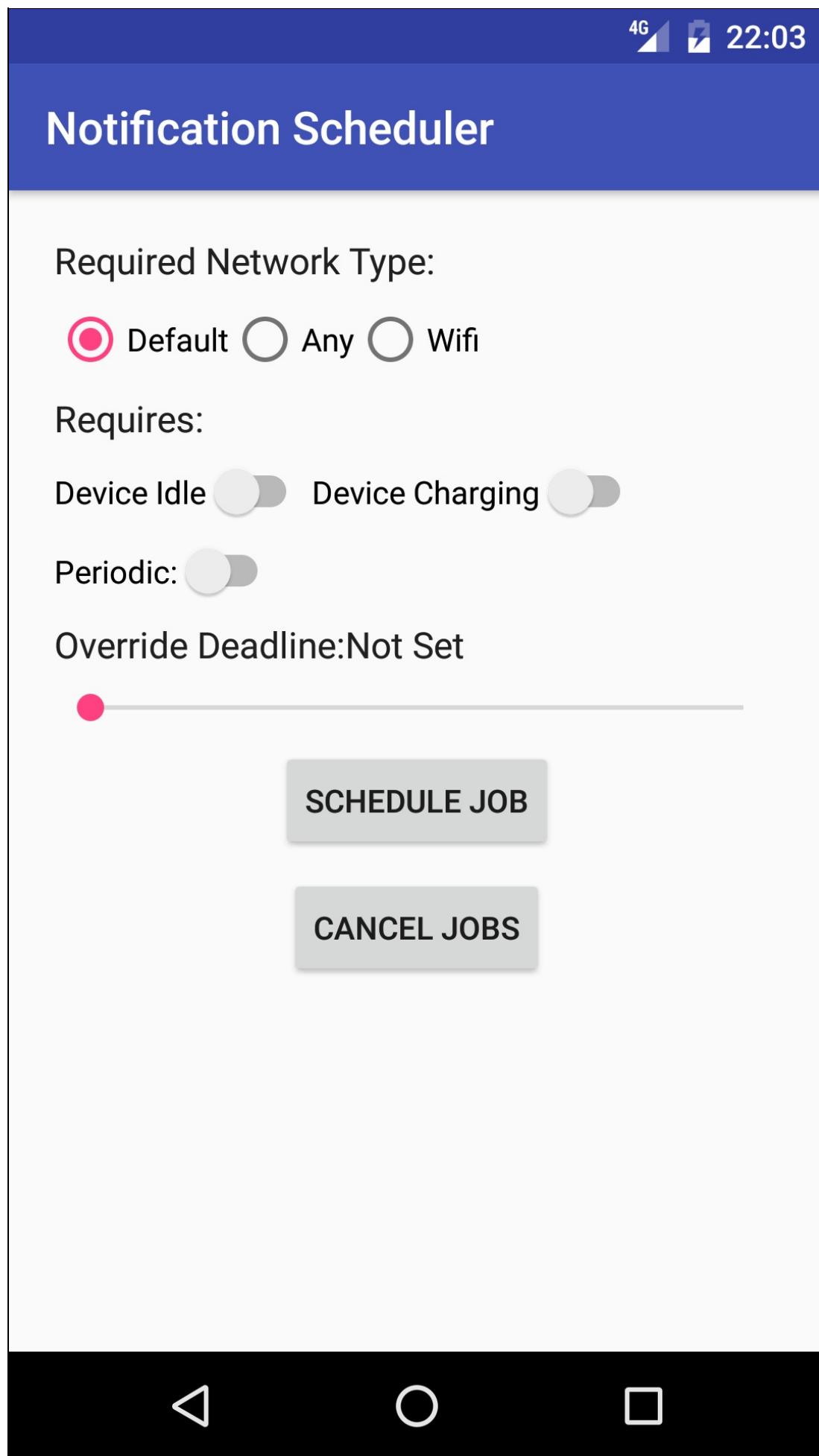
What you will DO

In this practical, you will:

- Implement a JobService that delivers a simple notification to let the user know the job is running.
- Get user input to configure the constraints (such as waiting until the device is charging) on the JobService you are scheduling.
- Schedule the job using JobScheduler.

App Overview

For this practical you will create an app called "Notification Scheduler". Your app will demonstrate the JobScheduler framework by allowing the user to select constraints and schedule a job. When that job is executed, it will post a notification (in this app, your notification is effectively your "job").



To use the JobScheduler, you need two additional parts: [JobService](#) and [JobInfo](#). A JobInfo object contains the set of conditions that will trigger the job to run. A JobService is the implementation of the job that is to run under those conditions.

Task 1. Implement a JobService

To begin with, you must create a service that will be run at the time determined by the conditions. The JobService is automatically executed by the system, and the only parts you need to implement are:

onStartJob() callback

- called when the system determines that your task should be run. You implement the job to be done in this method.
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 safely on the main thread.
- returns a boolean indicating whether the job needs to continue on a separate thread. If true, the work is offloaded to a different thread, and your app must call `jobFinished()` explicitly in that thread to indicate that the job is complete. If the return value is false, the framework knows that the job is completed by the end of `onStartJob()` and it will automatically call `jobFinished()` on your behalf.

onStopJob() callback

- called if the conditions are no longer met, meaning that the job must be stopped.
- returns a boolean that determines what to do if the job is not finished. If the return value is true, the job will be rescheduled, otherwise, it will be dropped.

1.1 Create the Project and the NotificationJobService

Verify that the minimum SDK you are using is API 21. Prior to API 21, JobScheduler does not work, as it is missing some of the required APIs.

1. Use the empty template , and create a new project called "Notification Scheduler".
2. Create a new Java class called NotificationJobService that extends JobService.
3. Add the required methods: `onStartJob()` and `onStopJob()` .
4. In your AndroidManifest.xml file, register your JobService with the following permission inside the `<application>` tag:

```
<service
    android:name=".NotificationJobService"
    android:permission="android.permission.BIND_JOB_SERVICE"/>
```

1.2 Implement onStartJob()

1. Add a notification icon for the "Job Running" notification.
2. In `onStartJob()`, create a PendingIntent to launch the MainActivity of your app to be used as the content intent for your notification.
3. In `onStartJob()`, construct and deliver a notification with the following attributes:

Attribute	Title
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

4. Make sure `onStartJob()` returns false, because all of the work is completed in that callback.
5. Make `onStopJob()` return true, so that the job is rescheduled if it fails.

```
@Override
public boolean onStartJob(JobParameters jobParameters) {
    //Set up the notification content intent to launch the app when clicked
    PendingIntent contentPendingIntent = PendingIntent.getActivity
        (this, 0, new Intent(this, MainActivity.class),
        PendingIntent.FLAG_UPDATE_CURRENT);

    NotificationManager manager =
        (NotificationManager) getSystemService(NOTIFICATION_SERVICE);

    NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
        .setContentTitle(getString(R.string.job_service))
        .setContentText(getString(R.string.job_running))
        .setContentIntent(contentPendingIntent)
        .setSmallIcon(R.drawable.ic_job_running)
        .setPriority(NotificationCompat.PRIORITY_HIGH)
        .setDefaults(NotificationCompat.DEFAULT_ALL)
        .setAutoCancel(true);

    manager.notify(0, builder.build());

    return false;
}
```

Task 2. Implement the job conditions

Now that you have your JobService, it is time to identify the criteria for running the job. For this, use the JobInfo component. You will create a series of parameterized conditions for running a job using a variety of network connectivity types and device status.

To begin, you will create a group of radio buttons to determine the network type required for this job.

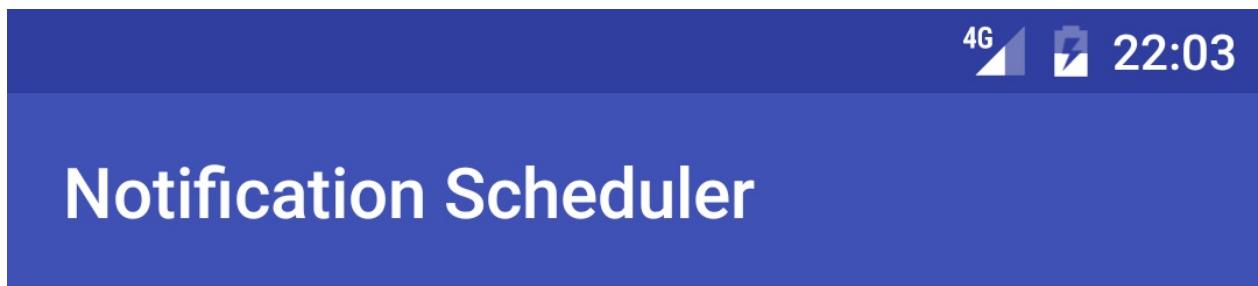
2.1 Implement the network constraint

One of the possible conditions for running a Job is the status of your device's network connectivity. You can limit the JobService to be executed only when certain network conditions are met. The options are:

- **NETWORK_TYPE_NONE**: the job will run with or without a network connection. This is the default value.
- **NETWORK_TYPE_ANY**: the job will run as long as a network (cellular, wifi) is available.
- **NETWORK_TYPE_UNMETERED**: the job will run as long as the device is connected to wifi that does not use a HotSpot.

Create the layout for your app

Create the layout for your app to show the buttons for the user to choose the network criteria.



Required Network Type:

Default Any Wifi

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 attributes:

RadioButton 1	
android:text	"None"
android:id	"@+id/noNetwork"
android:checked	true
RadioButton 2	
android:text	"Any"
android:id	"@+id/anyNetwork"
RadioButton 3	
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:

Button 1	
android:text	"Schedule Job"
android:onClick	"scheduleJob"
android:layout_gravity	"center_horizontal"
android:layout_margin	"4dp"
Button 2	
android:text	"Cancel Jobs"
android:onClick	"cancelJobs"
android:layout_gravity	"center_horizontal"
android:layout_margin	"4dp"

6. Add the method stubs for both of the `onClick()` methods in MainActivity.

Get the selected network option

1. In `scheduleJob()`, find the RadioGroup by id and save it in an instance variable called `networkOptions`.
2. Get the selected network id and save it in a integer variable:

```
int selectedNetworkID = networkOptions.getCheckedRadioButtonId();
```

3. Create a selected network option integer variable and set it equal to the default network option (no network required):

```
int selectedNetworkOption = JobInfo.NETWORK_TYPE_NONE;
```

4. Create a switch statement with the selected network id, and add a case for each of the possible id's:

```

switch(selectedNetworkID){
    case R.id.noNetwork:
        break;
    case R.id.anyNetwork:
        break;
    case R.id.wifiNetwork:
        break;
}

```

5. Assign the selected network option the appropriate JobInfo network constant, depending on the case:

```

switch(selectedNetworkID){
    case R.id.noNetwork:
        selectedNetworkOption = JobInfo.NETWORK_TYPE_NONE;
        break;
    case R.id.anyNetwork:
        selectedNetworkOption = JobInfo.NETWORK_TYPE_ANY;
        break;
    case R.id.wifiNetwork:
        selectedNetworkOption = JobInfo.NETWORK_TYPE_UNMETERED;
        break;
}

```

Create the JobScheduler and the JobInfo object

1. In MainActivity, create a member variable for the JobScheduler, and initialize it in `scheduleJob()` using `getSystemService()`:

```
mScheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
```

2. Create a member constant for the JOB_ID, and set it equal to 0.
3. Create a `JobInfo.Builder` object in `scheduleJob()`. The constructor for the `JobInfo.Builder` class takes two parameters:
 - The JOB_ID.
 - The ComponentName for the JobService you created. A ComponentName is used to identify the JobService with the JobInfo object.

```

ComponentName serviceName = new ComponentName(getPackageName(),
NotificationJobService.class.getName());
JobInfo.Builder builder = new JobInfo.Builder(JOB_ID, serviceName)

```

4. Call `setRequiredNetworkType()` on the `JobInfo.Builder` object, passing in the selected network option:

```
.setRequiredNetworkType(selectedNetworkOption);
```

5. Call `schedule()` on the JobScheduler object, passing in the JobInfo object with the `build()` method:

```
JobInfo myJobInfo = builder.build();
mScheduler.schedule(myJobInfo);
```

6. Show a Toast message, letting the user know the job was scheduled.
7. 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(this, "Jobs Canceled", Toast.LENGTH_SHORT).show();
}
```

8. Run the app. You can now set tasks that have network restrictions and see how long it takes for them to be executed. In this case, the task is to deliver a notification. To dismiss the notification, either swipe it away or tap on it to open the notification.

You may notice that if you do not change the network constraint to either "Any" or "Wifi", the app will crash with the following exception:

```
java.lang.IllegalArgumentException:
    You're trying to build a job with no constraints, this is not allowed.
```

This is because the "No Network Required" condition is the default and does not actually count as a constraint. The JobScheduler needs at least one constraint to properly schedule the JobService. In the following section you will create a conditional that is `true` when at least one constraint is set, and `false` otherwise. You will then schedule the task if it's `true`, and show a Toast to tell the user to set a constraint if it isn't.

2.2 Check for constraints

JobScheduler requires at least one constraint to be set. In this task you will create a boolean that will track if this requirement has been met, so that you can notify the user to set at least one constraint if they haven't already. As you create additional options in the further steps, you will need to modify this boolean so it is always `true` if at least one constraint is set, and `false` otherwise.

1. Create a boolean variable called `constraintSet` that is true if selected network option is not the default `JobInfo.NETWORK_TYPE_NONE`:

```
boolean constraintSet = selectedNetworkOption != JobInfo.NETWORK_TYPE_NONE;
```

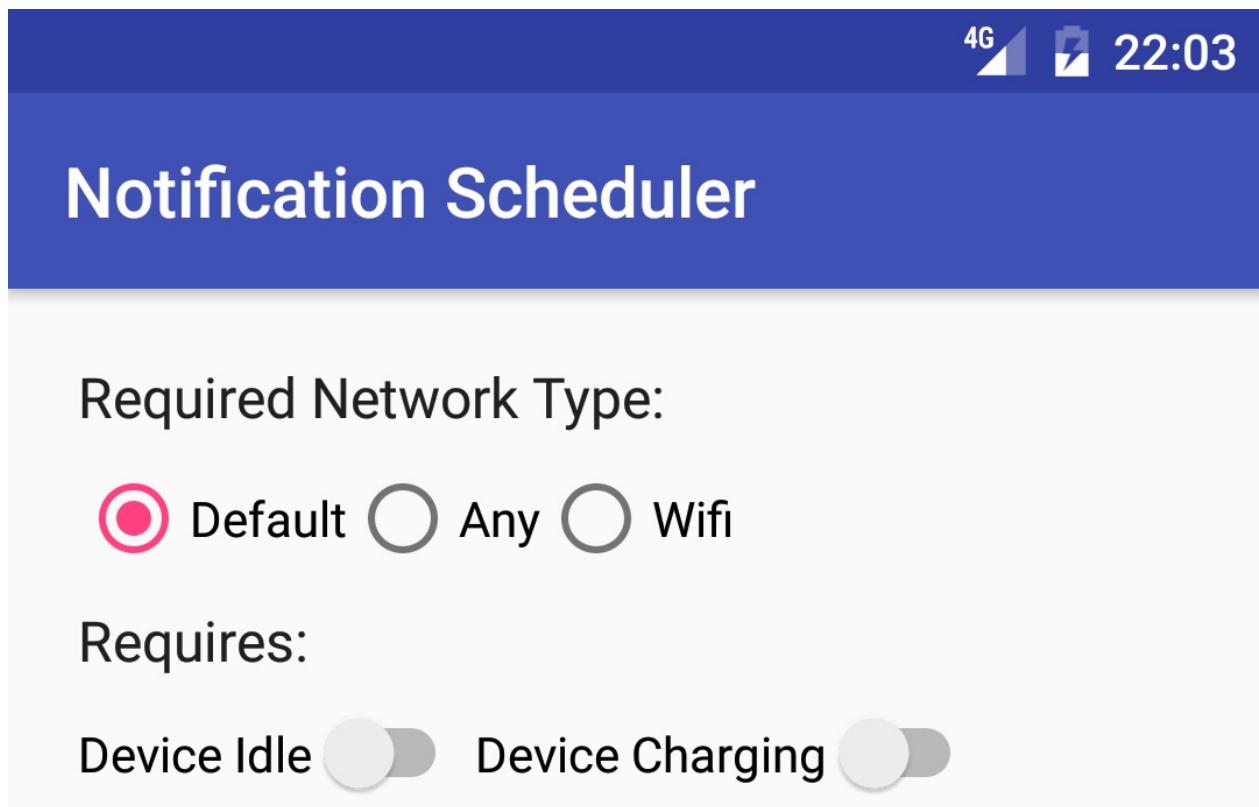
2. Create an if/else block using the `constraintSet` boolean.
3. Move the code that schedules the task and shows the Toast message into the `if` block.
4. If `constraintSet` is `false`, show a Toast message to the user to set at least one constraint. Don't forget to extract your string resources:

```
if(constraintSet) {
    //Schedule the job and notify the user
    JobInfo myJobInfo = builder.build();
    mScheduler.schedule(myJobInfo);
    Toast.makeText(this, R.string.job_scheduled, Toast.LENGTH_SHORT).show();
} else {
    Toast.makeText(this, R.string.no_constraint_toast, Toast.LENGTH_SHORT).show();
}
```

2.3 Implement the Device Idle and Device Charging constraints

JobScheduler includes the ability to wait until the device is charging, or in an idle state (the screen is off, and the CPU has gone to sleep) to execute your JobService. You will now add switches to your app to toggle these constraints on your JobService.

Add the UI elements for the new constraints



1. In your activity_main.xml file, copy the network type label TextView and paste it below the RadioGroup.
2. Change the `android:text` attribute to "Requires:".
3. Below this textView, insert a horizontal LinearLayout with a 4dp margin.
4. Create two Switch views as children to the horizontal LinearLayout with height and width set to "wrap_content" and the following attributes:

Switch 1	
android:text	"Device Idle"
android:id	"@+id/idleSwitch"
Switch 2	
android:text	"Device Charging"
android:id	"@+id/chargingSwitch"

Add the code for the new constraints

1. In MainActivity, create member variables, `mDeviceIdle` and `mDeviceCharging`, for the switches and initialize them in `onCreate()`.
2. In the `scheduleJob()` method, add the following calls to set the constraints on the JobScheduler based on the user selection in the switches:

```
builder.setRequiresDeviceIdle(mDeviceIdle.isChecked());
builder.setRequiresCharging(mDeviceCharging.isChecked());
```

3. Update the code that sets `constraintSet` to consider these new constraints:

```
boolean constraintSet = (selectedNetworkOption != JobInfo.NETWORK_TYPE_NONE)
    || mDeviceChargingSwitch.isChecked() || mDeviceIdleSwitch.isChecked();
```

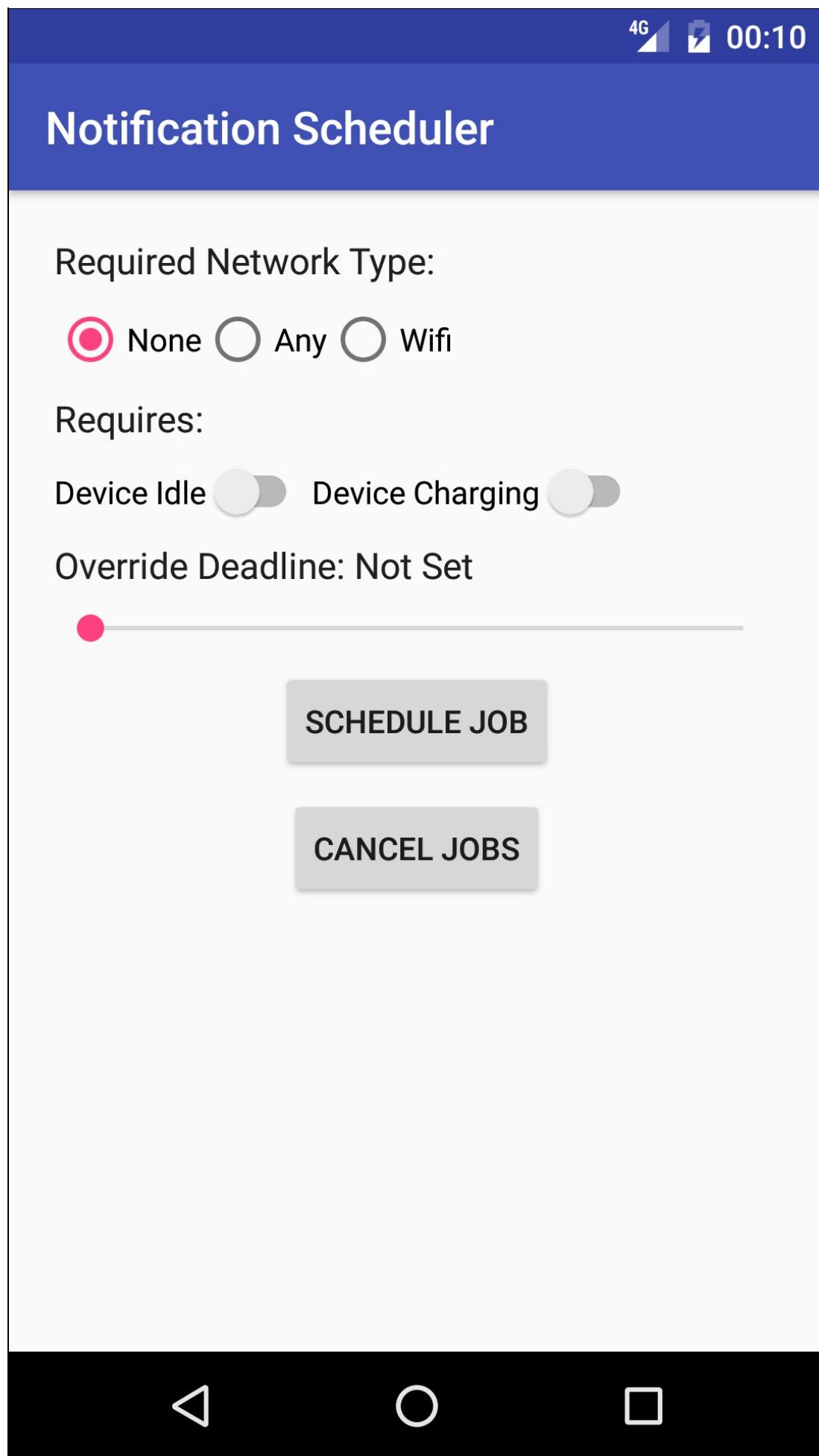
4. Run your app, now with the additional constraints. Try the difference combinations of switches to see when the notification gets sent (that indicates that the job ran). You can test the charging state constraint in an emulator by opening the menu (the ellipses icon next to the emulated device), go to the **Battery** pane and toggle the Battery Status dropdown. There is no way to manually put the emulator in Idle mode as of the writing of this practical.

Waiting until the device is idle and plugged in is a common pattern for battery intensive tasks such as downloading or uploading large files.

2.4 Implement the Override Deadline constraint

Up to this point, there is no way to know precisely when the framework will execute your task. The system takes into account effective resource management which may delay your task depending on the state of the device, and does not guarantee that your task will run on time. For example, a news app may want to download the latest news only when wifi is available and the device is plugged in and charging; but a user may inadvertently forget to enable their wifi or charge their device. If you don't add a time parameter to your scheduled Job, that user 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.

Add the new UI for setting the deadline to run the task



In this step you will use a new UI component, a [SeekBar](#), to allow the user to set a deadline between 0 and 100 seconds to execute your task.

The user sets the value by dragging the SeekBar.

1. Create a horizontal LinearLayout below the existing LinearLayout with the switches, which will contain the labels for the SeekBar.
2. The SeekBar will have two labels: a static one just like the label for the RadioGroup of buttons, and a dynamic one that will be updated with the value from the SeekBar. Add two TextViews to the LinearLayout with the following attributes:

TextView 1	
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:text	"Override Deadline: "
android:id	"@+id/seekBarLabel"
android:textAppearance	"@style/TextAppearance.AppCompat.Subhead"
TextView 2	
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
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_width	"match_parent"
android:layout_height	"wrap_content"
android:id	"@+id/seekBar"
android:layout_margin	"4dp"

Write the code for adding the deadline

1. In MainActivity, create a member variable for the SeekBar and initialize it in

```
onCreate() :  
    mSeekBar = (SeekBar) findViewById(R.id.seekBar);
```

2. Create final variables for both TextViews (they will be accessed from an inner class) and initialize them in `onCreate()` :

```
final TextView label = (TextView) findViewById(R.id.seekBarLabel);  
final TextView seekBarProgress = (TextView) findViewById(R.id.seekBarProgress);
```

3. In `onCreate()`, call `setOnSeekBarChangeListener()` on the SeekBar, passing in a new `OnSeekBarChangeListener` (Android Studio should generate the required methods):

```
mSeekBar.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) {}  
});
```

4. The second argument of `onProgressChanged()` is the current value of the SeekBar. In the `onProgresschanged()` callback, check if the integer value is greater than 0 (meaning a value has been set by the user), and if it is, set the SeekBar progress label to the integer value, followed by "s" to indicate seconds:

```
if (i > 0){  
    mSeekBarProgress.setText(String.valueOf(i) + " s");  
}
```

5. Otherwise, set the TextView to read "Not Set":

```
else {
    mSeekBarProgress.setText("Not Set");
}
```

6. The override deadline should only be set if the integer value of the SeekBar is greater than 0. In the `scheduleJob()` method, create an integer to store the SeekBar progress and a boolean variable that is true if the SeekBar has an integer value greater than 0:

```
int seekBarInteger = mSeekBar.getProgress();
boolean seekBarSet = seekBarInteger > 0;
```

7. 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 (seekBarSet) {
    builder.setOverrideDeadline(seekBarInteger * 1000);
}
```

8. Modify the `constraintSet` boolean to include the value of `seekBarSet` as a possible constraint:

```
boolean constraintSet = selectedNetworkOption != JobInfo.NETWORK_TYPE_NONE
    || mDeviceChargingSwitch.isChecked() || mDeviceIdleSwitch.isChecked()
    || seekBarSet;
```

9. Run the app. The user can now set a hard deadline in seconds by which time the JobService must be run!

2.5 Implement the Periodic constraint

JobScheduler also allows you to schedule a repeated task, much like AlarmManager. This option has a few caveats:

- The task is not guaranteed to run in a given period (the other conditions may not be met, or there might not be enough system resources).
- Using this constraint prevents you from also setting an override deadline or a minimum latency (), since these options do not make sense for repetitive tasks. See [JobInfo.Builder](#) documentation for more information.

Add the Periodic Switch to the layout

You will add a `Switch` to allow the user to switch between having the job run once or repeatedly at periodic intervals.

1. In `activity_main.xml`, add a `Switch` view between the two horizontal `LinearLayouts`. Use the following attributes:

Attribute	Value
<code>android:layout_width</code>	<code>"wrap_content"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:text</code>	<code>"Periodic"</code>
<code>android:id</code>	<code>"@+id/periodicSwitch"</code>
<code>android:layout_margin</code>	<code>"4dp"</code>

2. Create a member variable for the switch and initialize it in `onCreate()`:

```
mPeriodicSwitch = (Switch) findViewById(R.id.periodicSwitch);
```

Write the code to use the Periodic Switch

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.

1. Call `setOnCheckedChangeListener()` on the periodic switch, passing in a new `OnCheckedChangeListener`.
2. If checked, set the label to "Periodic Interval: ", otherwise to "Override Deadline: ":

```
mPeriodicSwitch.setOnCheckedChangeListener(
    new CompoundButton.OnCheckedChangeListener() {
        @Override
        public void onCheckedChanged(CompoundButton compoundButton, boolean isChecked)
        {
            if (isChecked){
                label.setText(R.string.periodic_interval);
            } else {
                label.setText(R.string.override_deadline);
            }
        }
    });
});
```

All that remains now is to implement the logic in the `scheduleJob()` method to properly set the constraints on the `JobInfo` object.

If the periodic option is **on**:

- If the SeekBar has a **non-zero value**, set the constraint by calling `setPeriodic()` on the `JobInfo.Builder` object.
- If the SeekBar has a **value of 0**, show a Toast message asking the user to set a periodic interval with the SeekBar.

If the periodic option is **off**:

- If the SeekBar has a **non-zero value**, the user has set an override deadline. Apply the override deadline using the `setOverrideDeadline()` option.
- If the SeekBar has a **value of 0**, the user has simply not specified an override deadline or a periodic task, so add nothing to the `JobInfo.Builder` object.
- Replace the code that sets the override deadline to the `JobInfo.Builder` in `scheduleJob()` with the following code to implement this logic:

```
if (mPeriodicSwitch.isChecked()) {
    if (seekBarSet) {
        builder.setPeriodic(seekBarInteger * 1000);
    } else {
        Toast.makeText(MainActivity.this,
            "Please set a periodic interval", Toast.LENGTH_SHORT).show();
    }
} else {
    if (seekBarSet) {
        builder.setOverrideDeadline(seekBarInteger * 1000);
    }
}
```

Solution code

Android Studio project: [NotificationScheduler](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: Up until now, your tasks scheduled by the `JobService` focused on delivering 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. Since background tasks can be more complex in nature, both from a programmatic and from a functionality standpoint, the job of notifying the framework when the task is complete falls on the developer. Fortunately, the developer can do this by calling `jobFinished()`.

1. Implement a JobService that starts an AsyncTask when the given constraints are met. The AsyncTask should sleep 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 also reschedule the job.

Summary

- JobScheduler provides a flexible framework to intelligently accomplish background services.
- JobScheduler is only available on devices running API 21+.
- To use the JobScheduler, you need two parts: JobService and JobInfo.
- JobInfo is a set of conditions that will trigger the job to run.
- JobService implements the job to run under the conditions specified by JobInfo.
- You only have to implement the `onStartJob()` and `onStopJob()` callback methods in your JobService.
- The implementation of your job occurs (or is started) in `onStartJob()`.
- `onStartJob()` returns a boolean that indicates whether the service needs to process the work in a separate thread.
- If `onStartJob()` returns true, you must explicitly call `jobFinished()`. If `onStartJob()` returns false, the runtime will call `jobFinished()` on your behalf.
- JobService is processed on the main thread, so avoid lengthy calculations or I/O.
- JobScheduler is the manager class responsible for scheduling the task. JobScheduler batches tasks together to maximize the efficiency of system resources, which means you do not have exact control of when it will be executed.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Transferring Data Efficiently](#)

Learn more

Android Developer Documentation

Reference

- [JobScheduler](#)
- [JobInfo](#)

- [JobInfo.Builder](#)
- [JobService](#)
- [JobParameters](#)

9.1: 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](#)
- [Coding challenge](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

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 Android 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.

The data you save to shared preferences is different from the data in the saved activity state you learned about in an earlier chapter. The data in the activity instance state is retained across activity instances in the same user session. Shared preferences persist across user sessions, even if your app is killed and restarted or if the device is rebooted.

Use shared preferences only when you need to save a small amount data as simple key/value pairs. 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

You will learn to:

- Identify what shared preferences are.
- Create a shared preferences file for your app.
- Save data to shared preferences, and read those preferences back again.
- Clear the data in the shared preferences.

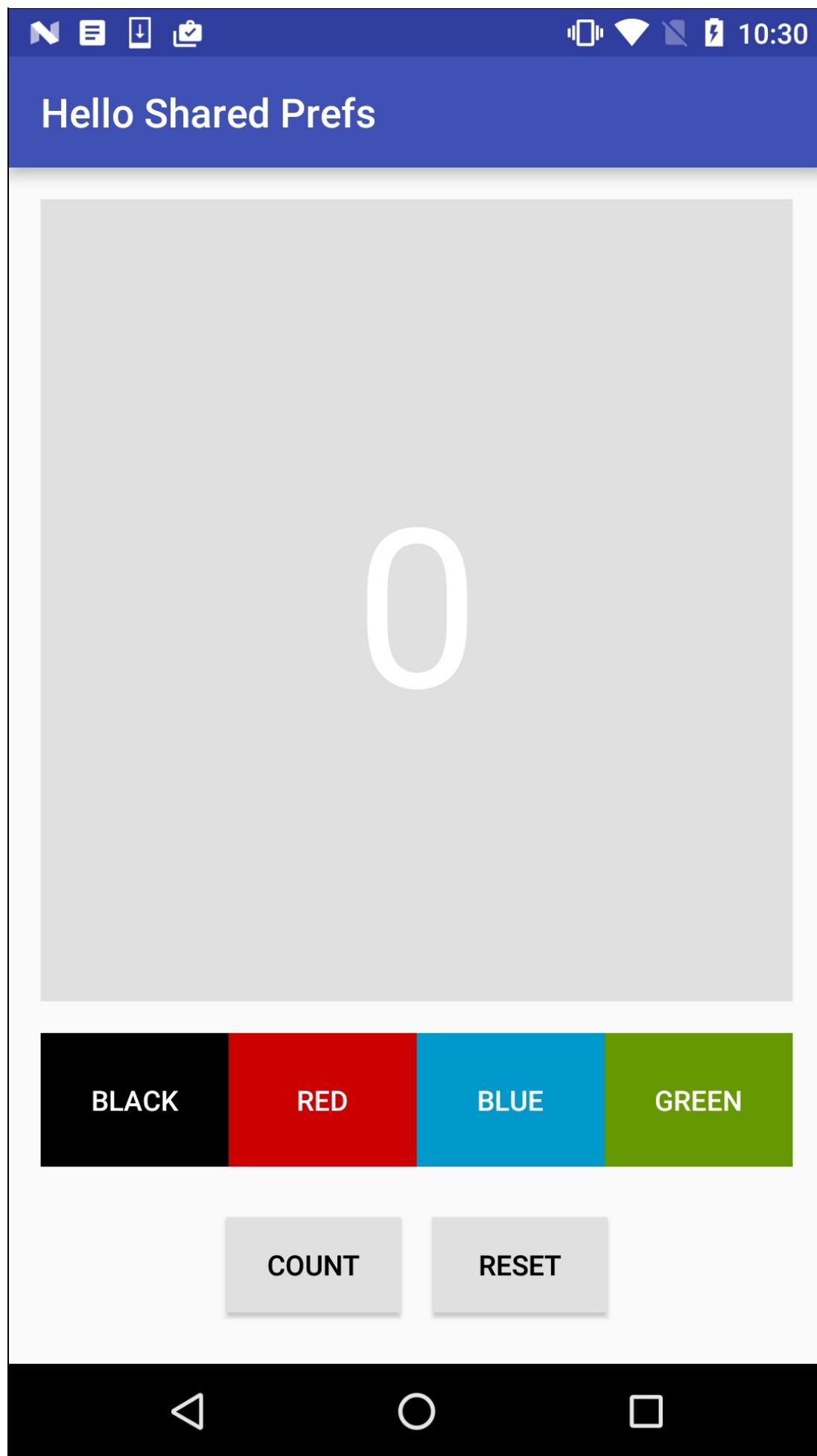
What you will DO

In this practical, you will:

- Add the ability to save, retrieve, and reset shared preferences to an app.

App Overview

The HelloSharedPrefs app is another variation of the HelloToast app you created in Lesson 1. It includes buttons to increment the number, to change the background color, and to reset both the number and color to their defaults. The app also uses themes and styles to define the buttons.



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 this practical is available at [HelloSharedPrefs-Start](#). 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-Start](#) 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.
 - Click the Reset button to set the color and count back to the defaults.
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 (`mColor`) 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()`. The bundle keys for count and color are defined by private variables (`COUNT_KEY`) and (`COLOR_KEY`).

Task 2. Save and restore data to a shared preferences file

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. Because the state data you're saving to the shared preferences (the current count and color) are the **same** data you preserve in the instance state, you don't have to do it twice -- you can replace the instance state altogether with the shared preference state.

2.1 Initialize the preferences

1. 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.

2. 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 to a Bundle object as a key/value pair. For shared preferences, however, you save that data in the onPause() lifecycle callback, and you need a shared 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_KEY, mCount);
preferencesEditor.putInt(COLOR_KEY, mCurrentColor);
```

The SharedPreferences.Editor class includes multiple put methods for different data types, including.putInt() and putString().

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.

7. Delete the entire onSaveInstanceState() method. Since the activity instance state contains the same data as the shared preferences, you can replace the instance state altogether.

Solution Code (MainActivity - onPause() method)

```
@Override  
protected void onPause(){  
    super.onPause();  
  
    SharedPreferences.Editor preferencesEditor = mPreferences.edit();  
    preferencesEditor.putInt(COUNT_KEY, mCount);  
    preferencesEditor.putInt(COLOR_KEY, mColor);  
    preferencesEditor.apply();  
}
```

2.3 Restore preferences in onCreate()

As with the instance state, your app reads any saved shared preferences in the onCreate() method. Again, since the shared preferences contain the same data as the instance state, we can replace the state with the preferences here as well. Every time onCreate() is called -- when the app starts, on configuration changes -- the shared preferences are used to restore the state of the view.

1. Locate the part of the onCreate() method that tests if the savedInstanceState argument is null and restores the instance state:

```
if (savedInstanceState != null) {  
    mCount = savedInstanceState.getInt(COUNT_KEY);  
    if (mCount != 0) {  
        mShowCountTextView.setText(String.format("%s", mCount));  
    }  
    mColor = savedInstanceState.getInt(COLOR_KEY);  
    mShowCountTextView.setBackgroundColor(mColor);  
}
```

2. Delete that entire block.
3. In the `onCreate()` method, in the same spot where the save instance state code was, get the count from the preferences with the `COUNT_KEY` key and assign it to the `mCount` variable.

```
mCount = mPreferences.getInt(COUNT_KEY, 0);
```

When you read data from the preferences you don't need to get a shared preferences editor. Use any of the `get` methods on a shared preferences object to retrieve preference data.

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. In this case the default value is 0, which is the same as the initial value of `mCount`.

4. Update the value of the main text view with the new count.

```
mShowCountTextView.setText(String.format("%s", mCount));
```

5. Get the color from the preferences with the `COLOR_KEY` key and assign it to the `mColor` variable.

```
mColor = mPreferences.getInt(COLOR_KEY, mColor);
```

As before, the second argument to `getInt()` is the default value to use in case the key doesn't exist in the shared preferences. In this case you can just reuse the value of `mColor`, which was just initialized to the default background further up in the method.

6. Update the background color of the main text view.

```
mShowCountTextView.setBackgroundColor(mColor);
```

7. Run the app. Click the count button and change the background color to update the instance state and the preferences.

8. Rotate the device or emulator to verify that the count and color are saved across configuration changes.
9. 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.
10. 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);  
  
    // Initialize views, color, preferences  
    mShowCountTextView = (TextView) findViewById(R.id.count_textview);  
    mColor = ContextCompat.getColor(this, R.color.default_background);  
    mPreferences = getSharedPreferences(mSharedPrefFile, MODE_PRIVATE);  
  
    // Restore preferences  
    mCount = mPreferences.getInt(COUNT_KEY, 0);  
    mShowCountTextView.setText(String.format("%s", mCount));  
    mColor = mPreferences.getInt(COLOR_KEY, mColor);  
    mShowCountTextView.setBackgroundColor(mColor);  
}
```

2.4 Reset preferences in the reset() click handler

The reset button in the starter app resets both the count and color for the activity to their default values. Since the preferences hold the state of the activity, it's important to also clear the preferences at the same time.

1. In the reset() click handler method, after the color and count are reset, get an editor for the SharedPreferences object:

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
```

2. Delete all the shared preferences:

```
preferencesEditor.clear();
```

3. Apply the changes:

```
preferencesEditor.apply();
```

Solution Code (reset() method) :

```
public void reset(View view) {  
    // Reset count  
    mCount = 0;  
    mShowCountTextView.setText(String.format("%s", mCount));  
  
    // Reset color  
    mColor = ContextCompat.getColor(this, R.color.default_background);  
    mShowCountTextView.setBackgroundColor(mColor);  
  
    // Clear preferences  
    SharedPreferences.Editor preferencesEditor = mPreferences.edit();  
    preferencesEditor.clear();  
    preferencesEditor.apply();  
}
```

Solution code

Android Studio project: [HelloSharedPrefs](#)

Coding challenge

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

Challenge: Modify the HelloSharedPrefs app so that instead of automatically saving the state to the preferences file, add a second activity to change, reset, and save those preferences. Add a button to the app named Settings to launch that activity. Include toggle buttons and spinners to modify the preferences, and Save and Reset buttons for saving and clearing the preferences.

Summary

- The [SharedPreferences](#) class allows an app to store small amounts of primitive data as [key-value pairs](#).
- Shared preferences persist across different user sessions of the same app.

- To write to the shared preferences, get a `SharedPreferences.Editor` object.
- Use the various `put*` methods in a `SharedPreferences.Editor` object, such as `putInt()` or `putString()`, to put data into the shared preferences with a key and a value.
- Use the various `get*` methods in a `SharedPreferences` object, such as `getInt()` or `getString()`, to get data out of the shared preferences with a key.
- Use the `clear()` method in a `SharedPreferences.Editor` object to remove all the data stored in the preferences.
- Use the `apply()` method in a `SharedPreferences.Editor` object to save the changes to the preferences file.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Shared Preferences](#)

Learn more

- [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)

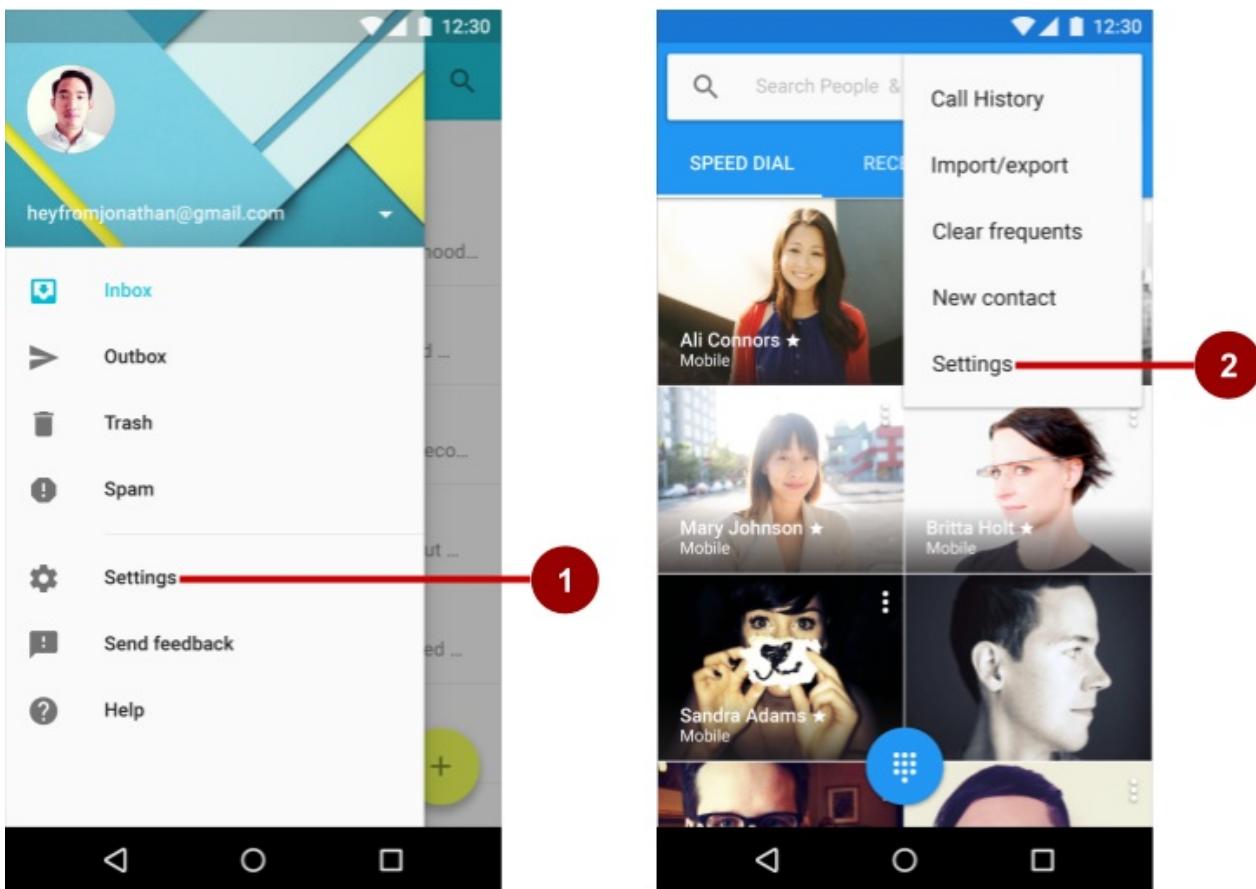
9.2: Adding Settings to an App

Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1: Add a switch setting to an app](#)
- [Task 2: Using the Settings Activity template](#)
- [Coding challenges](#)
- [Summary](#)
- [Related Concept](#)
- [Learn more](#)

Apps often include settings that allow users to modify app features and behaviors. For example, some apps allow users to set their home locations, default units for measurements, dining options, and other settings that apply to the entire app. Settings are usually accessed infrequently, because once a user changes a setting, such as a home location, they rarely need to go back and change it again.

Users expect to navigate to app settings by tapping **Settings** in side navigation, such as a navigation drawer as shown on the left side of the figure below, or in the options menu in the app bar, shown on the right side of the figure below.



In the figure above:

1. **Settings** in side navigation (a navigation drawer)
2. **Settings** in the options menu of the app bar

In this practical you will add a settings activity to an app. Users will be able to navigate to the app settings by tapping **Settings**, which will be located in the options menu in the app bar.

What you should already KNOW

From the previous practicals, you should be able to:

- Add an activity to an app.
- Design layouts with buttons and text views.
- Extract string resources and edit string and string array values.
- Create an options menu in the app bar.
- Add and edit the menu items in the options menu.
- Add the event handler for menu item clicks.
- Edit the `AndroidManifest.xml` file to add Up navigation for a second activity.
- Read preferences from `sharedPreferences`.

What you will LEARN

You will learn to:

- Add an activity and understand the use of fragments for managing settings.
- Create an XML resource file of settings with their attributes.
- Create navigation to the settings activity.
- Set the default values of settings.
- Read the settings values changed by the user.
- Customize the Settings Activity template for your own use.

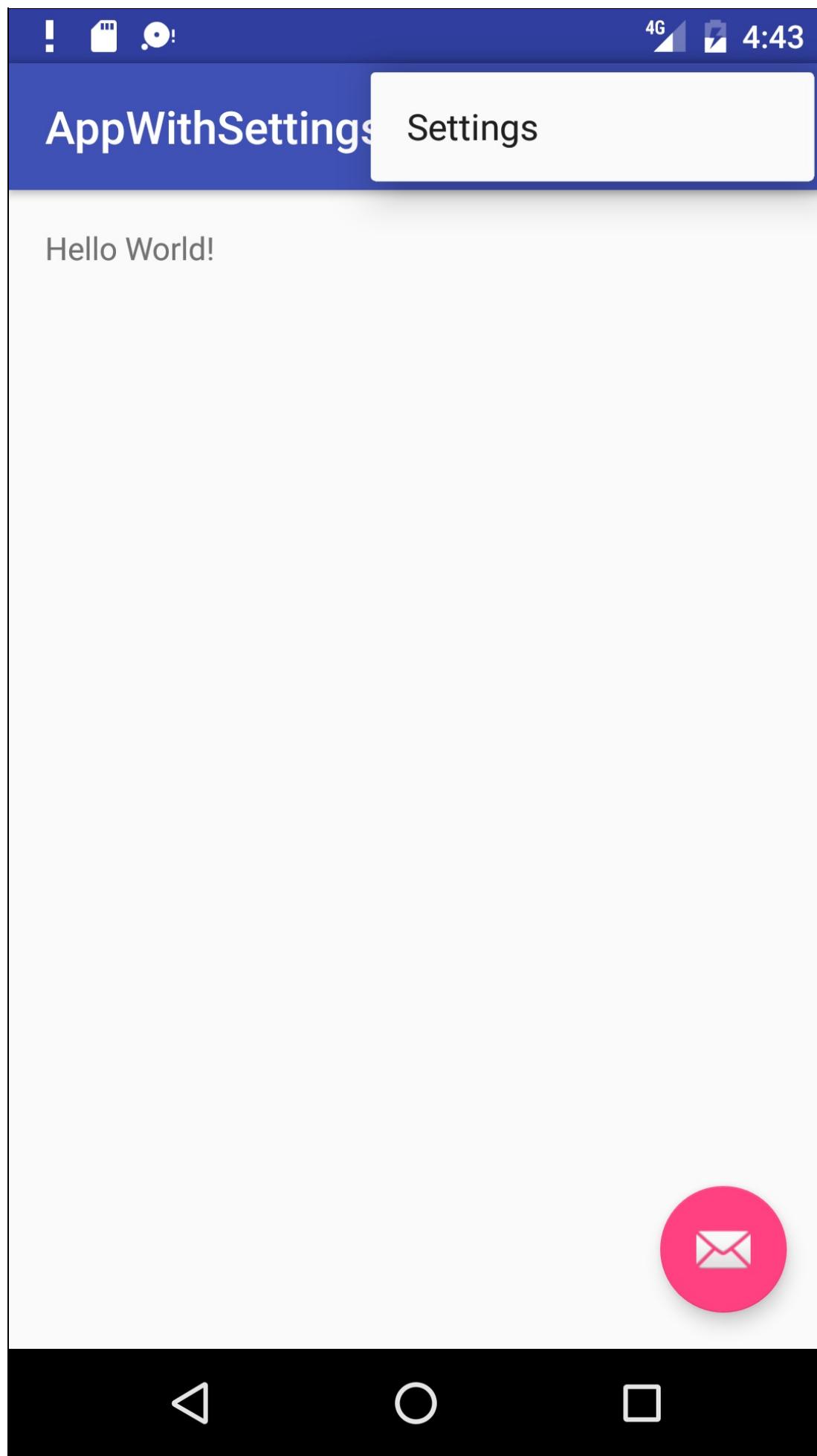
What you will DO

In this practical, you will:

- Create an app that includes **Settings** in the options menu.
- Add "Settings option" as a toggle switch.
- Add code to set the default value for the setting, and access the setting value after it has changed.
- Use and customize the Android Studio Settings Activity template.

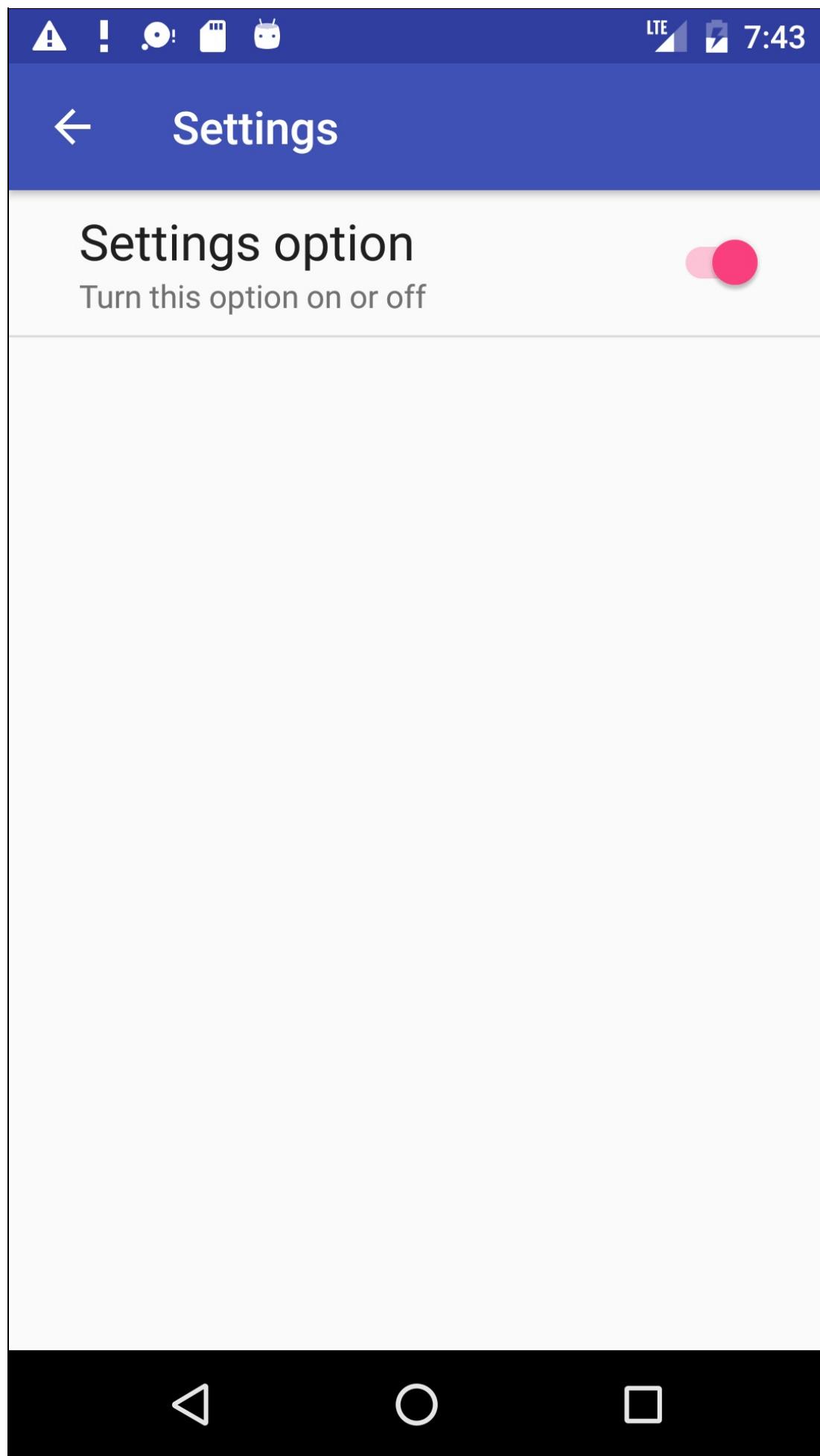
App overview

Android Studio provides a shortcut for setting up an options menu with **Settings**. If you start an Android Studio project for a smartphone or tablet using the Basic Activity template, the new app includes **Settings** as shown below:



The template also includes a floating action button in the lower right corner of the screen with an envelope icon. You can ignore this button for this practical, as you won't be using it.

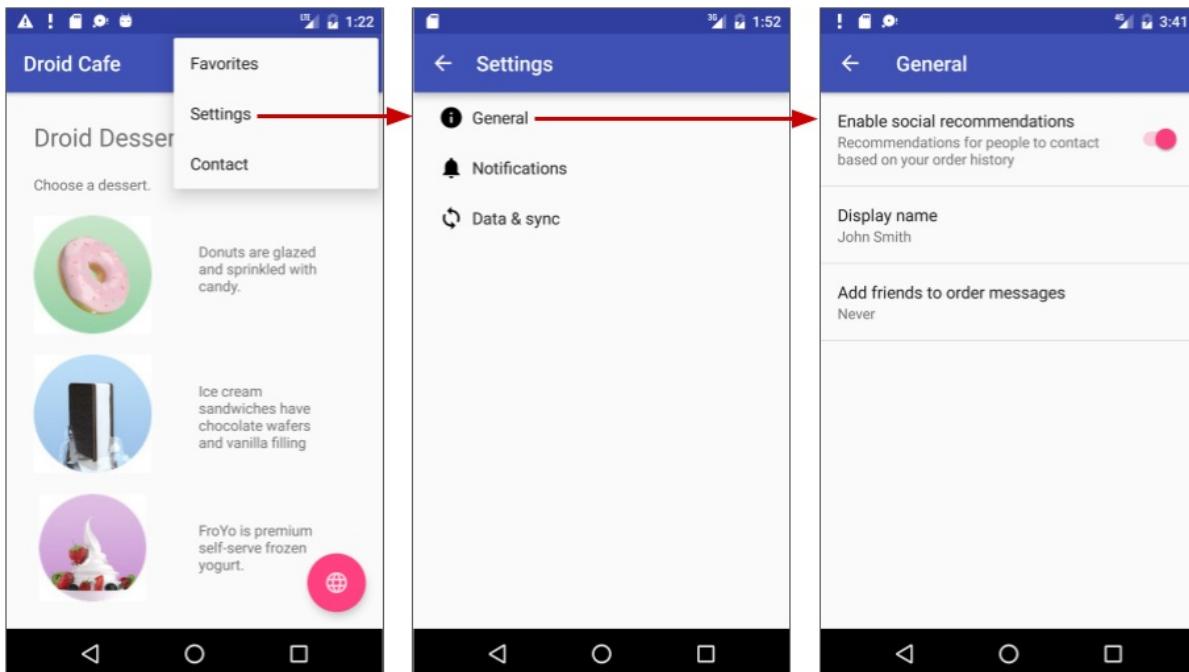
You'll start by creating an app named AppWithSettings using the Basic Activity template, and add a settings activity that provides one toggle switch setting that the user can turn on or off:



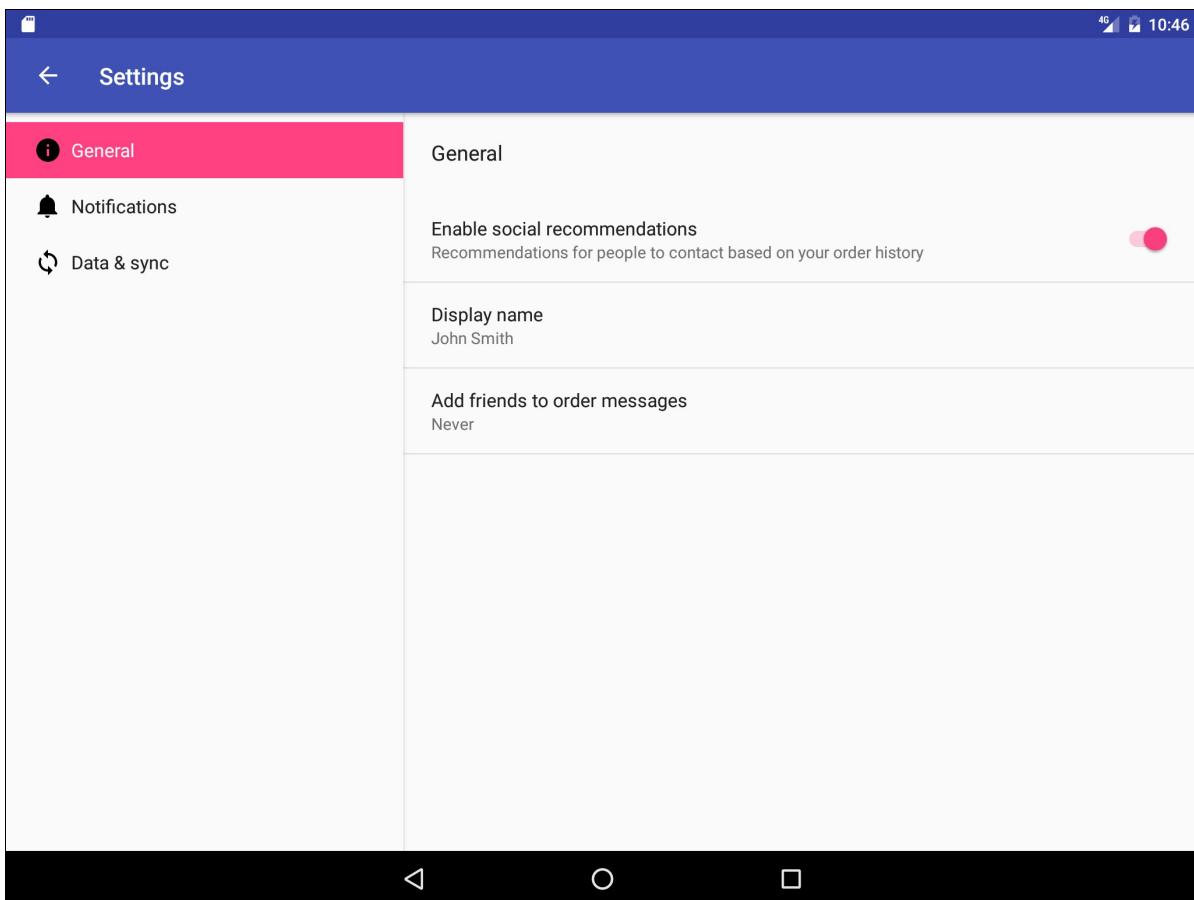
You will add code to read the setting and perform an action based on its value. For the sake of simplicity, the action will be to display a toast message with the value of the setting.

In the second task, you will add the standard Settings Activity template provided by Android Studio to the DroidCafe app you created in a previous lesson. The Settings Activity template is pre-populated with settings you can customize for an app, and provides a different layout for smartphones and tablets:

- *Smartphones*: A main Settings screen with a header link for each group of settings, such as General for general settings, as shown below.



- *Tablets*: A master/detail screen layout with a header link for each group on the left (master) side, and the group of settings on the right (detail) side, as shown in the figure below.



All you need to do to customize the template is change the headers, setting titles, setting descriptions, and values for the settings, and write the code you would normally write to use the values of the settings.

The Droid Cafe app was created in a previous lesson from the Basic Activity template, which provides an options menu in the app bar for placing the **Settings** option. You will customize the supplied Settings Activity template by changing a single setting's title, description, values, and default values. You will add code to read the setting's value after the user changes it, and display that value.

Task 1: Add a switch setting to an app

In this task, you will:

- Create a new project based on the Basic Activity template (which provides an options menu).
- Add a toggle switch setting with attributes in a preference XML file.
- Add an activity for settings and a fragment for a specific setting. You will use the `PreferenceFragmentCompat` version of `PreferenceFragment` in order to maintain compatibility with `AppCompatActivity`. You will also add the `android.support:preference-v7` library.

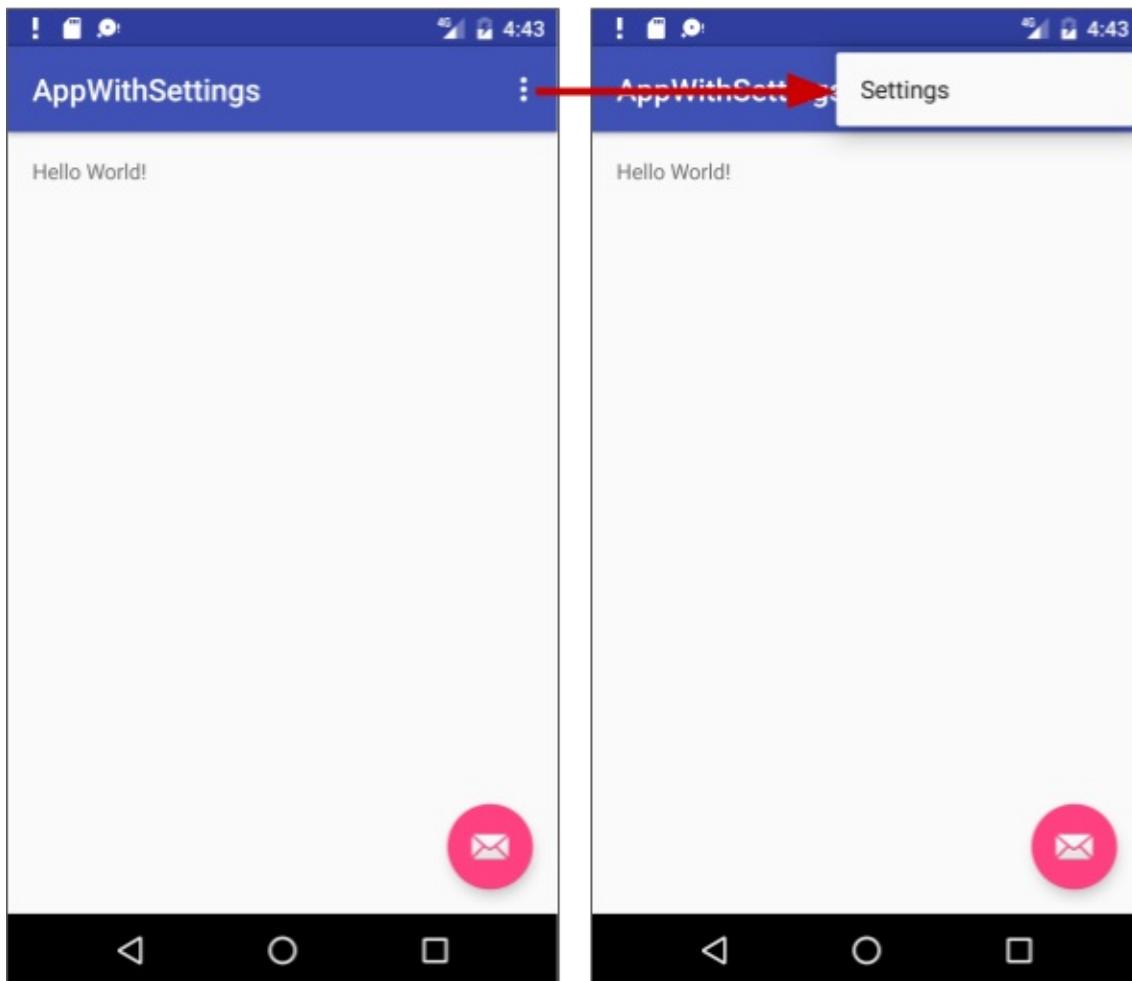
- Connect the **Settings** item in the options menu to the settings activity.

1.1 Create the project and add the xml directory and resource file

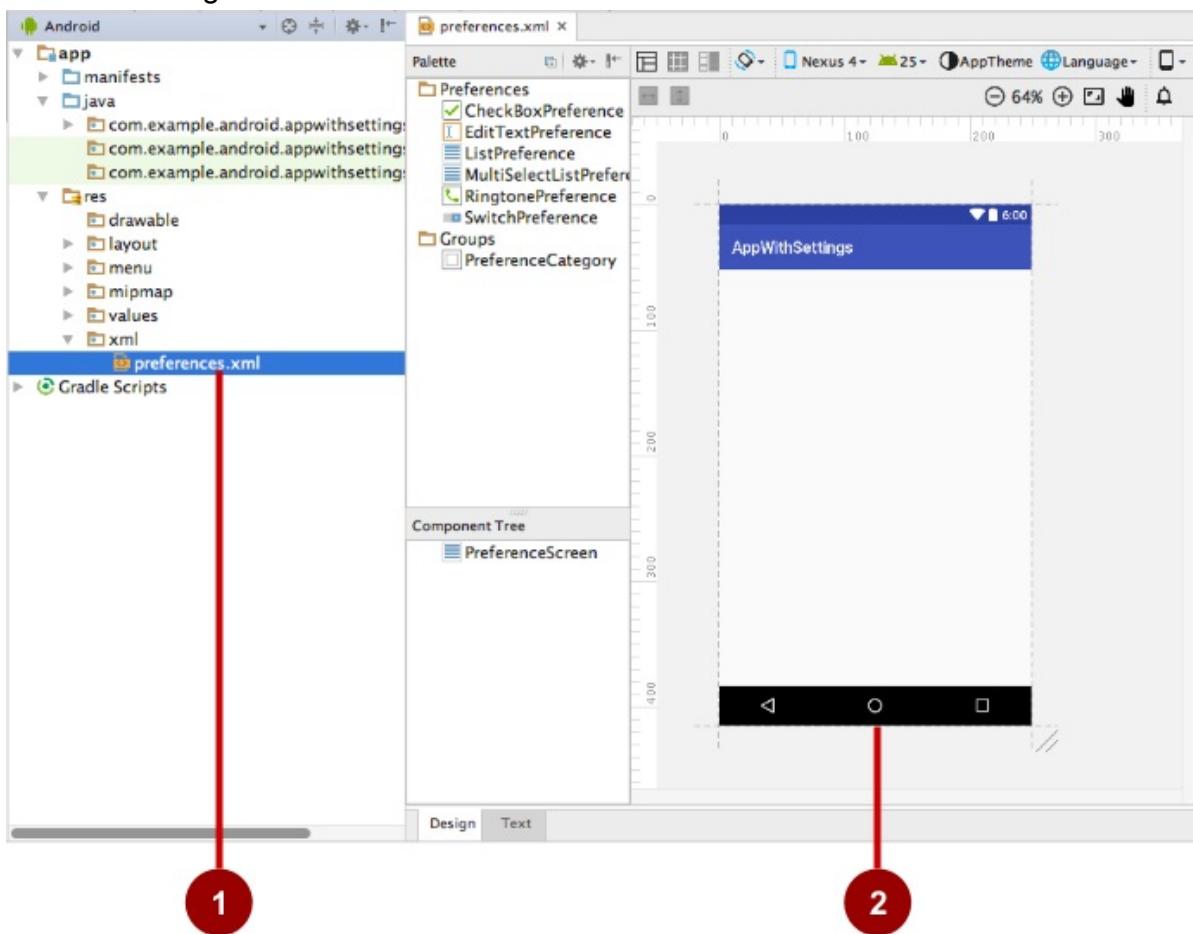
1. In Android Studio, create a new project with the following parameters:

Attribute	Value
Application Name	AppWithSettings
Company Name	android.example.com (or your own domain)
Phone and Tablet Minimum SDK	API15: Android 4.0.3 IceCreamSandwich
Use a Fragment?	Leave unchecked
Template	Basic Activity

2. Run the app, and tap the overflow icon in the app bar to see the options menu, as shown in the figure below. The only item in the options menu is **Settings**.



3. Create a new resource directory to hold the XML file containing the settings:
 - i. Select the **res** directory in the Project: Android view, and choose **File > New > Android Resource Directory**. The New Resource Directory dialog appears.
 - ii. In the Resource type drop-down menu, choose **xml**. The Directory name automatically changes to `xml`.
 - iii. Click **OK**.
4. The **xml** directory appears in the Project: Android view inside the **res** directory. Select the **xml** directory and choose **File > New > XML resource file** (or right-click the **xml** directory and choose **New > XML resource file**).
5. Enter the name of the XML file, **preferences**, in the File name field, and click **OK**. The **preferences.xml** file appears inside the **xml** directory, and the layout editor appears, as shown in the figure below.

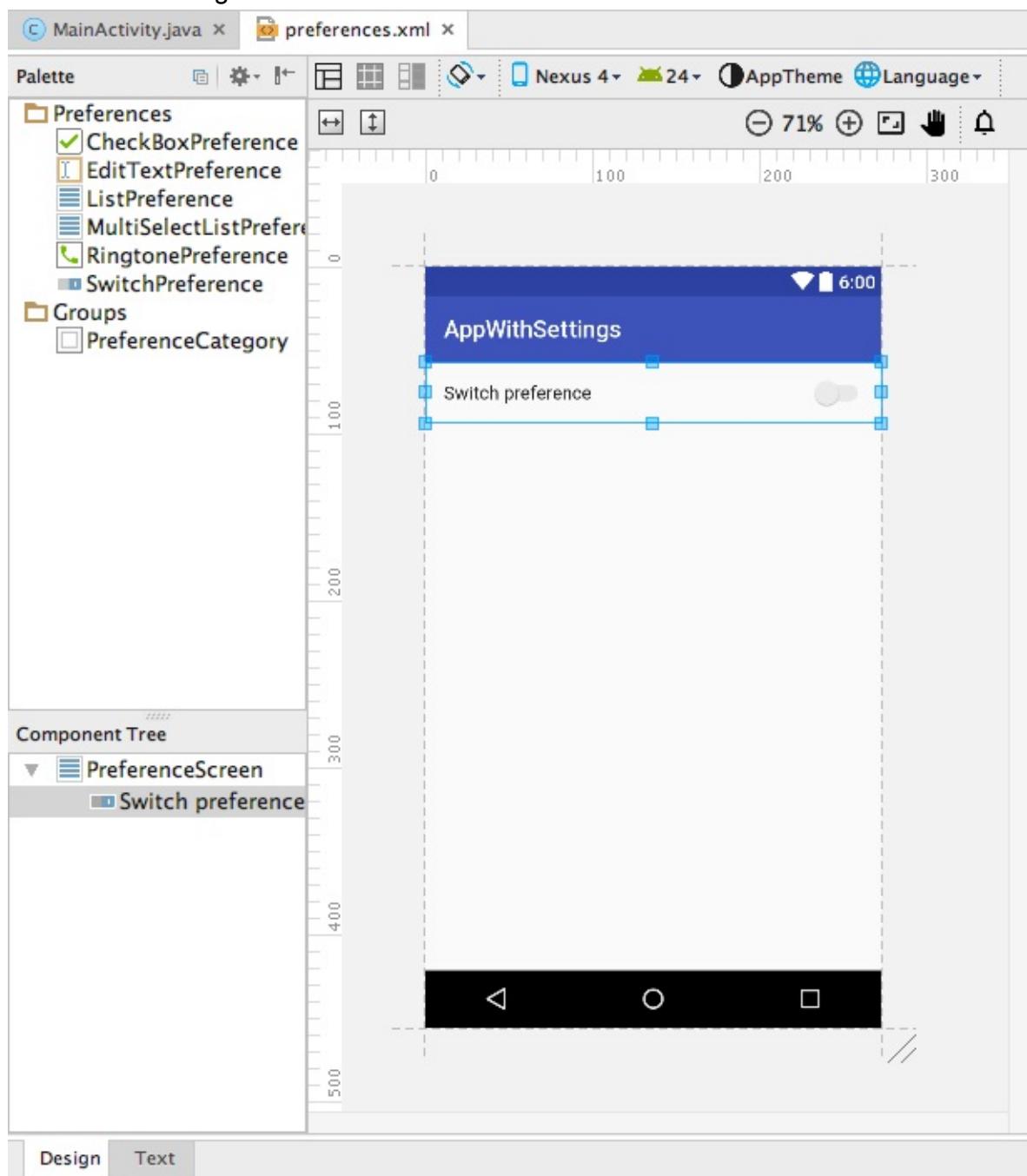


In the figure above:

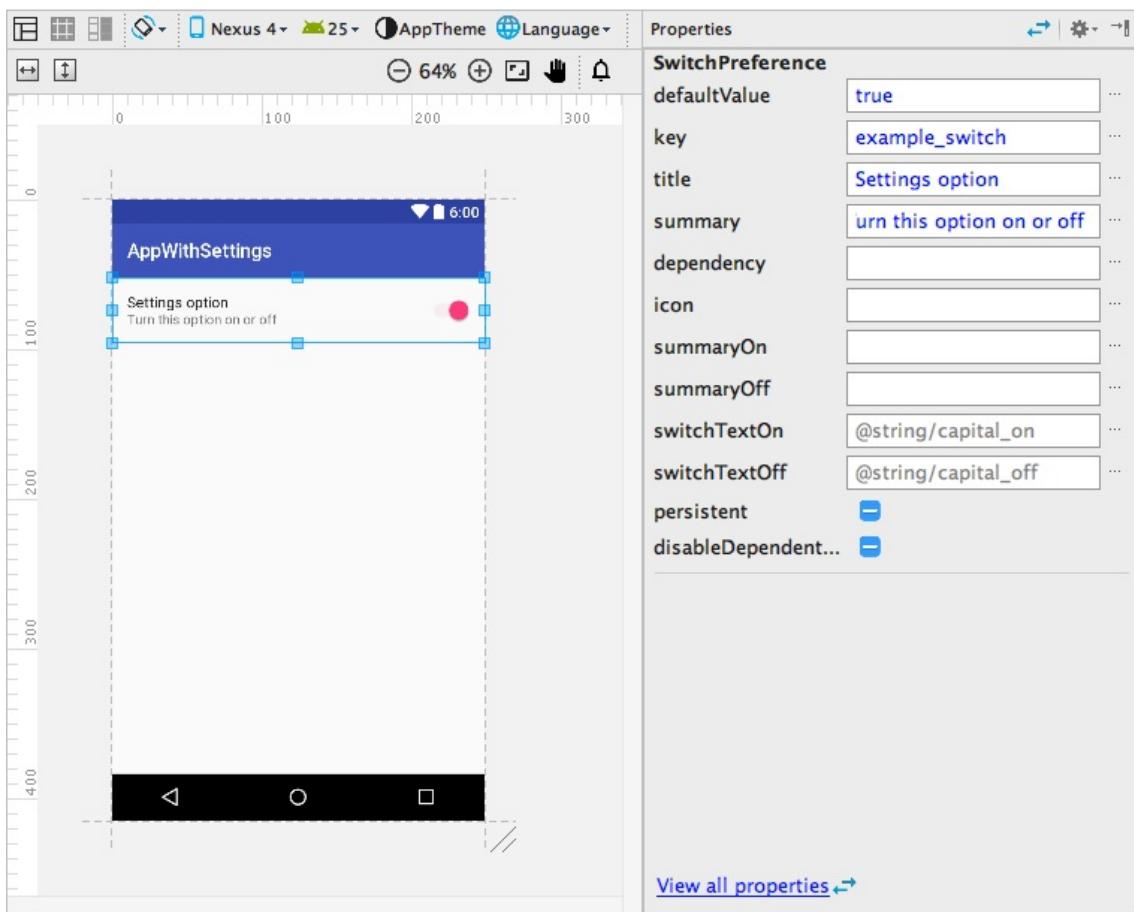
1. The **preferences.xml** file inside the **xml** directory.
2. The layout editor showing the **preferences.xml** contents.

1.2 Add the XML preference and attributes for the setting

1. Drag a **SwitchPreference** from the Palette pane on the left side to the top of the layout, as shown in the figure below.



2. Change the values in the Properties pane on the right side of the layout editor as follows (refer to the figure below):
 - i. **defaultValue: true**
 - ii. **key: example_switch**
 - iii. **title: Settings option**
 - iv. **summary: Turn this option on or off**



- Click the Text tab at the bottom of the layout editor to edit the XML code:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <SwitchPreference
        android:defaultValue="true"
        android:title="Settings option"
        android:key="example_switch"
        android:summary="Turn this option on or off" />
</PreferenceScreen>
```

The Properties values you entered represent XML attributes:

- `android:defaultValue` : The default value of the setting when the app starts for the first time.
- `android:title` : The title of the setting. For a `SwitchPreference`, the title appears to the left of the toggle switch.
- `android:key` : The key to use for storing the setting value. Each setting has a corresponding key-value pair that the system uses to save the setting in a default `SharedPreferences` file for your app's settings.
- `android:summary` : The text summary appears underneath the setting.

- Extract the string resources for the `android:title` and `android:summary` attribute values to `@string/switch_title` and `@string/switch_summary`.

5. Change `<SwitchPreference` in the code to

```
<android.support.v7.preference.SwitchPreferenceCompat :>

<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <android.support.v7.preference.SwitchPreferenceCompat
        ...
    </PreferenceScreen>
```

In order to use the `PreferenceFragmentCompat` version of `PreferenceFragment`, you must also use the `android.support.v7` version of `SwitchPreference` (`SwitchPreferenceCompat`).

The `SwitchPreferenceCompat` line above may show a yellow light bulb icon with a warning, but you can ignore it.

6. Open the **styles.xml** file, and add the following `preferenceTheme` declaration to the `AppTheme`:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    ...
    <item name="preferenceTheme">@style/PreferenceThemeOverlay</item>
</style>
```

In order to use the `PreferenceFragmentCompat` version of `PreferenceFragment`, you must also declare `preferenceTheme` with the `PreferenceThemeOverlay` style to the app theme.

7. Open the **build.gradle (Module: app)** file, and add the following to the `dependencies` section:

```
dependencies {
    ...
    compile 'com.android.support:preference-v7:25.0.1'
}
```

The above adds the `android.support:preference-v7` library in order to use the `PreferenceFragmentCompat` version of `PreferenceFragment`.

1.3 Add an activity for settings and a fragment for a specific setting

1. In order to create a Settings activity that provides a UI for settings, add an Empty Activity to the app:
 - i. Select **app** at the top of the Project: Android view.
 - ii. Choose **New > Activity > Empty Activity**.

- iii. Name the activity **SettingsActivity**.
- iv. Uncheck the option to generate a layout file (you don't need one).
- v. Leave checked the Backwards Compatibility (AppCompat) option.
- vi. The Package name should already be set to **com.example.android.projectname**, and the Target Source Set should be set to **main**. If not, make these selections in the drop-down menus.
- vii. Click **Finish**.

The result is the following class definition in SettingsActivity:

```
public class SettingsActivity extends AppCompatActivity {

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

2. Add a blank fragment for a group of similar settings (*without* a layout, factory methods, or interface callbacks) to the app, in order to swap them into the Settings activity screen when needed:
 - i. Select **app** at the top of the Project: Android view again.
 - ii. Choose **New > Fragment > Fragment (Blank)**.
 - iii. Name the fragment **SettingsFragment**.
 - iv. Uncheck the option to generate a layout file (you don't need one).
 - v. Uncheck the option to include fragment factory methods.
 - vi. Uncheck the option to include interface callbacks.
 - vii. Click **Finish**.

The result is the following class definition in SettingsFragment:

```
public class SettingsFragment extends Fragment {

    public SettingsFragment() {
        // Required empty public constructor
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
                           ViewGroup container, Bundle savedInstanceState) {
        TextView textView = new TextView(getActivity());
        textView.setText(R.string.hello_blank_fragment);
        return textView;
    }
}
```

3. Change the class definition of `SettingsFragment` to extend `PreferenceFragmentCompat`:

```
public class SettingsFragment extends PreferenceFragmentCompat {  
    ...  
}
```

You use a specialized [Fragment](#) subclass to display a list of settings. The best practice is to use a regular [Activity](#) that hosts a [PreferenceFragment](#) that displays the app settings. [Fragments](#) like `PreferenceFragment` provide a more flexible architecture for your app, compared to using activities alone. A fragment is like a modular section of an activity—it has its own lifecycle and receives its own input events, and you can add or remove a fragment while the activity is running.

Use the `PreferenceFragmentCompat` version of `PreferenceFragment` with an activity that extends `AppCompatActivity`. In order to extend the fragment, you may have to add the following import statement:

```
import android.support.v7.preference.PreferenceFragmentCompat;
```

4. Replace the entire `onCreateView()` method in the fragment with this `onCreate()` method:

```
@Override  
public void onCreatePreferences(Bundle savedInstanceState,  
    String rootKey) {  
}
```

The reason why you replace `onCreateView()` with `onCreatePreferences()` in `SettingsFragment` is because you will be adding this fragment to the existing `SettingsActivity` to display preferences, rather than showing a separate fragment screen. Adding it to the existing activity makes it easy to add or remove a fragment while the activity is running. The preference fragment is rooted at the `PreferenceScreen` using `rootKey`.

You can safely remove the empty constructor from the fragment as well, since the fragment is not displayed by itself:

```
public SettingsFragment() {  
    // Required empty public constructor  
}
```

5. At the end of the `onCreatePreferences()` method in `SettingsFragment`, you need to associate with this fragment the **preferences.xml** settings resource you just created. Add a call to `setPreferencesFromResource()` passing the id of the XML file (`R.xml.preferences`) and the `rootKey` to identify the preference root in `PreferenceScreen`:

```
setPreferencesFromResource(R.xml.preferences, rootKey);
```

The `onCreatePreferences()` method should now look like this:

```
@Override
public void onCreatePreferences(Bundle savedInstanceState,
                             String rootKey) {
    setPreferencesFromResource(R.xml.preferences, rootKey);
}
```

6. Add the following code to the end of the `SettingsActivity` `onCreate()` method so that the fragment is displayed as the main content:

```
getSupportFragmentManager().beginTransaction()
    .replace(android.R.id.content, new SettingsFragment())
    .commit();
```

The above code is the typical pattern used to add a fragment to an activity so that the fragment appears as the main content of the activity. You use:

- `getFragmentManager()` if the class extends `Activity` and the fragment extends `PreferenceFragment`.
- `getSupportFragmentManager()` if the class extends `AppCompatActivity` and the fragment extends `PreferenceFragmentCompat`.

The entire `onCreate()` method in `SettingsActivity` should now look like the following:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getSupportFragmentManager().beginTransaction()
        .replace(android.R.id.content, new SettingsFragment())
        .commit();
}
```

1.4 Connect the Settings menu item to the settings activity

Use an `intent` to launch the `SettingsActivity` from the `MainActivity`.

1. Find the `if` block in the `onOptionsItemSelected()` method in `MainActivity`, which handles the tap on **Settings** in the options menu:

```
if (id == R.id.action_settings) {  
    return true;  
}
```

2. Add an `intent` to the `if` block to launch the `SettingsActivity`:

```
if (id == R.id.action_settings) {  
    Intent intent = new Intent(this, SettingsActivity.class);  
    startActivity(intent);  
    return true;  
}
```

3. Add **Up**-button navigation to `SettingsActivity` by editing its declaration in the `AndroidManifest.xml` file to define the activity's parent as `MainActivity`.

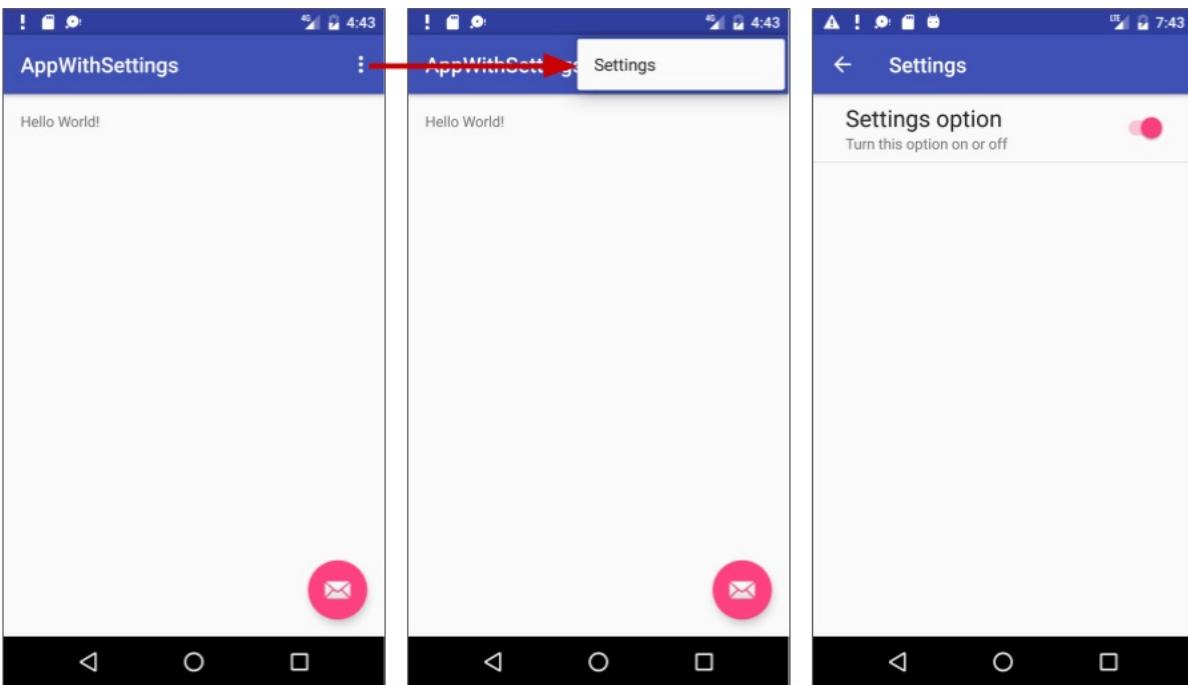
- i. Find the `SettingsActivity` declaration in `AndroidManifest.xml`:

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

- ii. Change the declaration to the following:

```
<activity android:name=".SettingsActivity"  
        android:label="Settings"  
        android:parentActivityName=".MainActivity">  
    <meta-data  
        android:name="android.support.PARENT_ACTIVITY"  
        android:value=".MainActivity"/>  
    </activity>
```

4. Run the app. Tap the overflow icon for the options menu (as shown on the left side of the figure below), and tap **Settings** to see the settings activity (as shown in the center of the figure below). Tap the **Up** button in the app bar of the settings activity, shown on the right side of the figure below, to return to the main activity.



1.5 Save the default values in shared preferences

Although the default value for the toggle switch setting has already been set in the `android:defaultValue` attribute (in Step 1.2 of this task), the app must save the default value in the SharedPreferences file for each setting when the user first opens the app. Follow these steps to set the default value for the toggle switch:

1. In MainActivity, add the following to the end of the existing `onCreate()` method:

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    PreferenceManager.setDefaultValues(this, R.xml.preferences, false);
}
```

The above code ensures that the settings are properly initialized with their default values. The `setDefaultValues()` method takes three arguments:

2. The app `context`, such as `this`.
3. The resource ID (`preferences`) for the XML resource file with one or more settings.
4. A boolean indicating whether the default values should be set more than once. When `false`, the system sets the default values only if this method has never been called in the past. As long as you set this third argument to `false`, you can safely call this method every time the main activity starts without overriding the user's saved settings values. However, if you set it to `true`, the method will override any previous values with the defaults.

1.6 Read the changed settings value from shared preferences

When the app starts, the `MainActivity`'s `onCreate()` method can read setting values that have changed, and use the changed values rather than the default values.

Each setting is identified using a key-value pair. The Android system uses this key-value pair when saving or retrieving settings from a `SharedPreferences` file for your app. When the user changes a setting, the system updates the corresponding value in the `SharedPreferences` file. To use the value of the setting, the app can use the key to get the setting from the `SharedPreferences` file.

Follow these steps to add that code:

1. Before adding code to read the setting value, create a static string variable in `SettingsActivity` to hold the key for the value:

```
public class SettingsActivity extends AppCompatActivity {  
    public static final String  
        KEY_PREF_EXAMPLE_SWITCH = "example_switch";  
    ...  
}
```

2. In the `onCreate()` method in `MainActivity`, and add the following at end of the method:

```
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    SharedPreferences sharedPref =  
        PreferenceManager.getDefaultSharedPreferences(this);  
    Boolean switchPref = sharedPref.getBoolean  
        (SettingsActivity.KEY_PREF_EXAMPLE_SWITCH, false);  
}
```

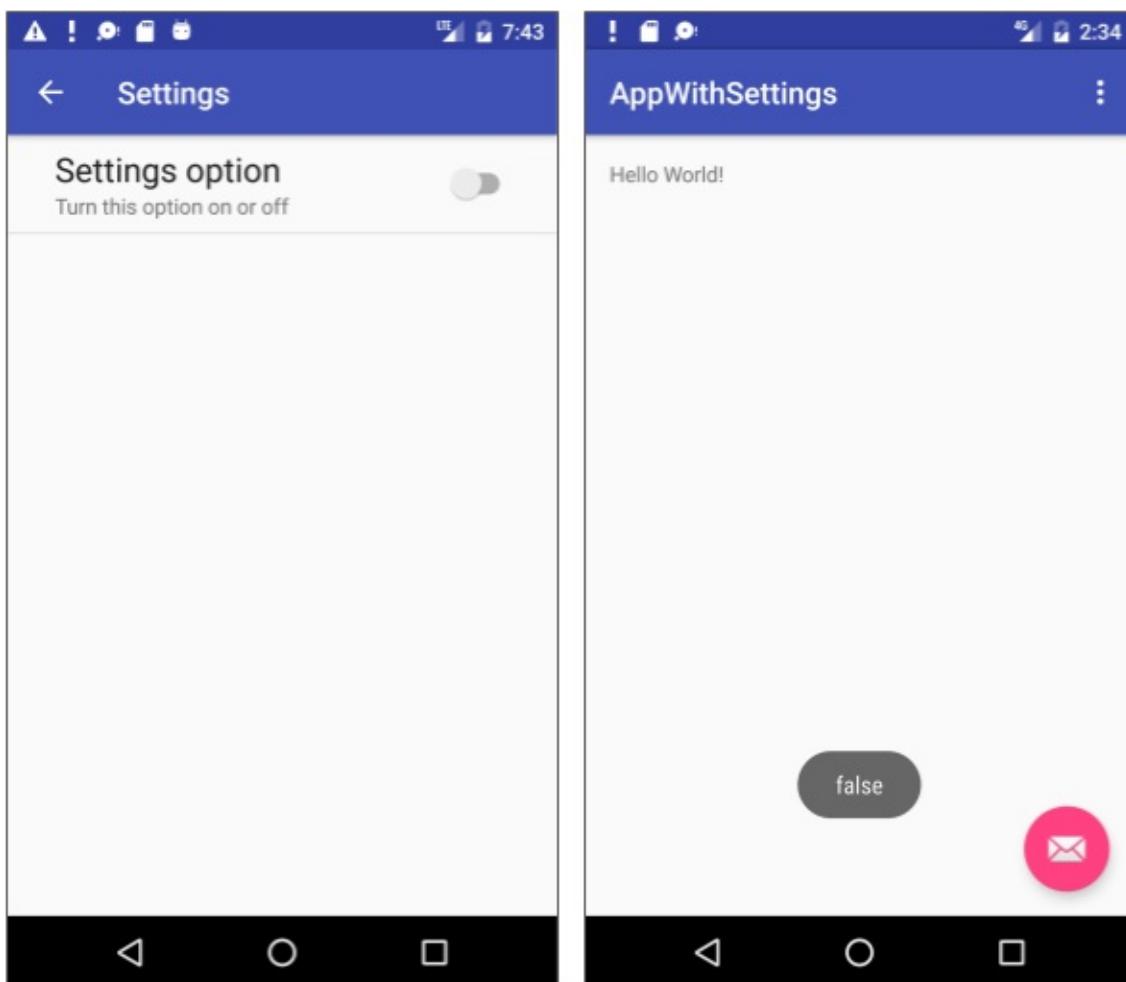
The above code snippet uses

- o `PreferenceManager.getDefaultSharedPreferences(this)` to get the setting as a `SharedPreferences` object (`sharedPref`).
- o `getBoolean()` to get the Boolean value of the setting that uses the key (`KEY_PREF_EXAMPLE_SWITCH` defined in `SettingsActivity`) and assign it to `switchPref`. If there is no value for the key, the `getBoolean()` method sets the setting value (`switchPref`) to `false`. For other values such as strings, integers, or floating point numbers, you can use the `getString()`, `getInt()`, or `getFloat()` methods respectively.

3. Add a `Toast.makeText()` method to `onCreate()` that displays the value of the `switchPref` setting in a toast:

```
Toast.makeText(this, switchPref.toString(), Toast.LENGTH_SHORT).show();
```

4. Run the app and then follow these steps:
 - i. Tap **Settings** to see the settings activity.
 - ii. Tap the setting to change the toggle from on to off, as shown on the left side of the figure below.
 - iii. Tap the **Up** button in the settings activity to return to the main activity. The toast message should appear in the main activity with the value of the setting, as shown on the right side of the figure below.
 - iv. Repeat these steps to see the toast message change as you change the setting.



Whenever the `MainActivity` starts or restarts, the `onCreate()` method should read the setting values in order to use them in the app. The `Toast.makeText()` method would be replaced with a method that initializes the settings.

You now have a working settings activity in your app.

Solution code:

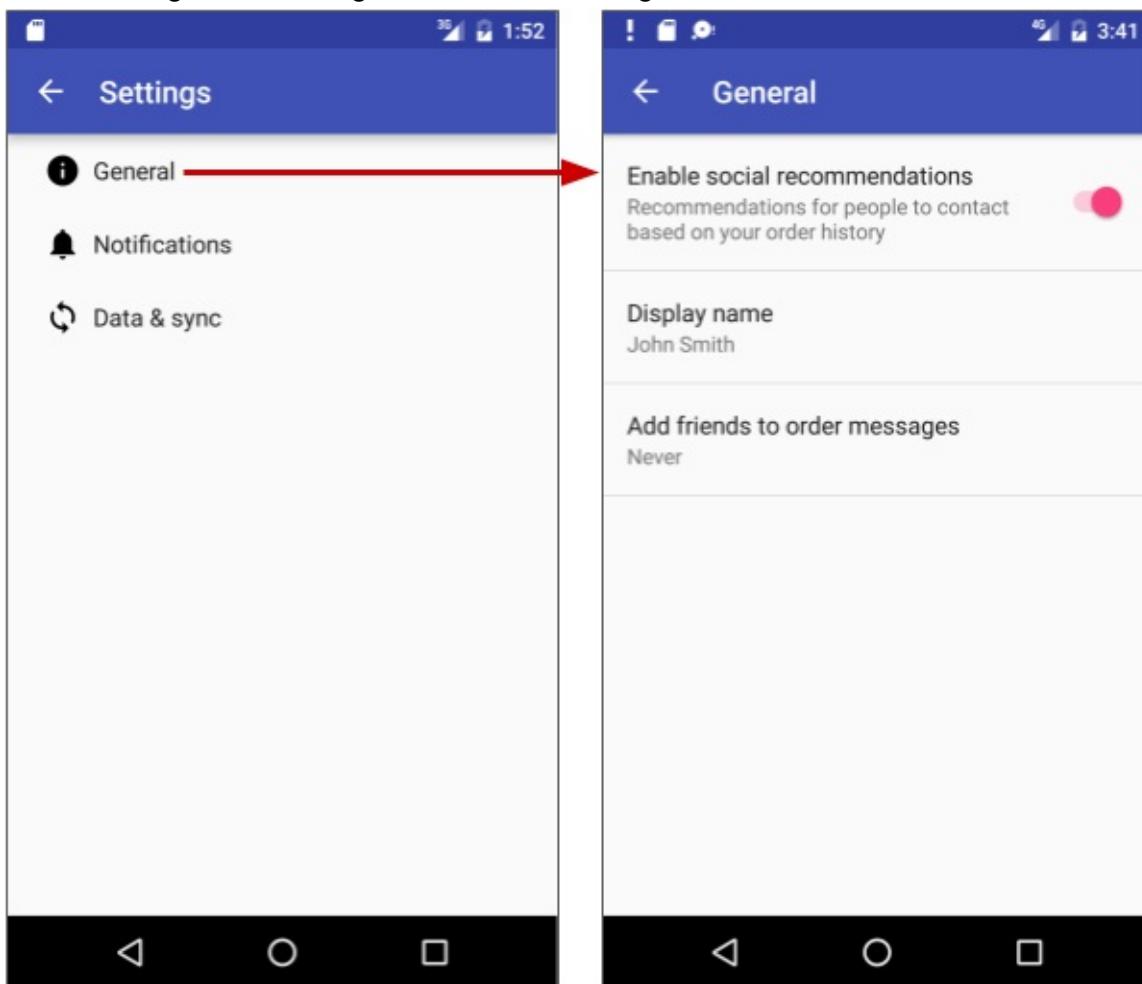
Android Studio project: [AppWithSettings](#)

Task 2: Using the Settings Activity template

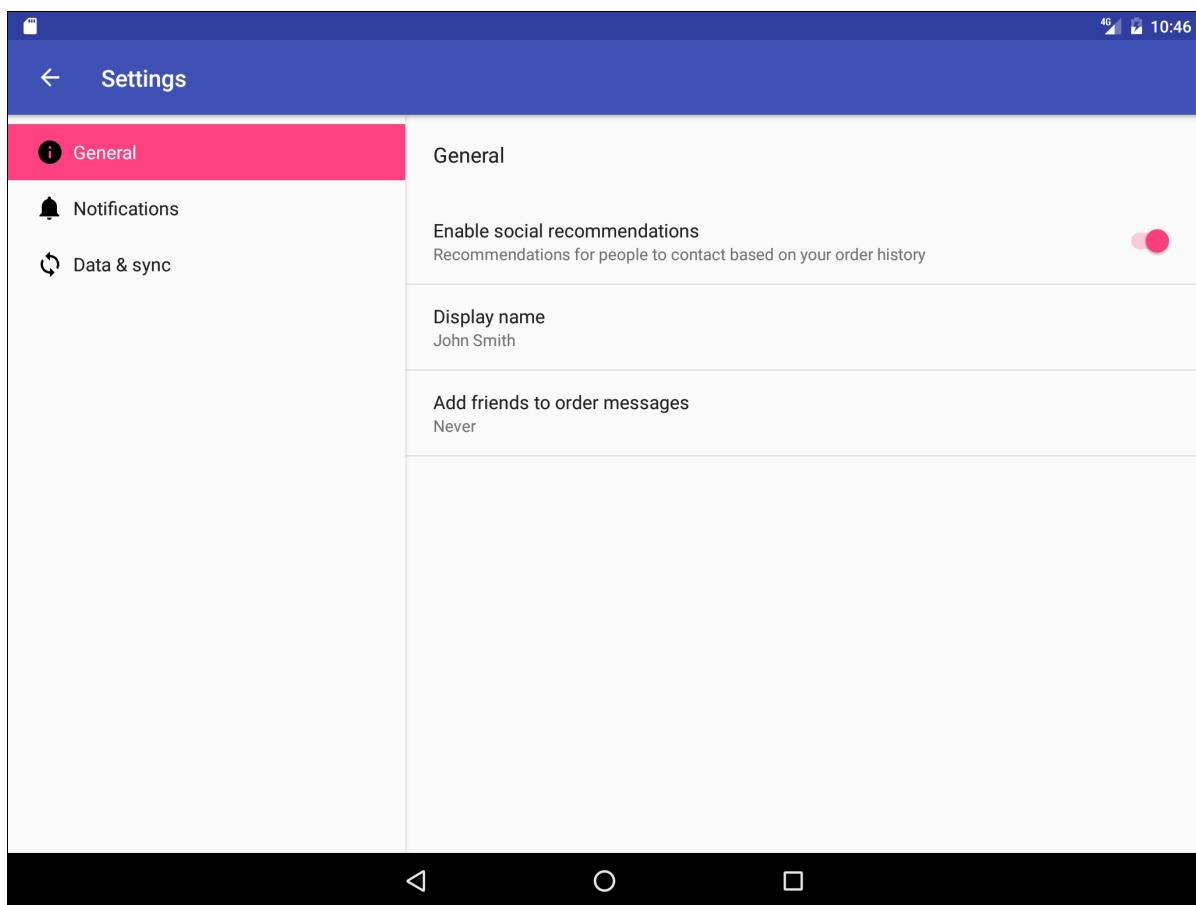
If you need to build several sub-screens of settings and you want to take advantage of tablet-sized screens as well as maintain compatibility with older versions of Android for tablets, Android Studio provides a shortcut: the Settings Activity template.

In the previous task you learned how to use an empty settings activity and a blank fragment in order to add a setting to an app. Task 2 will now show you how to use the Settings Activity template supplied with Android Studio to:

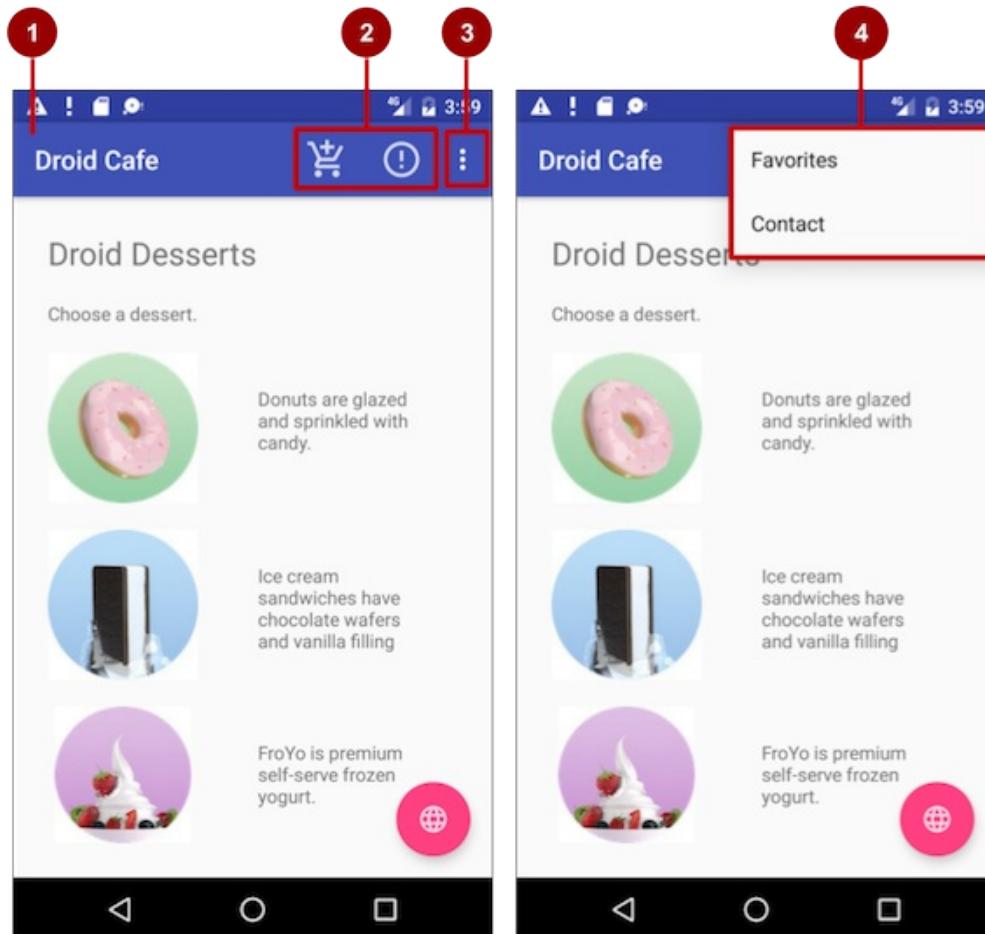
- Divide multiple settings into groups.
- Customize the settings and their values.
- Display a main Settings screen with a header link for each group of settings, such as General for general settings, as shown in the figure below.



- Display a master/detail screen layout with a header link for each group on the left (master) side, and the group of settings on the right (detail) side, as shown in the figure below.



In a previous practical you created an app called Droid Cafe using the Basic Activity template, which provides an options menu in the app bar as shown below.



In the above figure:

1. App bar.
2. Options menu action icons.
3. Overflow button.
4. Options overflow menu.

Android Studio project: To start the project from where you left off in the previous practical, download the Android Studio project [DroidCafe](#).

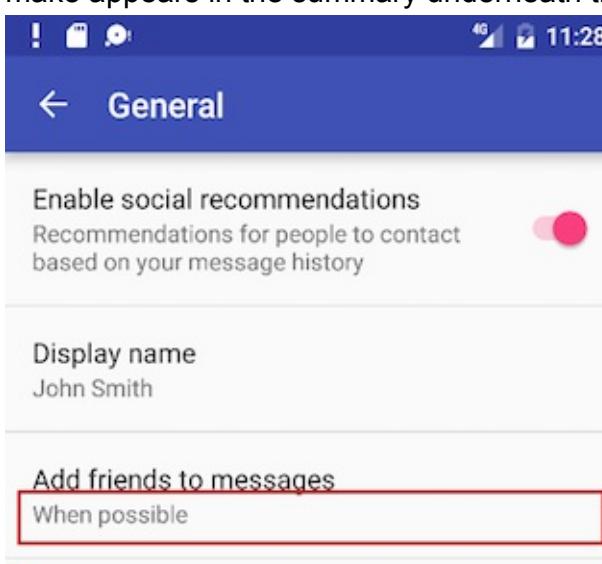
2.1 Explore the Settings Activity template

To include the Settings Activity template in your app project in Android Studio, follow these steps:

1. Copy the **DroidCafe** project folder, rename it to **DroidCafeWithSettings**, and refactor it. (See the [Appendix](#) for instructions on copying a project.) Run the app to make sure it runs properly.
2. Select **app** at the top of the Project: Android view, and choose **New > Activity > Settings Activity**.

3. In the dialog that appears, accept the Activity Name (**SettingsActivity** is the suggested name) and the Title (**Settings**).
4. Click the three dots at the end of the Hierarchical Parent field and choose **MainActivity** as the parent activity, so that the **Up** button in the Settings Activity returns the user to the MainActivity. Choosing the parent activity automatically updates the AndroidManifest.xml file to support **Up** button navigation.
5. Click **Finish**.

The Settings Activity template not only provides layouts for smartphone-sized and tablet-sized screens, but also provides the function of listening to a settings change, and changing the summary to reflect the settings change. For example, if you change the "Add friends to messages" setting (the choices are **Always**, **When possible**, or **Never**), the choice you make appears in the summary underneath the setting:



In general, you need not change the Settings Activity template code in order to customize the activity for the settings you want in your app. You can customize the settings titles, summaries, possible values, and default values without changing the template code, and even add more settings to the groups that are provided.

You use the Settings Activity template code as-is. To make it work for your app, add code to the Main Activity to set the default settings values, and to *read* and *use* the settings values, as shown later in this task.

The Settings Activity template creates the following for you:

- XML files in the **res > xml** directory, which you can add to or customize for the settings you want.
 - **pref_data_sync.xml**: PreferenceScreen layout for "Data & sync" settings.
 - **pref_general.xml**: PreferenceScreen layout for "General" settings.
 - **pref_headers.xml**: Layout of headers for the Settings main screen.

- `pref_notification.xml`: PreferenceScreen layout for "Notifications" settings.

The above XML layouts use various subclasses of the `Preference` class rather than `View` objects, and direct subclasses provide containers for layouts involving multiple settings. For example, `PreferenceScreen` represents a top-level `Preference` that is the root of a Preference hierarchy. The above files use `PreferenceScreen` at the top of each screen of settings. Other `Preference` subclasses for settings provide the appropriate UI for users to change the setting. For example:

- `CheckBoxPreference`: A checkbox for a setting that is either enabled or disabled.
- `ListPreference`: A dialog with a list of radio buttons.
- `SwitchPreference`: A two-state toggleable option (such as on/off or true/false).
- `EditTextPreference`: A dialog with an `EditText` widget.
- `RingtonePreference`: A dialog with ringtones on the device.

Tip: You can edit the XML files to change the default settings to settings you need for your app.

- String resources in the `strings.xml` file in the `res > values` directory, which you can customize for the settings you want.

All strings used in the Settings Activity, such as the titles for settings, string arrays for lists, and descriptions for settings, are defined as string resources at the end of this file.

They are marked by comments such as `<!-- Strings related to Settings -->` and `<!-- Example General settings -->`.

Tip: You can edit these strings to customize the settings you need for your app.

- `SettingsActivity` in the `java > com.example.android.projectname` directory, which you can use as is.

This is the activity that displays the settings. `SettingsActivity` extends `AppCompatActivity` for maintaining compatibility with older versions of Android.

- `AppCompatPreferenceActivity` in the `java > com.example.android.projectname` directory, which you use as is.

This activity is a helper class that `SettingsActivity` uses to maintain backwards compatibility with previous versions of Android.

2.2 Add the Settings menu item and connect it to the activity

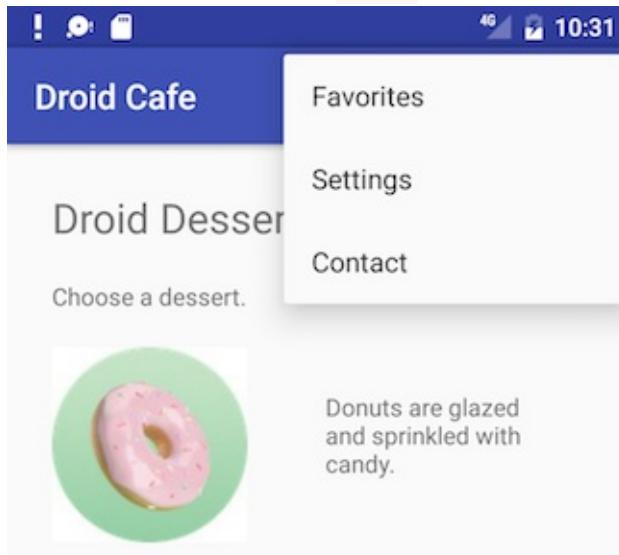
As you learned in a previous practical, you can edit the `menu_main.xml` file in the `res > menu` directory for the options menu to add or remove menu items.

1. Edit the `menu_main.xml` file to add another menu item called **Settings** with the new resource id `action_settings` :

```
<item  
    android:id="@+id/action_settings"  
    android:orderInCategory="50"  
    android:title="Settings"  
    app:showAsAction="never" />
```

Specify `"never"` for the `app:showAsAction` attribute so that **Settings** appears only in the overflow options menu and not in the app bar itself, since it should not be used often.

Specify `"50"` for the `android:orderInCategory` attribute so that **Settings** appears below **Favorites** (set to `"40"`) but above **Contact** (set to `"100"`).



2. Extract the string resource for `"Settings"` in the `android:title` attribute to the resource name `settings`.
3. In `MainActivity`, find the `switch-case` block in the `onOptionsItemSelected()` method which handles the tap on items in the options menu:

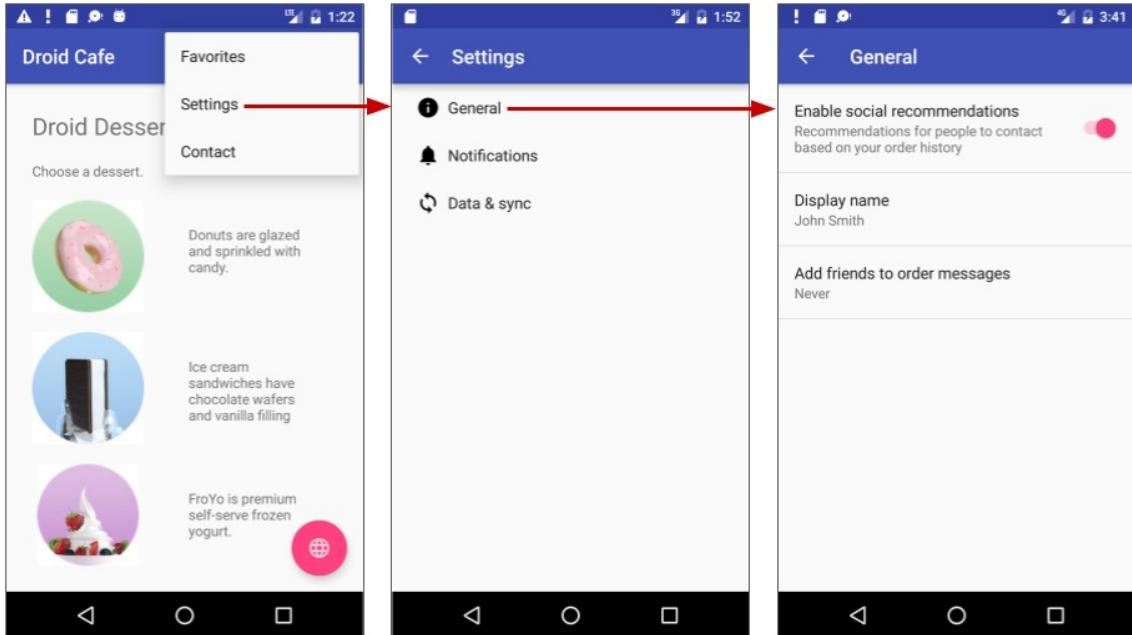
```
public boolean onOptionsItemSelected(MenuItem item) {  
    switch (item.getItemId()) {  
        case R.id.action_order:  
            showToast(getString(R.string.action_order_message));  
            return true;  
        case R.id.action_status:  
            showToast(getString(R.string.action_status_message));  
            return true;  
        case R.id.action_favorites:  
            showToast(getString(R.string.action_favorites_message));  
            return true;  
        case R.id.action_contact:  
            showToast(getString(R.string.action_contact_message));  
            return true;  
    }  
  
    return super.onOptionsItemSelected(item);  
}
```

4. Use an `intent` to launch the `SettingsActivity` from the `MainActivity`. Add the `intent` to the end of the `switch case` block:

```
...  
case R.id.action_settings:  
    Intent intent = new Intent(this, SettingsActivity.class);  
    startActivity(intent);  
    return true;  
}  
  
return super.onOptionsItemSelected(item);  
}
```

5. Run the app using a smartphone or smartphone emulator so that you can see how the `Settings Activity` template handles the smartphone screen size, and follow these steps:
 - i. Tap the overflow icon for the options menu, and tap **Settings** to see the settings activity, as shown on the left side of the figure below.
 - ii. Tap each setting header (**General**, **Notifications**, and **Data & sync**), as shown in the center of the figure below, to see the group of settings on each child screen of the `Settings` screen, shown on the right side of the figure below.

- iii. Tap the Up button in the settings activity to return to the main activity.



2.3 Customize the settings provided by the template

To customize the settings provided by the Settings Activity template, edit the string and string array resources in the strings.xml file and the layout attributes for each setting in the files in the **xml** directory. In this step you will change the "Data & sync" settings.

1. Open the **strings.xml** file in the **res > values** directory, and scroll the contents to the

`<!-- Example settings for Data & Sync -->` comment:

```

<!-- Example settings for Data & Sync -->
<string name="pref_header_data_sync">Data & sync</string>

<string name="pref_title_sync_frequency">Sync frequency</string>
<string-array name="pref_sync_frequency_titles">
    <item>15 minutes</item>
    <item>30 minutes</item>
    <item>1 hour</item>
    <item>3 hours</item>
    <item>6 hours</item>
    <item>Never</item>
</string-array>
<string-array name="pref_sync_frequency_values">
    <item>15</item>
    <item>30</item>
    <item>60</item>
    <item>180</item>
    <item>360</item>
    <item>-1</item>
</string-array>
...

```

2. Edit the `pref_header_data_sync` string resource, which is set to `Data & sync` (the `&` is HTML code for an ampersand). Change the value to **Account** (without quotation marks).
3. Refactor the resource name by following these steps (the app will still work without refactoring the names, but refactoring makes the code easier to understand):
 - i. Control-click (or right-click) the `pref_header_data_sync` resource name and choose **Refactor > Rename**.
 - ii. Change the name to `pref_header_account`, click the option to search in comments and strings, and click **Refactor**.
4. Edit the `pref_title_sync_frequency` string resource (which is set to `Sync frequency`) to **Market**.
5. **Refactor > Rename** the resource name to `pref_title_account` as you did previously.
6. **Refactor > Rename** the string array resource name `pref_sync_frequency_titles` to `pref_market_titles`.
7. Change each value in the `pref_market_titles` string array (`15 minutes`, `30 minutes`, `1 hour`, etc.) to be the titles of markets, such as **United States**, **Canada**, etc., rather than frequencies:

```
<string-array name="pref_market_titles">
    <item>United States</item>
    <item>Canada</item>
    <item>United Kingdom</item>
    <item>India</item>
    <item>Japan</item>
    <item>Other</item>
</string-array>
```

8. **Refactor > Rename** the string array resource name `pref_sync_frequency_values` to `pref_market_values`.
9. Change each value in the `pref_market_values` string array (`15`, `30`, `60`, etc.) to be values for the markets—abbreviations such as **US**, **CA**, etc.:

```
<string-array name="pref_market_values">
    <item>US</item>
    <item>CA</item>
    <item>UK</item>
    <item>IN</item>
    <item>JA</item>
    <item>-1</item>
</string-array>
```

10. Scroll down to the `pref_title_system_sync_settings` string resource, and edit the resource (which is set to `System sync settings`) to **Account settings**.

11. **Refactor > Rename** the string array resource name `pref_title_system_sync_settings` to `pref_title_account_settings`.
12. Open the `pref_data_sync.xml` file. The `ListPreference` in this layout defines the setting you just changed. Note that the string resources for the `android:entries`, `android:entryValues` and `android:title` attributes are now changed to the values you supplied in the previous steps:

```
<ListPreference  
    android:defaultValue="180"  
    android:entries="@array/pref_market_titles"  
    android:entryValues="@array/pref_market_values"  
    android:key="sync_frequency"  
    android:negativeButtonText="@null"  
    android:positiveButtonText="@null"  
    android:title="@string/pref_title_account" />
```

13. Change the `android:defaultValue` attribute:

```
    android:defaultValue="US"
```

Since the key for this setting preference (`"sync_frequency"`) is hard-coded elsewhere in the Java code, don't change the `android:key` attribute—keep using `"sync_frequency"` as the key for this setting in this example. If you are thoroughly customizing the settings for a real-world app, you would spend the time changing the hard-coded keys throughout the code.

Note: Why not use a string resource for the key? Because string resources can be localized for different languages using multiple-language XML files, and the key string might be inadvertently translated along with the other strings, which would cause the app to crash.

2.4 Add code to set the default values for the settings

Find the `onCreate()` method in `MainActivity`, and add the following `PreferenceManager.setDefaultValues` statements at the end of the method:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    PreferenceManager.setDefaultValues(this, R.xml.pref_general, false);  
    PreferenceManager.setDefaultValues(this, R.xml.pref_notification, false);  
    PreferenceManager.setDefaultValues(this, R.xml.pref_data_sync, false);  
}
```

The default values are already specified in the XML file with the `android:defaultValue` attribute, but the above statements ensure that the Shared Preferences file is properly initialized with the default values. The `setDefaultValues()` method takes three arguments:

- The app `context`, such as `this`.
- The resource ID for the settings layout XML file which includes the default values set by the `android:defaultValue` attribute.
- A boolean indicating whether the default values should be set more than once. When `false`, the system sets the default values only if this method has never been called in the past. As long as you set this third argument to `false`, you can safely call this method every time your activity starts without overriding the user's saved settings values by resetting them to the default values. However, if you set it to `true`, the method will override any previous values with the defaults.

2.5 Add code to read values for the settings

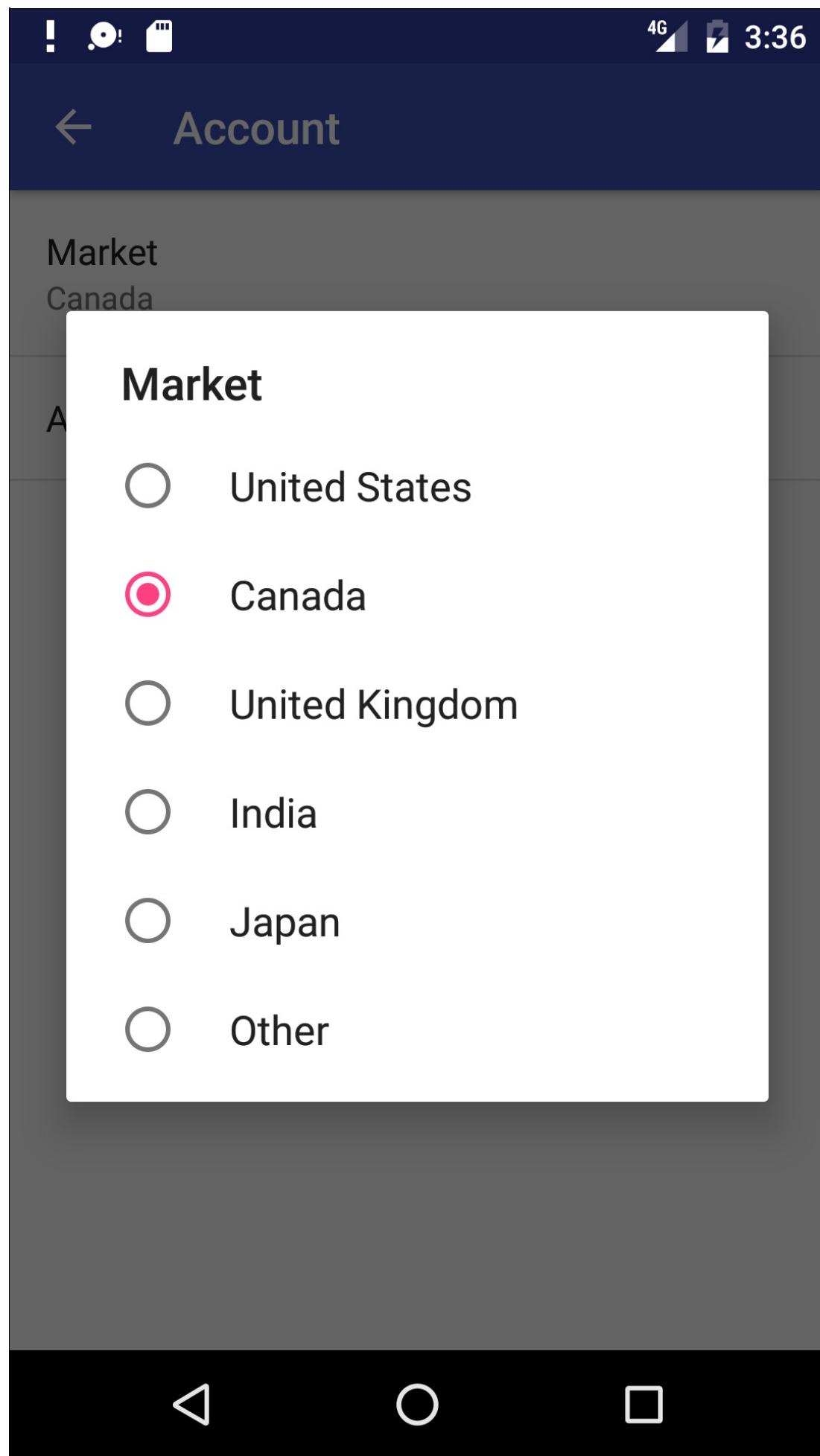
1. Add the following code at the end of the `MainActivity` `onCreate()` method. You can add it immediately after the code you added in the previous step to set the defaults for the settings:

```
...
SharedPreferences sharedPref =
    PreferenceManager.getDefaultSharedPreferences(this);
String marketPref = sharedPref.getString("sync_frequency", "-1");
Toast.makeText(this, marketPref, Toast.LENGTH_SHORT).show();
}
```

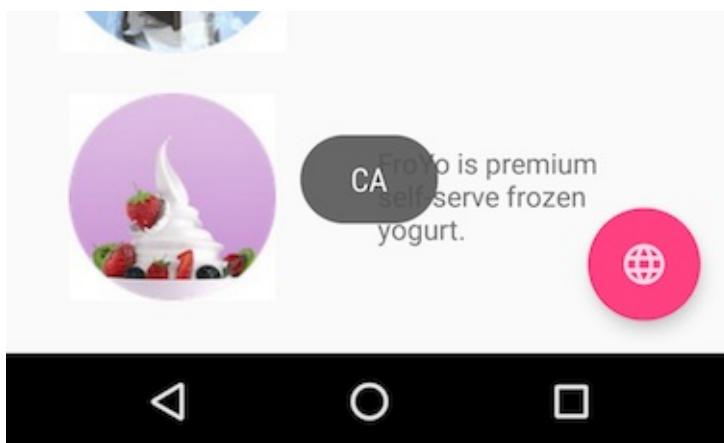
As you learned in the previous task, you use

`PreferenceManager.getDefaultSharedPreferences(this)` to get the setting as a `SharedPreferences` object (`marketPref`). You then use `getString()` to get the string value of the setting that uses the key (`sync_frequency`) and assign it to `marketPref`. If there is no value for the key, the `getString()` method sets the setting value of `marketPref` to `-1`, which is the value of `other` in the `pref_market_values` array.

2. Run the app, again using a smartphone or smartphone emulator. When the app's main screen first appears, you see a toast message at the bottom of the screen. The first time you run the application, you should see "-1" displayed in the toast because you haven't changed the setting yet.
3. Tap **Settings** in the options menu, and tap **Account** in the Settings screen. Choose **Canada** under "Market" as shown below:

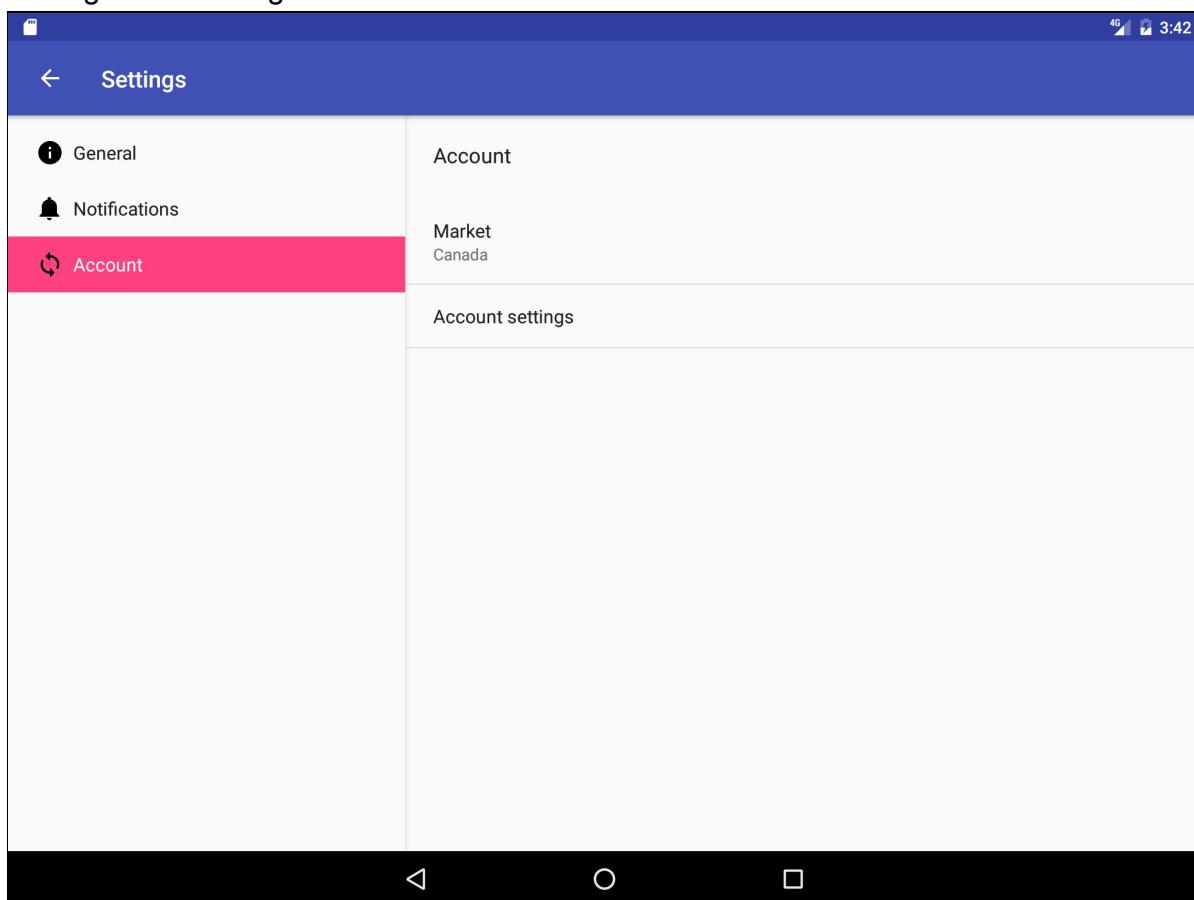


4. Tap the **Up** button in the app bar to return to the Settings screen, and tap it again to return to the main screen. You should see a toast message with "CA" (for Canada):



You have successfully integrated the Settings Activity with the Droid Cafe app.

5. Now run the app on a tablet or tablet emulator. Because a tablet has a physically larger screen, the Android runtime takes advantage of the extra space. On a tablet, the settings and details are displayed on the same screen making it easier for users to manage their settings.



Solution code

Android Studio project: [DroidCafeWithSettings](#) (Includes coding challenge #1.)

Android Studio project: [DroidCafeWithSettingsChallenge](#) (Includes coding challenge #2.)

Coding challenges

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

Challenge 1: Add code to DroidCafeWithSettings that reads the value of the toggle switch "Enable social recommendations" on the General child screen of Settings, and displays its value along with the "Market" setting in the same toast message on the main screen.

Hint: Use a `Boolean` variable with `shared.Pref.getBoolean` and the key `"example_switch"`.

Challenge 2: The DroidCafeWithSettings app displays the settings on a tablet-sized screen properly, but the **Up** button in the app bar doesn't return the user to the MainActivity as it does on a smartphone-sized screen. This is due to the `onOptionsItemSelected()` method in each fragment in SettingsActivity. It uses the following to restart the SettingsActivity when the user taps the **Up** button:

```
startActivity(new Intent(getActivity(), SettingsActivity.class));
```

The above is the appropriate action on smartphone screens in which Settings headers (**General**, **Notifications**, and **Account**) appear in a separate screen. After changing a setting, you want the user's tap on the **Up** button to take them back to the Settings headers.

However, on a tablet, the headers are always visible in the left pane (while the settings are in the right pane). As a result, tapping the **Up** button doesn't take the user to MainActivity.

Find a way to make the **Up** button work properly in SettingsActivity on tablet-sized screens.

Hint: There are several ways to fix this problem. Consider the following:

- You can use multiple `dimens.xml` files in your app to accommodate different screen sizes. When the app runs on a specific device, the appropriate `dimens.xml` file is chosen based on the qualifiers for the `dimens.xml` files. For example, the app already has a `dimens.xml (w820dp)` file in the **res > values** directory, using the `(w820dp)` qualifier to specify a device with an 820dp screen width or larger. You can add another `dimens.xml` file with the `Large` qualifier to specify any device with a large screen, such as a tablet. The app also includes a `dimens.xml` file in the **res > values** directory for all other devices, such as smartphones.

- You can add the following `bool` resource between the `<resources>` and `</resources>` tags in the `dimens.xml (large)` file, which is automatically chosen for tablets:

```
<resources>
    <bool name="isTablet">true</bool>
</resources>
```

- You can add the following `bool` resource to the `dimens.xml` file, which is chosen when the app runs on any device that is *not* large:

```
<bool name="isTablet">false</bool>
```

- Now you can add an if-else block to the `onOptionsItemSelected()` method in each fragment in `SettingsActivity` that checks to see if `isTablet` is true. If it is, your code can redirect the **Up** button action to `MainActivity`.

Summary

In this practical you learned to:

- Add a toggle switch setting (`SwitchPreference`) with attributes in a preference XML file, and set its attributes:
 - `android:defaultValue` : The setting default value.
 - `android:title` : The setting title.
 - `android:key` : The setting key.
 - `android:summary` : The setting summary.
- Add a settings activity to view settings, and a fragment that extends `PreferenceFragment` for each specific setting.
 - Use `getFragmentManager()` to add the fragment to the settings activity
 - Use `addPreferencesFromResource()` in each fragment to load the appropriate preferences XML file for that fragment.
- Use an `intent` to connect the **Settings** item in the options menu to the settings activity.
- Set the default values for settings using `PreferenceManager.setDefaultValues()`.
- Read the settings values from `SharedPreferences` using `PreferenceManager.getDefaultSharedPreferences()`, and obtain each setting value using `.getString` , `.getBoolean` , etc.

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [App Settings](#)

Learn more

- Android Studio documentation:
 - [Android Studio User Guide](#)
- Android API Guide, "Develop" section:
 - [Settings \(coding\)](#)
 - [Preference class](#)
 - [PreferenceFragment](#)
 - [Fragment](#)
 - [SharedPreferences](#)
 - [Saving Key-Value Sets](#)
 - [Supporting Different Screen Sizes](#)
- Material Design Specification:
 - [Settings \(design\)](#)
- Other:
 - Stack Overflow: [How does one get dimens.xml into Android Studio?](#)
 - Stack Overflow: [Determine if the device is a smartphone or tablet?](#)

10.2A: 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
- Coding challenges
- Summary
- Related Concept
- Learn More

A 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 a 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 practical, you will create a SQLite database for a set of data, display retrieved data in a RecyclerView, add functionality to add, delete, and edit the data in the RecyclerView and store it in the database.

Note: A database that persistently stores your data and abstracts your data into a data model is sufficient for small Android 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. These classes will help to move work off the UI thread to assist in making the user's experience as smooth and natural as possible. In addition to improving the user experience by removing a potential performance issue, they improve your ability to extend and maintain your 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 a [loader](#) such as `AsyncTaskLoader` and `CursorLoader`.

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 and dynamically creating `onClick` handlers.
- Starting a second activity and returning data from it.
- 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

In this practical you will learn to:

- Create and manage a SQLite database with an `SQLiteOpenHelper`.
- Implement insert, delete, update, and query functionality through your open helper.
- Use an adapter and custom click handler to let users interact with the database from the user interface.

What you will DO

You start with 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:

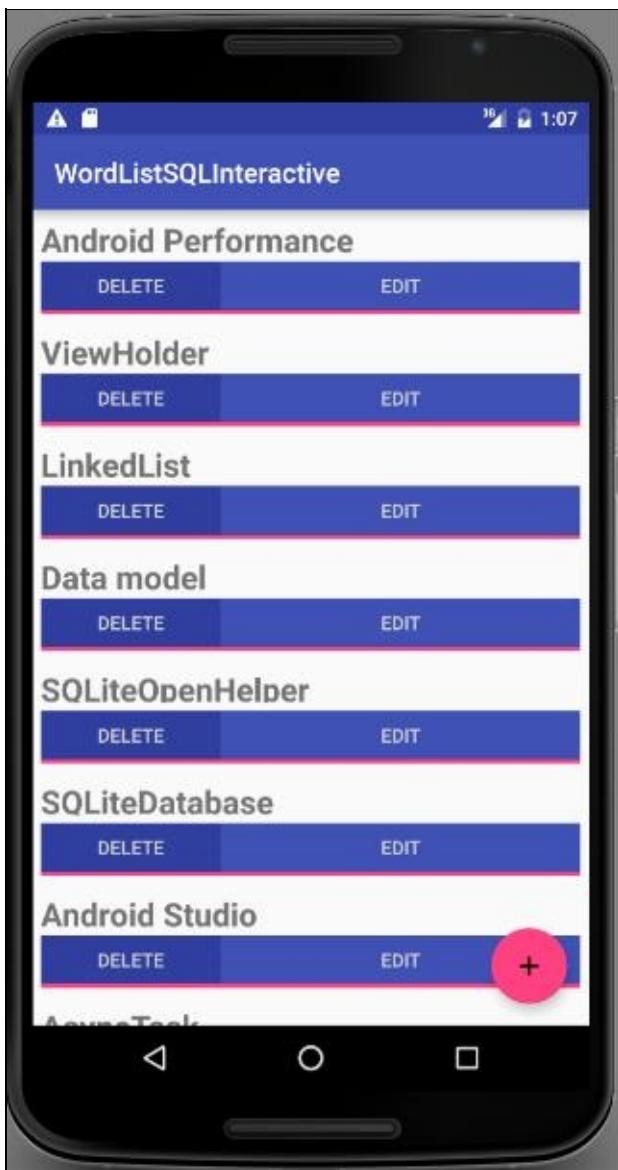
- Implement a custom class to model your data.
- Create a subclass of `SQLiteOpenHelper` that creates and manages your app's database.
- Display data from the database in the `RecyclerView`.
- Implement functionality to add, modify, and delete data in the UI, and store the changes

in the database.

App Overview

Starting from a skeleton app, you will add functionality to:

- Display words from a 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 [WordListSqlStarterCode](#) starter code.
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 in the previous chapter.

[SQLOpenHelper](#) is a utility class in the Android SDK for interacting with a [SQLite database](#) object. It includes `onCreate()` and `onUpdate()` methods that you must implement, and `insert`, `delete`, `update`, and `query` convenience 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.

Note: You can have more than one database per app, and more than one open helper managing them. However consider creating multiple tables in the same database instead of using multiple databases for performance and architectural simplicity

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 image 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

- 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...
public static final String KEY_ID = "_id";
public static final String KEY_WORD = "word";

// ... and a string array of columns.
private static final String[] COLUMNS = { KEY_ID, KEY_WORD };
```

- 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:

- 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 );";
```

- 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 course.

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. Inside the method, 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 SQL workaround 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. Before you run and test your app, you should clear the data from your SQLite database and delete the database. Then we can run our app and 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 > Storage > Clear Data** on your Android emulator or physical device
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 a data model 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. This will identify the word 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 a SQL query.

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.

As such, the query method 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 a 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().

- i. Move the cursor to the first item.

```
cursor.moveToFirst();
```

- ii. Set the 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, "QUERY EXCEPTION! " + e.getMessage());
    } 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. This may happen if you have existing users of your app that use an older version of the database. This method is triggered when a database is first opened. 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 valuable data.

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 will notice 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 parameter for the WordListOpenHelper.
5. Assign the value of the parameter 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. 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 started by clicking the FAB. 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, "INSERT EXCEPTION! " + e.getMessage());
```

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, "INSERT EXCEPTION! " + e.getMessage());
    }
    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 just have to fill in the `onActivityResult()` method.

1. Check to ensure the result is from the correct 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);
    }
}

```

- To update the UI, notify the adapter that the underlying data has changed.

```
mAdapter.notifyDataSetChanged();
```

- 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:

- Implement the delete() method in WordListOpenHelper
- Add a click handler to the DELETE button in WordListAdapter

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. The "?" is a placeholder that gets filled with the string. This is a more secure way of building queries.

```
deleted = mWritableDatabase.delete(WORD_LIST_TABLE,
    KEY_ID + " = ? ", new String[]{String.valueOf(id)});
```

6. Print a log message for exceptions.

```
Log.d (TAG, "DELETE EXCEPTION! " + e.getMessage());
```

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, "DELETE EXCEPTION! " + e.getMessage());
    }
    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 MyButtonOnClickListener class in your starter code. The MyButtonOnClickListener class implements a click listener that stores the id, and the word that you need to make changes to the database.

Each view holder, when attached (bound) to the RecyclerView in the onBindViewHolder method of WordListAdapter, needs to also attach a click listener to the DELETE button, passing the id, and word to the MyButtonOnClickListener constructor. These values are then used by the onClick handler to delete the correct item and notify the adapter, which item has been removed.

Note that you cannot use the position argument passed into onBindViewHolder, because it may be stale by the time the click handler is called. You have to keep a reference to the view holder and get the position with [getAdapterPosition\(\)](#).

Solution:

```
// Keep a reference to the view holder for the click listener
final WordViewHolder h = holder; // needs to be final for use in callback

// Attach a click listener to the DELETE button.
holder.delete_button.setOnClickListener(
    new MyButtonOnClickListener(current.getId(), null) {

        @Override
        public void onClick(View v) {
            int deleted = mDB.delete(id);
            if (deleted >= 0)
                notifyItemRemoved(h.getAdapterPosition());
        }
    });
}
```

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, do the following steps:

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, "UPDATE EXCEPTION: " + e.getMessage());
```

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, "UPDATE EXCEPTION! " + e.getMessage());
    }
    return mNumberOfRowsUpdated;
}

```

7.2. Add a click listener to the EDIT button

And here is the code for the Edit click listener when we bind the View in the onBindViewHolder method of WordListAdapter. This listener has nothing database specific. It starts the EditWordActivity Activity using an Intent and passes it the current id, position, and word in the Extras.

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(
    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, h.getAdapterPosition());
        intent.putExtra(EXTRA_WORD, word);

        // Start an empty edit activity.
        ((Activity) mContext).startActivityForResult(
            intent, MainActivity.WORD_EDIT);
    }
});

```

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 the onActivityResult method in your MainActivity.

```
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 a SQLite database, and so not every edge case is considered, not every value is tested, and everybody is assumed to be well behaved. If this were a production app, you would have greater security considerations, and content would need to be tested for validity until you know it is not malicious.
- 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.

Solution code

Android Studio project: [WordListSql finished](#)

Coding challenges

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge 1: Extend the app to have an editable definition for each word in the database.

Challenge 2: Add a confirmation dialog to the delete functionality.

Summary

In this chapter, you learned how to

- Use a [SQLiteDatabase](#) to store user data persistently.
- Work with a [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

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [SQLite Database](#)

Learn more

Developer Documentation:

- [Storage Options](#)
- [Saving Data in SQL Databases](#)

10.2B: Searching a 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](#)
- [Coding challenge](#)
- [Summary](#)
- [Related Concept](#)
- [Learn More](#)

What you should already KNOW

For this practical you should be familiar with:

- SQLite databases
- Writing basic SQLite queries

What you will LEARN

You will learn to:

- Add search functionality to your app via the options menu
- Build search queries for the SQLite database from user input.

What you will DO

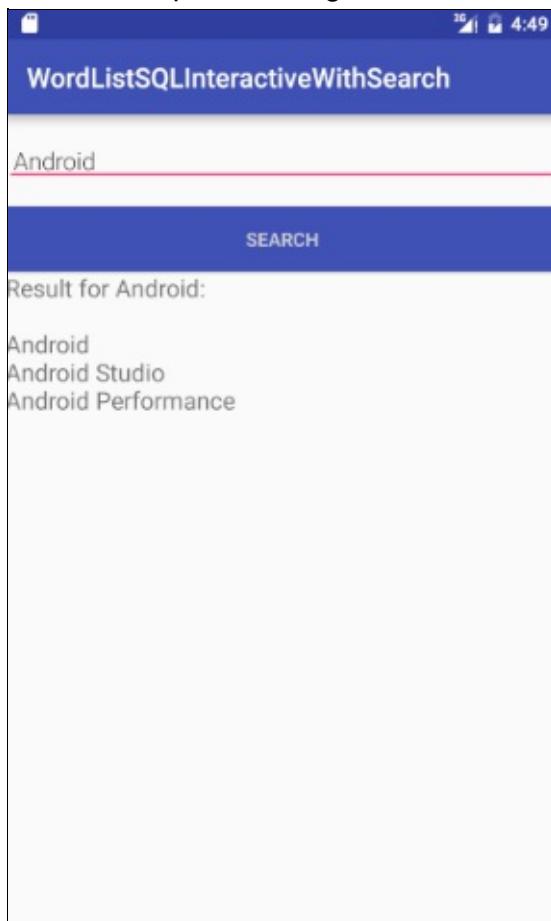
In this practical you will add an item to the options menu for searching the database, and an activity that allows users to enter a search string and displays the result of the search in a text view.

Why: Users should always be able to search the data on their own terms.

Note: The focus of this practical is not optimizing the UX of the search request, but showing you how to query the database.

App Overview

You will make a copy of the finished WordListSQLInteractive app (or WordListSqlStarterCode if you didn't rename it; from a previous practical), call it WordListSQLInteractiveWithSearch, and 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 a production environment, building on existing application code is a common developer task to add features or fix problems.

1. Create your project

1. Download the [WordListSQL finished](#) app.

You can use your own app, or download the base app. As long as the app uses a 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 to ensure it builds and functions correctly.

Task 1. Add Search

1.1. Add an Options Menu with Search item

Use the OptionsMenuSample code from your 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 WordListOpenHelper variable mDB.
5. In onCreate, initialize mDB with a new WordListOpenHelper(this).
6. 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);
    }
}

```

7. 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 onOptionsItemSelected() method in MainActivity.

```

Intent intent = new Intent(getApplicationContext(), SearchActivity.class);
startActivity(intent);

```

2. Build and run your app to make sure SearchActivity is launched when the "Search" menu item is selected from the OptionsMenu.
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 in the SearchActivity

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 mEditWordView, 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 SQLite database cursor. You will implement the search function in the next task.
 - You process the cursor and add the result to mTextView.
1. In SearchActivity, create the showResult function. It is public, takes a View argument, and returns nothing.
 2. Create a `String` variable called `word` and initialize it with the contents of the input edit text view, `mEditWordView`.
 3. Show the search term in the search results TextView; something like

```
"Search term: " + word
```

4. Search the database and get the cursor.

```
Cursor cursor = mDB.search(word);
```

5. To process the cursor, you need to do the following:

- i. Make sure the cursor is not null.
- ii. Move the cursor to the first entry.
- iii. Iterate over the cursor processing the current entry, then advancing the cursor.
- iv. Extract the word.
- v. Display the word in the text view.

6. Close the cursor.
7. If no results are found, the user sees a blank screen with no results. You would want this to be handled in a production app.
8. 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()); // 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.

9. Open WordListOpenHelper.
10. Implement a stub for search, with a String parameter, that returns a null cursor.
11. 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.

A more 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.

Important: For security reasons, you should always validate user input 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 the documentation for various query() methods.

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 spacing.

```
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.

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, nu  
ll, null);  
    } catch (Exception e) {  
        Log.d(TAG, "SEARCH EXCEPTION! " + e);  
    }  
  
    return cursor;  
}
```

Solution code

Android Studio project: [WordListSqlSearchable](#)

Coding challenges

Note: All coding challenges are optional and are not prerequisites for later lessons.

- Handle the case where no results are found in a more user-friendly way.

- 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 [Security Tips](#).

Summary

- An options menu can be an effective UI for searching a SQLite database
- A separate activity to handle the UX for search can help focus the user
- In a production application, SQLite queries should be managed carefully to avoid data corruption or security issues
- SQLite search queries can be constructed dynamically using user input for the query parameters.
- The query() method searches a database for matching words.
- The query() method returns a database cursor which can traverse the result set efficiently
- The cursor can be used to display the results to the user.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [SQLite Database](#)

Learn more

Developer Documentation:

- [Storage Options](#)
- [Saving Data in SQL Databases](#)

11.1A: Implement a Minimalist Content Provider

Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1: Create the MinimalistContentProvider project](#)
- [Task 2: Create a Contract class, a URI scheme, and mock data](#)
- [Task 3: Implement the MiniContentProvider class](#)
- [Task 4: Use a ContentResolver to get data](#)
- [Coding challenges](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

A content provider is a component that securely manages access to a shared repository of data. It provides a consistent interface for applications to access the shared data.

Applications do not access the provider directly but use a content resolver object that provides an interface to and manages the connection with the content provider.

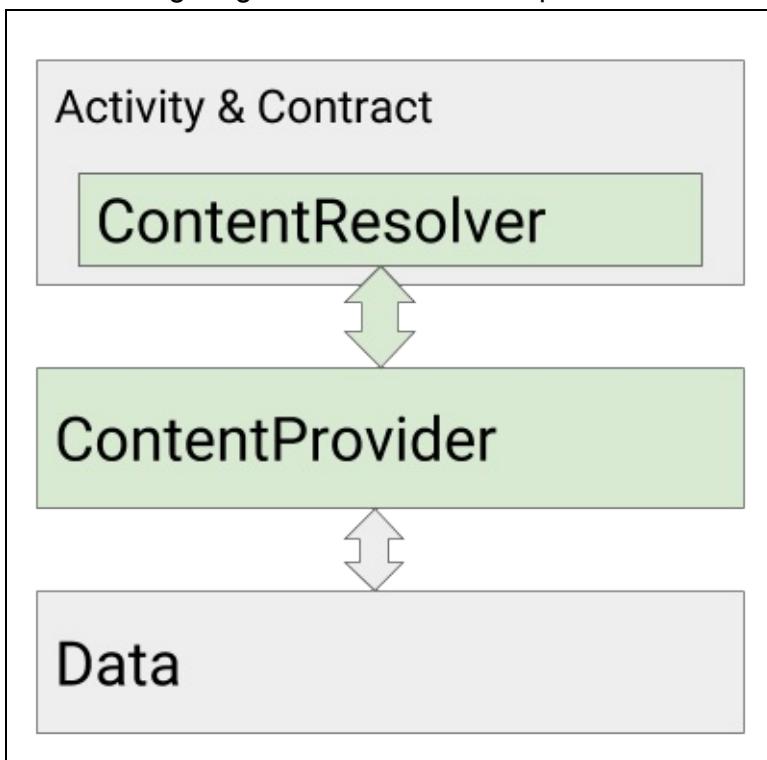
Content providers are useful because:

- Apps cannot share data in Android—except through content providers.
- Content providers allow multiple apps to securely access, use, and modify a single data source. Examples: Contacts, game scores, spell-checking dictionary.
- You can specify levels of access control (permissions) for your content provider.
- You can store data independently from the app. Having a content provider allows you to change how the data is stored without needing to change the user interface. 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 data locally, and the user interface would remain the same to your application.
- This architecture separates data from the user interface so development teams can work independently on the client-facing application and server-side components of your app. For larger, complex apps it is very common that the user interface and the data services are developed by different teams. They can even be separate apps. It is not even required that an app with a content provider has a user interface.

- You can use CursorLoader and other classes that expect to interact with a content provider.

Note: 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 more complex applications.

The following diagram summarizes the parts of the content provider architecture.



Data: The app that creates the content provider owns the data and specifies what permissions other apps have to work with the data.

The data is often stored in a SQLite database, but this is not mandatory. Typically, the data is made available 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 in a contact database contains one entry and that entry may have columns for email addresses and phone numbers.

ContentProvider: The content provider provides a public and secure interface to the data, so that other apps can access the data with the appropriate permissions.

ContentResolver: Used by the Activity to query the content provider. The content resolver returns data as a Cursor object which can then be used, for example, by an adapter, to display the data.

Contract class (not shown): The contract is a public class that exposes important information about the content provider to other apps. This usually includes the URIs to access the data, important constants, and the structure of the data that will be returned.

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://authority/path-to-data/dataset-name

- 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 name or table name)

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 and of itself and is not covered in this course.

Content Resolver: The ContentResolver object provides query(), insert(), update(), and delete() methods for accessing data from a content provider 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.

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 minimal. 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 understand how to:

- Create, build and run interactive apps in Android Studio.
- Display data in a RecyclerView using an adapter.
- Abstract and encapsulate data with data models.
- Create, manage, and interact with a SQLite database using a SQLiteOpenHelper.

What you will LEARN

You will learn:

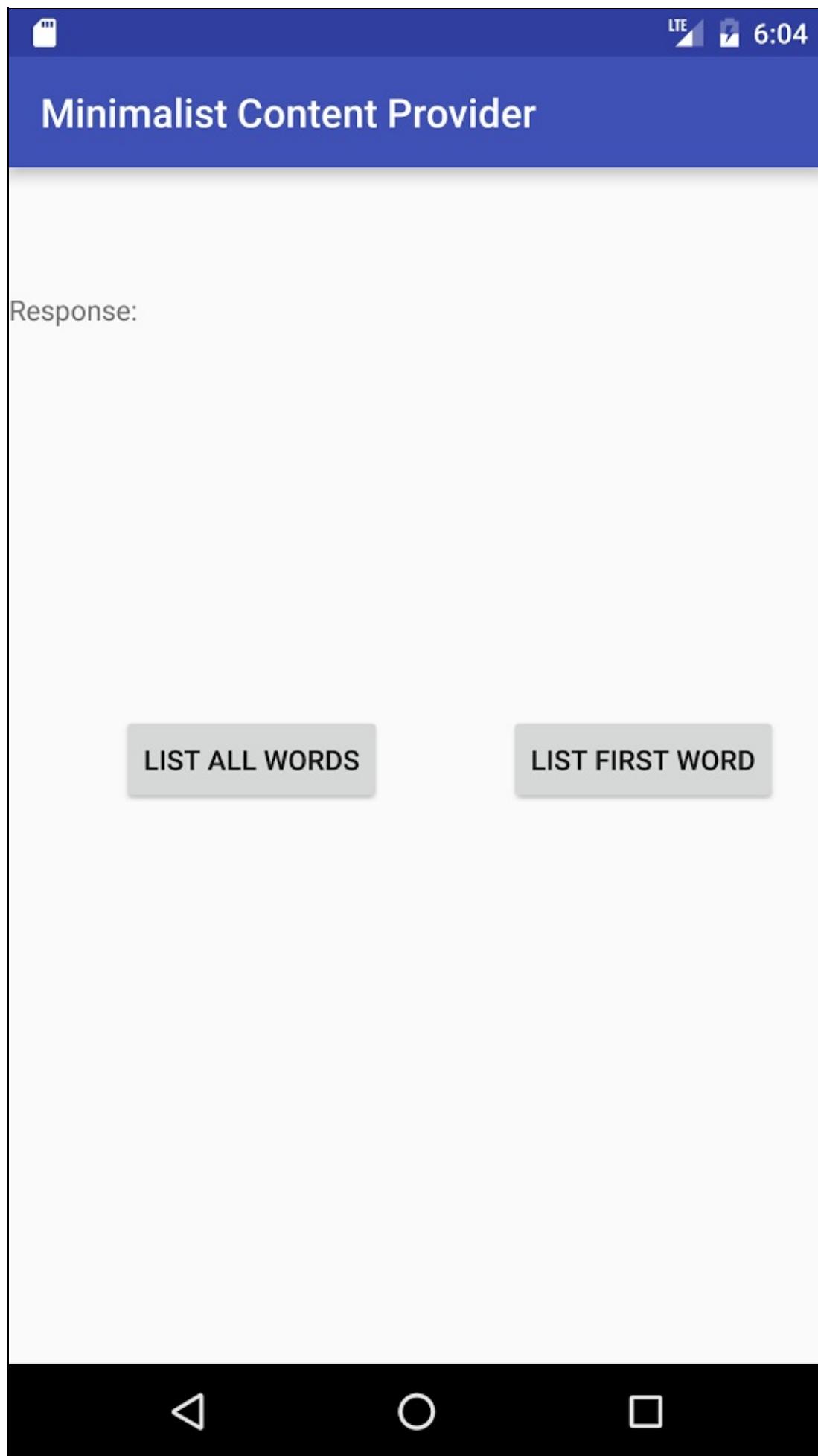
- The architecture and anatomy of a content provider.
- What you need to do to build a minimal content provider that you can use as a template for creating other content providers.

What you will DO

- You will build a stand-alone app to learn the mechanics of building a content provider.

App Overview

- This app generates mock data and stores it in a linked list called "words".
- The app requests data through a content resolver and displays it. The UI consists of one activity with a TextView and two Buttons. The "List all words" button displays all the words, and the "List first word" button displays the first word in the text view.
- The content provider abstracts and manages the interaction between the data source and the user interface.
- The Contract defines URIs and public constants.



Note: 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

1.1. Create a project within the given constraints

Create an app with one activity that shows one text view and two buttons. One button shows the first word in our data (the list), and the other button will list all words. Both buttons call `onClickDisplayEntries()` when they are clicked. For now, this method will use a switch statement to just display a statement that a particular button was clicked. Use the table below as a guideline for setting up your project.

App name	MinimalistContentProvider
One Activity	Empty Activity template Name: MainActivity <pre>private static final String TAG = MainActivity.class.getSimpleName(); public void onClickDisplayEntries (View view){Log.d (TAG, "Yay, I was clicked!");}</pre>
TextView	<code>@+id/textview</code> <code>android:text="response"</code>
Button	<code>@+id/button_display_all</code> <code>android:text="List all words"</code> <code>android:onClick="onClickDisplayEntries"</code>
Button	<code>@+id/button_display_first</code> <code>android:text="List first word"</code> <code>android:onClick="onClickDisplayEntries"</code>

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.
 5. Run the app.

Your MainActivity should be similar to this solution.

Solution:

```
package android.example.com.minimalistcontentprovider;

[... imports]

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, and mock data

The contract contains information about the data that apps need to build queries.

- Contract is a public class and includes important information for other apps that want to connect to this content provider and access your data.
- The URI shows how to build URIs to access the data. The URI scheme behaves as an API to access the data. It's very similar to designing REST calls for CRUD. Other applications will use these Content URIs.

2.1. Create the Contract class

1. Create a new public **Java class Contract** with the following signature. It must be final.

```
public final class Contract {}
```

2. To prevent someone from accidentally instantiating the Contract class, give it an empty private constructor.

```
private Contract() {}
```

2.2. Create the URI

A content URI for content providers has this general form:

```
scheme://authority/path/id
```

- **scheme** is always content:// for content URIs.
- **authority** represents the domain, and for content providers customarily ends in .provider
- **path** is the path to the data
- **id** uniquely identifies the data set to search

The following URI could be used to request all the entries in the "words" table:

```
content://com.android.example.wordcontentprovider.provider/words
```

The URI for accessing 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. To make Authority unique, use 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 identifies the data. You should use something descriptive, for example, the name of a table or file, or the kind of data, such as "words".

```
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. Since it is a never changing string for all instances of the Contract class, you can initialize it statically.

```
public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/" + CONTENT_PATH);
```

4. Create a convenience constant for ALL_ITEMS. This is the dataset name you will use when retrieving all the words. The value is -2 because that's the first lowest value not returned by a method call.
 5. Create a convenience constant for WORD_ID. This is the id you will use when retrieving a single word.
- ```
static final int ALL_ITEMS = -2;
```
- ```
static final String WORD_ID = "id";
```

2.3. Add the MIME Type

Content providers provide content, and you need to specify what type of content they provide. Apps need to know the structure and format of the returned data, so that they can properly handle it.

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 (a record)
- `vnd.android.cursor.dir` for a set of data (multiple records).

The subtype can be anything, but it is a good practice to make it informative. 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.

1. Declare the MIME type 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";
```

2.4. Create the mock data

The content provider always presents the results as a [Cursor](#) in a table format that resembles a SQL database. This is independent of how the data is actually stored. 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 **Java class** MiniContentProvider extending ContentProvider. (For this practical, to **not** use the Create Class > Other > Content Provider menu option.)
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(), 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.

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="com.android.example.minimalistcontentprovider.provider" />
```

2. Run your code to make sure it compiles.

3.3. Set up URI matching

A ContentProvider needs to respond to data requests from apps using a number of different URIs. To take appropriate action depending on a particular request URI, the content provider needs to analyze the URI to see if it matches. [UriMatcher](#) is a helper class that you can use for processing the accepted URI schemes for a given content provider.

Basic steps to use UriMatcher:

- 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 to the constructor 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 your own method for initializing the URI matcher.

```
private void initializeUriMatching(){}
```

3. Call `initializeUriMatching` in `onCreate()` of the `MiniContentProvider` class.
4. 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);
```

5. 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 named 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 of the content provider 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. It 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 your internal data access mechanism (for example a SQLite query), execute your internal data access code, and return the result in a [Cursor](#) object.

The query() method

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 a SQL query. Even if you are using another kind of data storage mechanism, 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.) The method returns a Cursor of any kind.

uri	The complete URI. This cannot be null.
projection	Indicates which columns/attributes to access.
selection	Indicates which rows/records of the objects 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, descending or by . If this is null, the default sort or no sort is applied.

Analyze the query() method

1. Identify the following processing steps in the query() method code shown below in the solutions section.

Query processing always consists of these steps:

- i. Match the URI.
 - ii. Switch on the returned code.
 - iii. Process the arguments and build a query appropriate for the backend.
 - iv. Get the data and (if necessary) drop it into a Cursor.
 - v. Return the cursor.
2. 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.

3. 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.
4. Add the code to your app.

Note: 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 full search.
            // For a real-life app, you need error-catching code;
            // here we assume that the
            // 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 query.
            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 your populateCursor() with the last segment of the path, which is the id (index) of the word to retrieve. The populateCursor() method separates the query matching from getting the data and creating the result cursor.

This is a good practice as in a real app, the query() method can become very large.

The query method must return a Cursor type, so the populateCursor() method has to create, fill in, and return a cursor.

- If your data were stored in a SQLite database, executing the query would 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 [MatrixCursor](#) to hold the data to return. A MatrixCursor is a general purpose cursor into an array of objects that grows as needed. To create a MatrixCursor, you supply it with a string array of column names.

The populateCursor() method does the following:

1. Receives the id extracted from the URI.
2. Creates a MatrixCursor to store received data (because the mock data received is not a cursor).
3. Creates and executes a query. For this app, 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.
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 < mData.length; i++) {
            String word = mData[i];
            cursor.addRow(new Object[]{word});
        }
    } else if (id >= 0) {
        // Execute the query to get the requested word.
        String word = mData[id];
        // Add the result to the cursor.
        cursor.addRow(new Object[]{word});
    }
    return cursor;
}
```

Task 4. Use a ContentResolver 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.

4.1. Get the content resolver

The content resolver interacts with the content provider on your behalf.

The content resolver expects a parsed Content URI along with query parameters that assist in retrieving the data.

You don't have to create your own content resolver. You can use the one provided in your application context by the Android framework by calling `getContentResolver()`.

1. In `MainActivity`, 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);
```

Note: the arguments to `getContentResolver.query()` are identical to the parameters of `ContentProvider.query()`.

Next you must define the arguments to `getContentResolver.query()`.

4.2. Define the query arguments

In order for `getContentResolver.query()` to work, you need to declare and assign values to all its arguments.

1. **URI:** Declare the ContentURI that identifies the content provider and the table. 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. 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. // only get words.

```
String[] projection = new String[] {Contract.CONTENT_PATH};
```

3. **selectionClause:** Argument clause for the selection criteria, that is, which rows to return. Formatted as an SQL WHERE clause (excluding the "WHERE" keyword). Passing null returns all rows for the given URI. Since this will vary depending on which button was pressed, declare it now and set it later.

```
String selectionClause;
```

4. **selectionArgs**: Argument values for the selection criteria. If you include ?s in the selection String, they are replaced by values from selectionArgs, in the order that they appear.

IMPORTANT: It is security best practices to always separate selection and selectionArgs.

```
String selectionArgs[];
```

5. **sortOrder**: The order in which to sort the results. Formatted as a SQL ORDER BY clause (excluding the ORDER BY keyword). Usually ASC or DESC; null requests the default sort order, which could be unordered.

```
// For this example, accept the order returned by the response.  
String sortOrder = null;
```

4.3. Decide on selection criteria

The selectionClause and selectionArgs values depend on which button was pressed in our UI.

- 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.)

1. 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;  
}
```

4.4. 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.

1. Examine the following code and make sure you understand everything.

```
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");  
    }  
    cursor.close();  
} else {  
    Log.d(TAG, "onClickDisplayEntries " + "Cursor is null.");  
    mTextView.append("Cursor is null." + "\n");  
}
```

2. Insert this code at the end of `onClickDisplayEntry()`.
3. Run your app.
4. Click the buttons to see the retrieved data in the text view.

Solution code

Android Studio project: [MinimalistContentProvider](#)

Coding challenges

Note: All coding challenges are optional and are not prerequisites for later lessons.

Implement missing methods

Coding Challenge 1: 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

Coding Challenge 2: 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

- Content providers are high-level data abstractions that manage access to a shared repository
- Content providers are primarily intended to be used by apps other than your own.
- Content providers (server-side) are accessed by Content resolvers (app-side)
- A Contract is a public class that exposes important information about a content provider.
- contracts can be useful beyond content providers,
- A content provider needs to define a set of content URIs so apps can access data through your content provider.
- The content URI consists of several components: "content://", a unique content authority (typically a fully-qualified package name) and the content-path.
- Use a content resolver to request data from a content provider and display it to the user.
- If your app does not share data with other apps, then your app does not require a content provider.
- Content providers must implement the `getType()` method which returns the MIME type for each content type.
- Content Providers need to be "published" in the Android manifest using the element within the element
- Content Providers need to inspect the incoming URI to determine URI pattern matches in order to access any data
- You must add target URL patterns to your content provider. The UriMatcher class is a

helpful class for this purpose.

- The essence of a content provider is implemented in its query() method.
- The method signature of the query() method in a content resolver (data requester) matches the method signature of the query() method in a content provider (data source).
- The query() method returns a database-style cursor object regardless if the data is relational or not.

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Content Providers](#)

Learn more

Developer Documentation:

- [Uniform Resource Identifiers or URIs](#)
- [MIME type](#)
- [MatrixCursor and Cursors](#)
- [Content Providers](#)

11.1B: Add a content provider to your database

Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App Overview](#)
- [Task 1. Download and run the base code](#)
- [Task 2: Add a Contract class to WordListSQLInteractive](#)
- [Task 3: Create a Content Provider](#)
- [Task 4: Implement Content Provider methods](#)
- [Coding challenge](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

Content providers in real apps are more complex than the basic 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.

You will rarely build an app from scratch. More often, you will 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.

You can go about this 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 how to:

- Display data in a RecyclerView.
- Start and return from an Activity.
- Create, change, and interact with a SQLite database using a SQLiteOpenHelper.
- Understand the architecture the minimal content provider you built in the previous practical.

What you will LEARN

You will learn how to:

- Create a fully developed content provider for an existing application.
- Refactor an application to accommodate a content provider.

What you will DO

This practical 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:

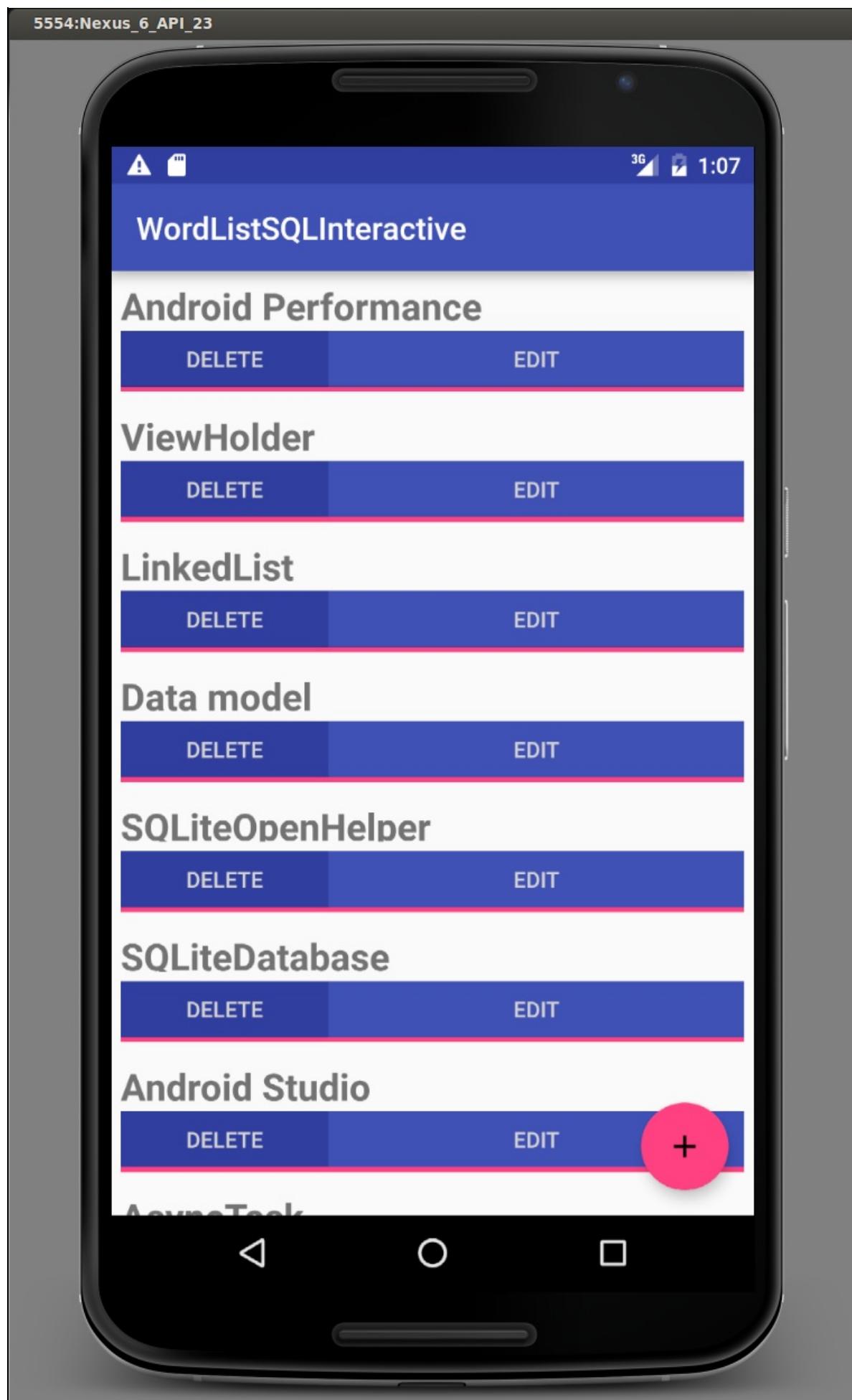
- Implement a Contract class to expose your app's interface to other apps.
- Implement a ContentProvider and query it using a ContentResolver.
- Refactor the MainActivity, WordListAdapter, and WordListOpenHelper classes to work with the content provider.

App Overview

The completed WordListSQLWithContentProvider app will have the following features:

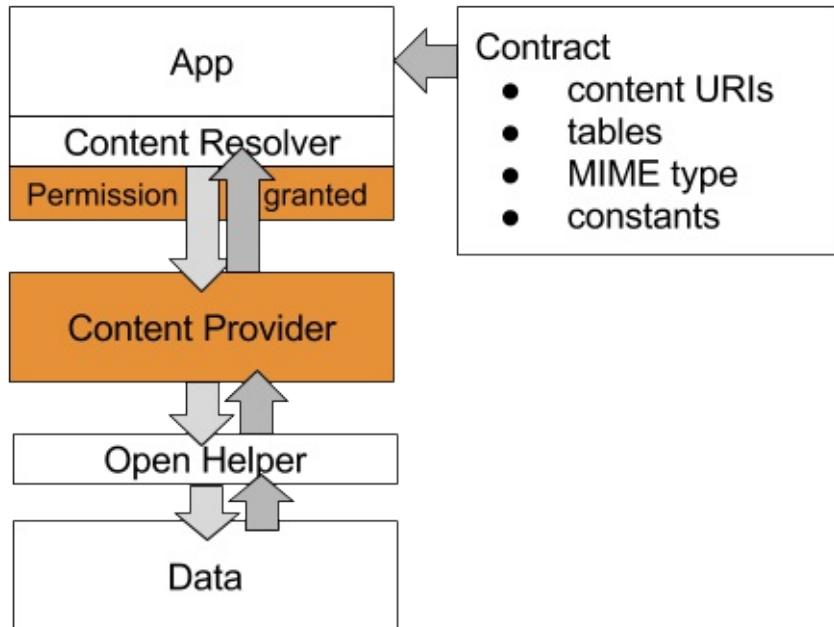
- A content provider that can insert, delete, update, and query the database.
- A contract and permissions set that allow other apps to access this content provider.
- A content resolver that interacts with the content provider to insert, delete, update, and query data.
- Unchanged user interface and functionality.

Your app will look that same as at the end of the data storage practical.

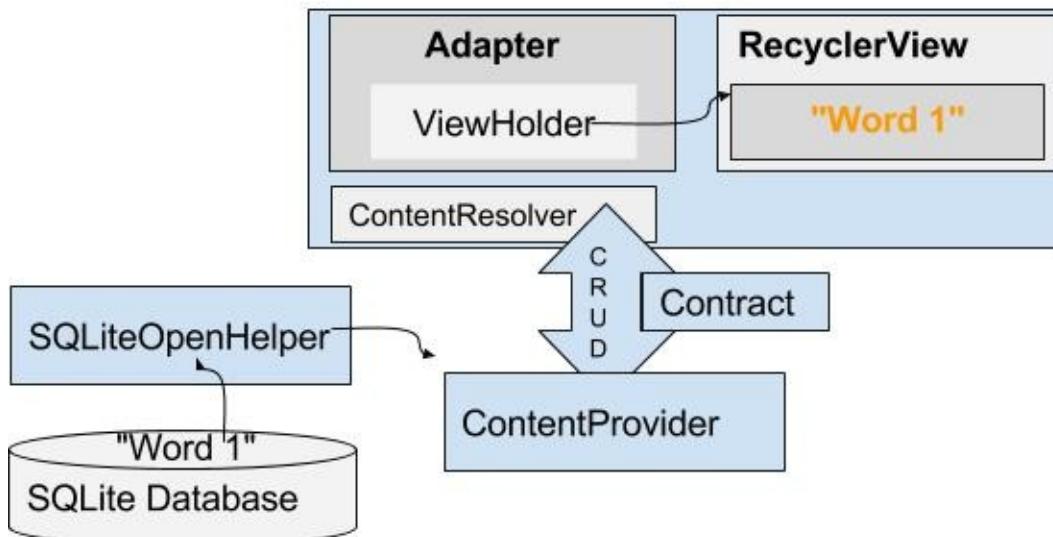


App component overview

The following diagram shows an overview of the components of an app that uses a SQLiteDatabase with a content provider. The only difference from the minimal content provider app is that the content provider fetches the data from a database through an open helper.



The diagram below shows the architecture of the WordListSQLInteractive app with a content provider added; this is the WordListSQLWithContentProvider app that you will build in this practical.



See the concepts chapter for a detailed explanation of all the components and how they interact.

Changes overview

This is a summary of the changes you will make to WordListInteractive to add a content provider.

- New Classes: Contract, ContentProvider, ContentResolver
- Classes that change: WordListOpenHelper, MainActivity, WordListAdapter
- Classes that should not change: WordItem, MyButtonOnClickListener, ViewHolder

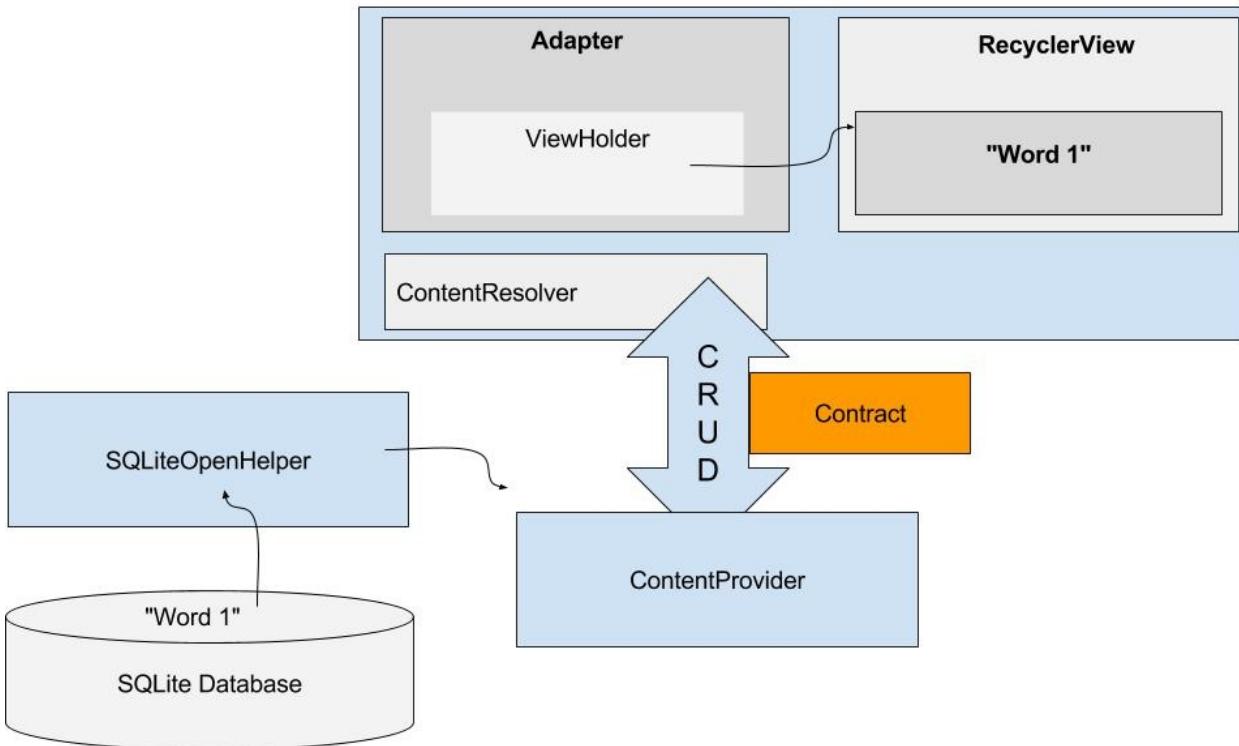
Task 1. Download and run the base code

This practical builds on the WordListSQLInteractive and MinimalistContentProvider apps that you built previously. You will extend a copy of WordListSQLInteractive. You can start from your own code, or download the apps.

- [WordListSQLInteractive](#)
- [MinimalistContentProvider](#)
- Make a copy of WordListSQLInteractive and load it into Android Studio.
- Change the package name to `wordlistsqlwithcontentprovider` .

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 class contains all the information that another app needs to use your app's content provider. You could name the class anything, but it is customarily called "Contract".

```
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 and make it public.

```
public static final String DATABASE_NAME = "wordlist";
```

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. These inner classes are not required, but can help your database work well 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 name, as well as KEY_ID and KEY_WORD column names 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.

Use **File > Settings > Editor > General > Auto Import** on Windows/Linux or **Android Studio > Preferences >Editor >General > Auto Import** on Mac to configure automated 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

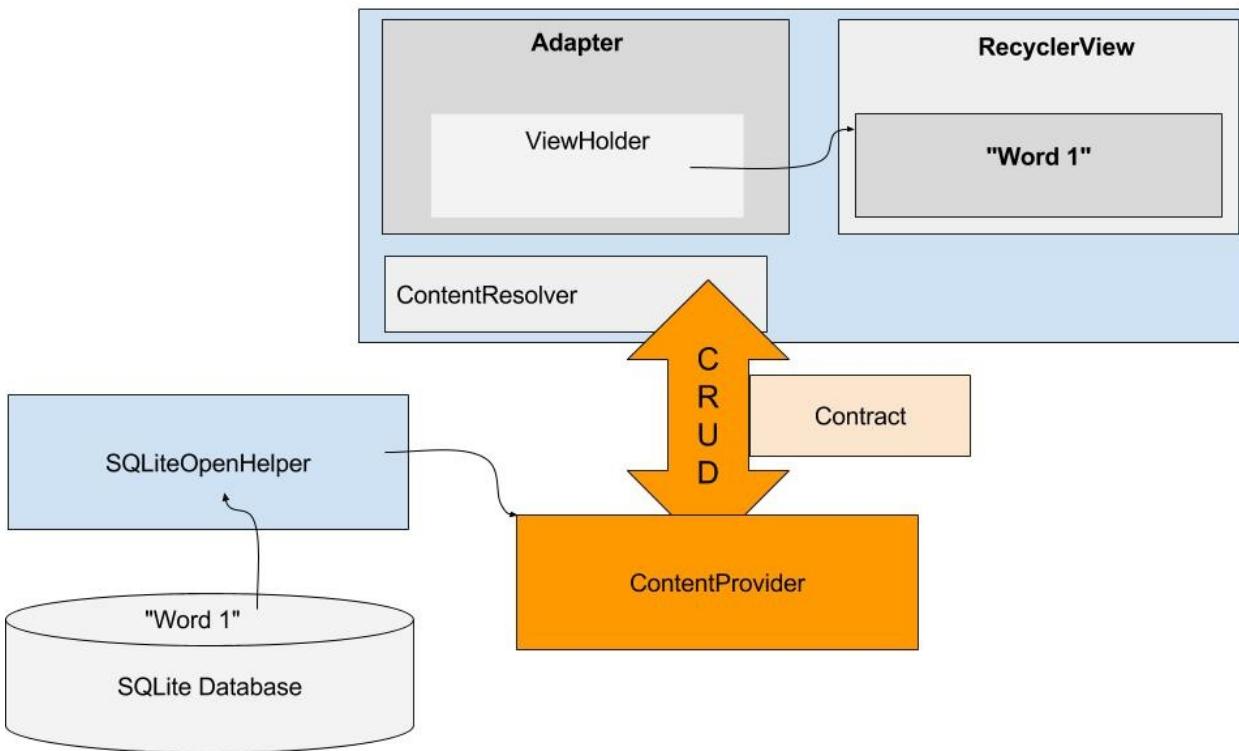
The [MIME type describes](#) the type and format of data. The MIME types is used to process the data appropriately. Common MIME types include `text/html` for web pages, and `application/json`. Read more about [MIME types for content providers](#) in the Android documentation.

1. Declare MIME types for single and multiple record responses:

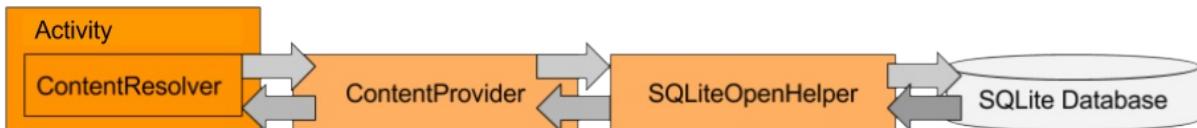
```
static final String SINGLE_RECORD_MIME_TYPE =
    "vnd.android.cursor.item/vnd.com.example.provider.words";
static final String MULTIPLE_RECORDS_MIME_TYPE =
    "vnd.android.cursor.item/vnd.com.example.provider.words";
```

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 WordListAdapter will use a content resolver to query the content provider, which in turn will query WordListOpenHelper which will query the database.



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);
```

5. Declare a WordListOpenHelper class variable, mDB.

```
private WordListOpenHelper mDB;
```

6. 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;
```

7. 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;
}
```

8. Create a private void method initializeUriMatching().

9. 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. Use a Switch statement for 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.

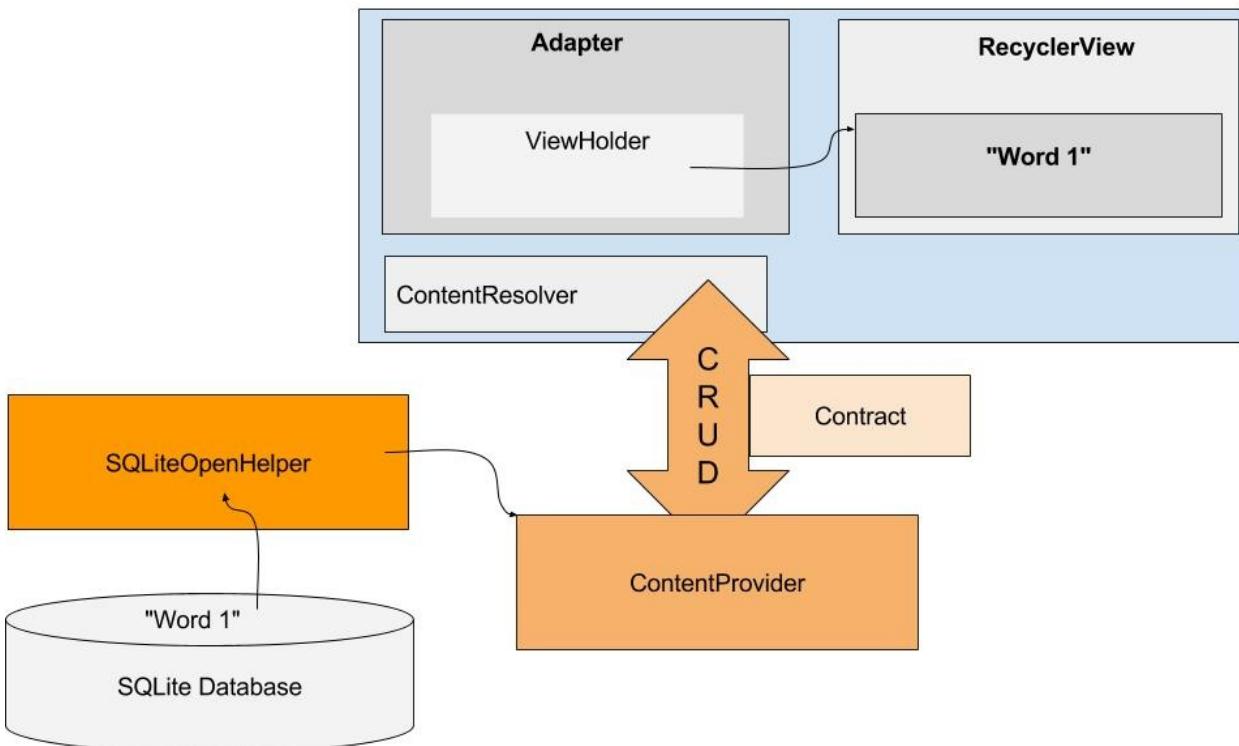
Notice how assigning the results from mDB.count() to a cursor generates an error, because WordListOpenHelper.count() returns a long.

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: " + uri);  
            break;  
        default:  
            // You should do some error handling here.  
            Log.d(TAG, "INVALID URI - URI NOT RECOGNIZED: " + uri);  
    }  
    return cursor;  
}
```

3.3 Fix WordListOpenHelper.query() to return a Cursor and handle returning all items



Since the content provider works with cursors, you can simplify the `WordListOpenHelper.query()` method to return a cursor.

1. Add code with a query to return all items from the database to handle the `cursor = mDB.query(ALL_ITEMS)` case from the above switch statement.
2. Simplify `WordListOpenHelper.query()` to return a cursor.

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 fix the errors one by one.

Solution:

```

/**
 * Queries the database for an entry at a given position.
 *
 * @param position The Nth row in the table.
 * @return a WordItem with the requested database entry.
 */
public Cursor query(int position) {
    String query;
    if (position != ALL_ITEMS) {
        position++; // Because database starts counting at 1.
        query = "SELECT " + KEY_ID + "," + KEY_WORD + " FROM "
            + WORD_LIST_TABLE
            +" WHERE " + KEY_ID + "=" + position + ";";
    } else {
        query = "SELECT * FROM " + WORD_LIST_TABLE
            + " ORDER BY " + KEY_WORD + " ASC ";
    }

    Cursor cursor = null;
    try {
        if (mReadableDB == null) {
            mReadableDB = this.getReadableDatabase();
        }
        cursor = mReadableDB.rawQuery(query, null);
    } catch (Exception e) {
        Log.d(TAG, "QUERY EXCEPTION! " + e);
    } finally {
        return cursor;
    }
}

```

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;
}

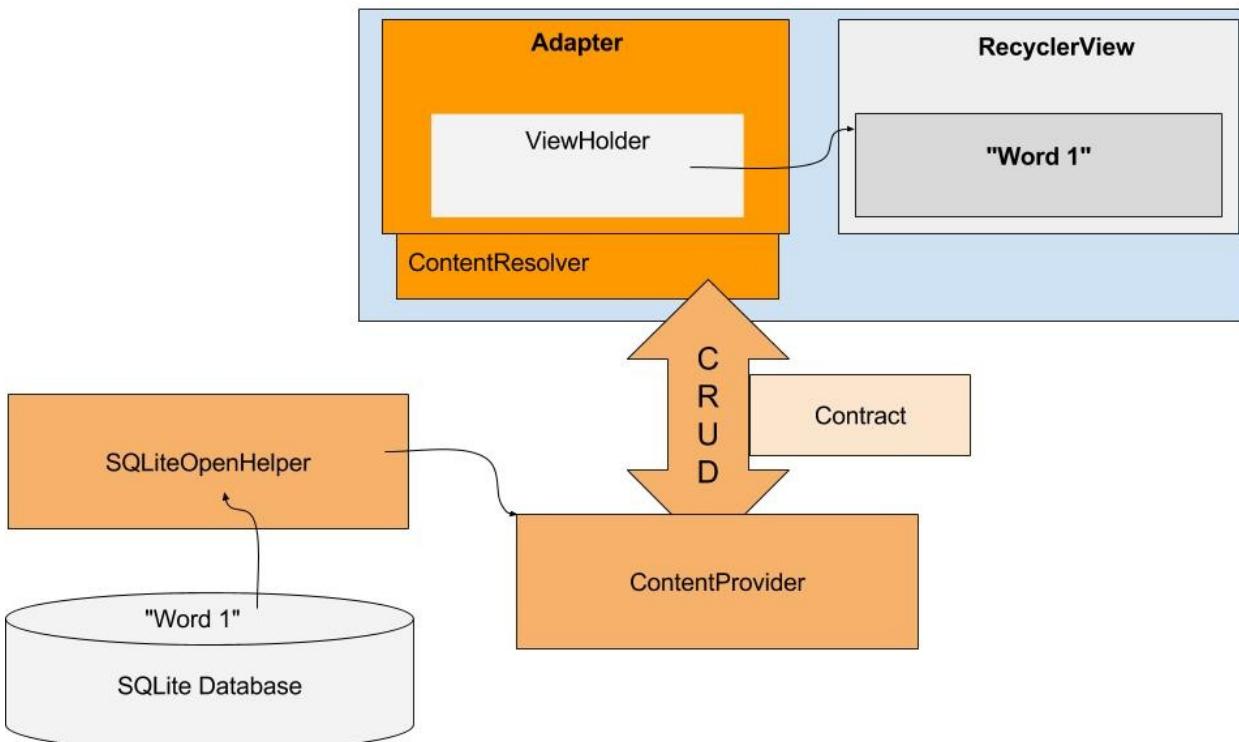
```

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

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 directly referencing the

database anymore. This shows errors in Android Studio that will guide your subsequent changes.

2. In the constructor, delete the assignment to mDB.
3. Refactor > Change the signature of the constructor and remove the db parameter.
4. Add instance 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.
 - WordItem current = mDB.query(position);
 - holder.wordItemView.setText(current.getWord());
6. Define an empty String variable named word.
7. Define an integer variable called id and set it to -1.
8. 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 word = "";
int id = -1;

Cursor cursor = mContext.getContentResolver().query(Uri.parse(
    queryUri), null, null, null, sortOrder);
```

9. 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.

- Implement any referenced string resources.

```

if (cursor != null) {
    if (cursor.moveToPosition(position)) {
        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(R.string.error_no_word);
    }

    cursor.close();
}

} else {
    Log.e (TAG, "onBindViewHolder: Cursor is null.");
}

```

1. Fix the parameters for the click listeners for the two buttons:

- current.getId() ⇒ id
- current.getWord() ⇒ word

The updated click listener for the DELETE button looks like this:

```

@Override
public void onClick(View v) {
    selectionArgs = new String[] {Integer.toString(id)};
    int deleted = mContext.getContentResolver().delete(
        Contract.CONTENT_URI, Contract.CONTENT_PATH, selectionArgs);
    if (deleted > 0) {
        // Need both calls
        notifyItemRemoved(h.getAdapterPosition());
        notifyItemRangeChanged(
            h.getAdapterPosition(), getItemCount());
    } else {
        Log.d (TAG, mContext.getString(R.string.not_deleted) + deleted);
    }
}

```

2. Replace the call to mDB.delete(id) in the DELETE button callback 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 Change WordListAdapter.getCount() to use a content resolver

Instead of requesting the count from the database, getCount() has to connect to the content resolver and request the count. In the Contract, you defined a URI for getting that count:

```
public static final String COUNT = "count";
public static final Uri ROW_COUNT_URI =
    Uri.parse("content://" + AUTHORITY + "/" + CONTENT_PATH + "/" + COUNT)
```

Change WordListAdapter.getCount() to:

- Use a content resolver query to get the item count
- Use the ROW_COUNT_URI in your query
- The count is an integer type and is the first element of the returned Cursor
- Extract `count` from the cursor and return it
- Return -1 otherwise
- Close the cursor

Use the code you just wrote for onBindViewHolder as a guideline.

Solution:

```
@Override
public int getItemCount() {
    Cursor cursor = mContext.getContentResolver().query(
        Contract.ROW_COUNT_URI, new String[] {"count(*) AS count"},
        selectionClause, selectionArgs, sortOrder);
    try {
        cursor.moveToFirst();
        int count = cursor.getInt(0);
        cursor.close();
        return count;
    } catch (Exception e){
        Log.d(TAG, "EXCEPTION getItemCount: " + e);
        return -1;
    }
}
```

3.7 Add the content provider to the Android Manifest

1. Run your app.
2. Examine logcat for the (very common) cause of the error.
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.

3.8 What's next?

- 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 will be a lot less work.

Task 4. Implement 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:

1. If the word length is not zero, create a ContentValues variable named "values" and add the user-inputted word to it using the string "word" as a key.
2. Replace mDB.insert(word); with an insert request to a to a content resolver.

Updating:

1. 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(Contract.WordList.KEY_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(Contract.WordList.KEY_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 your app still doesn't work, you should correct any issues. You will need the working code in a later practical. In that lesson you will write an app that uses this content provider to load word list data into its user interface.

Solution code

Android Studio project: [word_list_sql_with_content_provider](#)

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

- 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.

Summary

- In production, most application developers will typically refactor apps to accommodate a content provider.
- During refactoring, developers will typically experience cascading changes and errors
- You need to separate the UI from the database using a content provider and a content resolver
- The UI should not change during the refactor from an embedded database to an external data source.
- The Contract class defines the common constants for all the components in the refactored application.
- The Contract class localizes all the common constants for ease of maintenance.
- When refactoring, it is very useful to have diagrams of the database access classes
- Careful thought should be given to designing the database access URIs that other applications need to access the data.
- All access to your database must be changed to use a Content Resolver instead of directly accessing a helper class (for example: WordListOpenHelper)
- If the underlying data has changed, it is important to signal the UI to refresh using `notifyDataSetChanged()`.

Related concepts

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Content Providers](#)

Learn more

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](#)

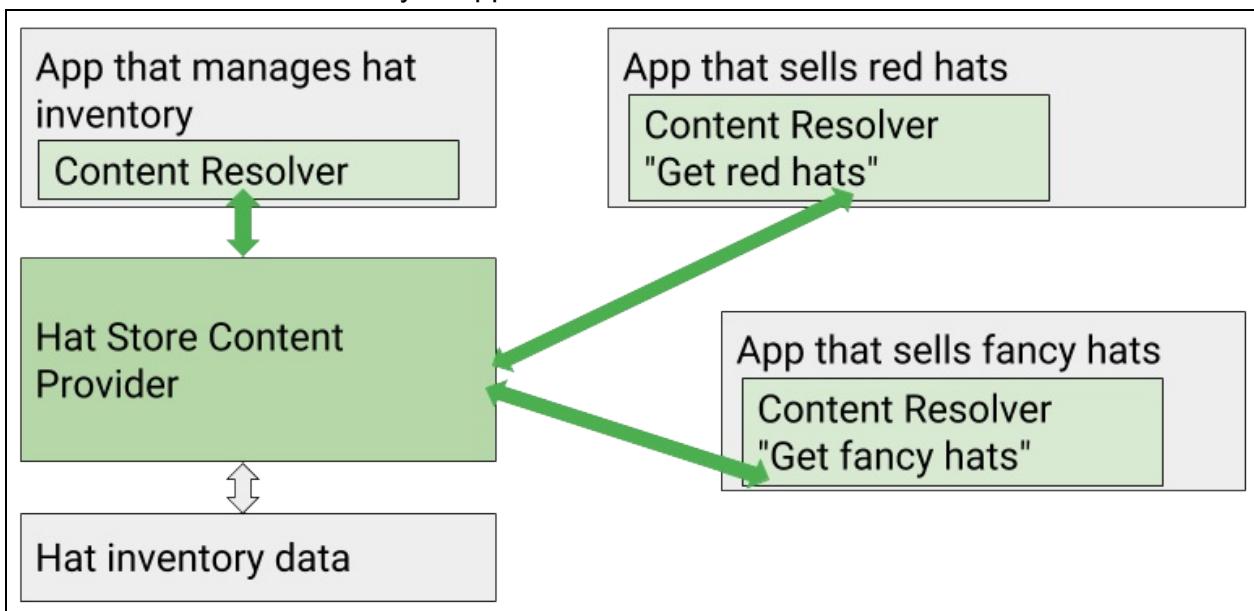
11.1C: Sharing content with other apps

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
- Related concept
- Learn more

To protect app and user data, apps cannot share data with other apps directly. However, apps can make data available to other apps by using a content provider. Client apps can then use a content resolver to access the data via the content provider's public interface.

The following diagram shows how a hat wholesaler might use a content provider to share information about its inventory to apps that sell hats.



In this practical you will modify `WordListSQLWithContentProvider` to allow other apps to access the data in its content provider. Then you will create a second app, `WordListClient`, that has no data of its own, but instead, fetches data from `WordListSQLWithContentProvider`'s content provider.

What you should already KNOW

For this practical you should be familiar with:

- Content providers and resolvers
- How to rename packages
- The WordListSQLWithContentProvider app from the previous practical

What you will LEARN

You will learn how to:

- Set permissions, so other apps can use your app's content provider.
- Build a client app that fetches data from your app's content provider.

What you will DO

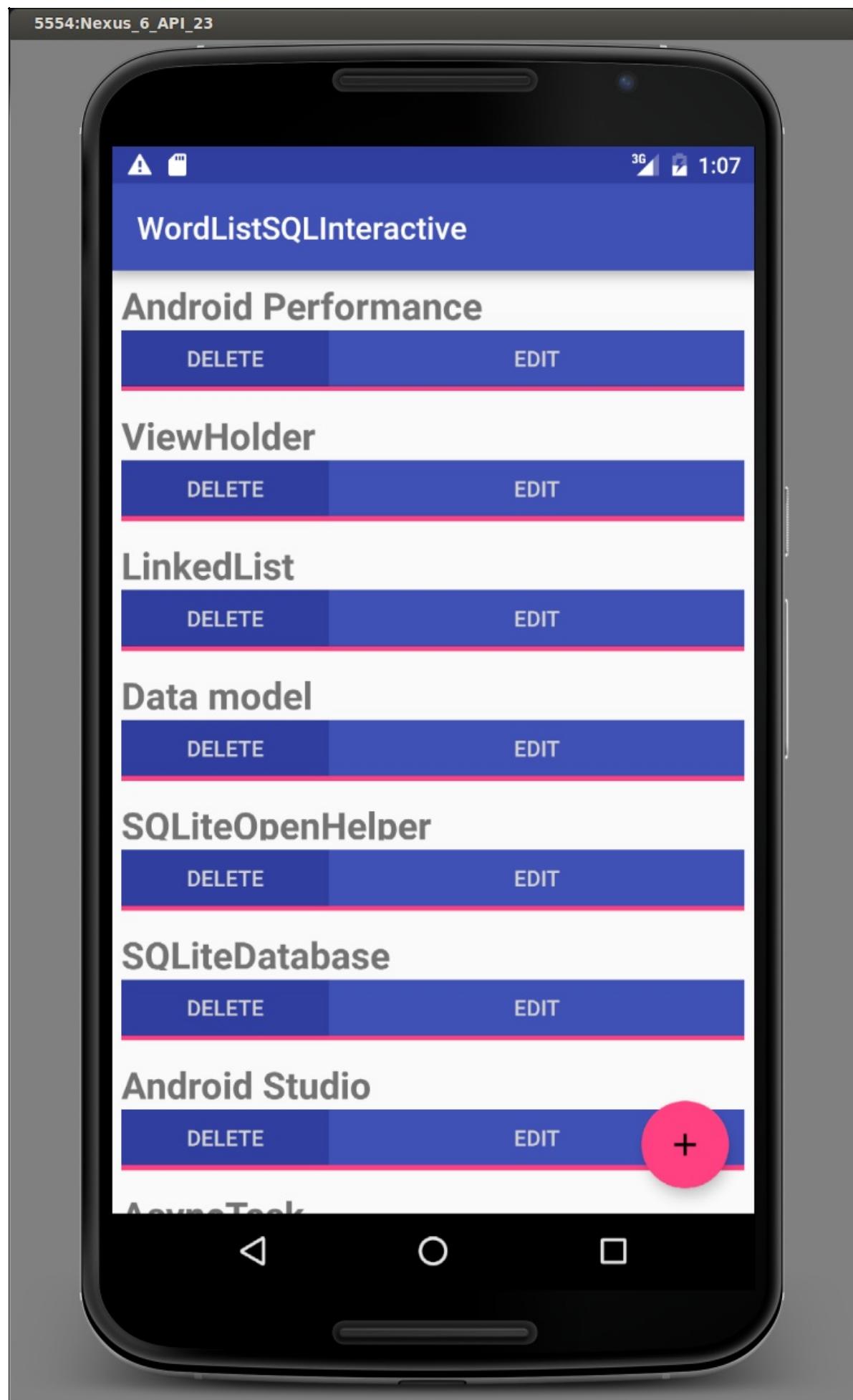
You will:

- Enable WordListSQLWithContentProvider to share its data.
- Create a client app that gets data from the content provider of WordListSQLWithContentProvider.

Apps Overview

You will use two apps in this practical.

- The existing WordListSQLWithContentProvider app that you built in the previous practical.
- A new WordListClient app that will query the content provider of WordListSQLWithContentProvider. The UI for this app is the same as WordListInteractive.



Task 1. Make your content provider available to other apps

By default, apps cannot access the data of other apps.

To make your content provider available to other apps, you must specify permissions in the AndroidManifest of your app. This is true for any app that has a content provider. Each content provider needs permissions specified in its AndroidManifest.

Permissions are not covered in detail in these practicals. You can learn more in [Implementing Content Provider Permissions](#).

1.1. Modify WordListWithContentProvider to allow apps access

1. Open [WordListSQLWithContentProvider](#) in Android Studio.
2. Open the AndroidManifest.xml file.
3. Add an export statement inside the `<provider>` .

```
    android:exported="true"
```

4. At the top level, inside the `<manifest>` tag add a permission for the content provider.

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" />
```

5. Run the app to make sure there are no errors, and leave it installed on the device.

In order for another app to access WordListWithContentProvider's content provider, the app with the content provider has to be installed on the device. It is not necessary for it to be running.

You now have a content provider on your device that another app can access. Next, you are going to build an app, WordListClient, that gets words from the content provider and displays them.

1.2. Create the WordListClient app

Instead of building a client app from scratch, you will create WordListClient from a copy of [WordListSQLWithContentProvider](#). You will keep the user interface and the adapter to display the data. You will remove the content provider and the database, and instead get data from the content provider of WordListSQLWithContentProvider.

1. Create a copy of the [WordListSQLWithContentProvider](#) **folder** and call it WordListClient.
2. Open the copied app in the WordListClient folder in Android Studio.
3. Rename the package (Refactor > Rename) to wordlistclient.
4. Open build.gradle(Module:app) and change the app id to wordlistclient.
5. In strings.xml, change the app name to WordListClient.
6. In the Android Manifest of WordListClient, remove the `<provider>` declaration as there won't be a provider in WordListClient.
7. At the top level, inside the `<manifest>` tag, add a `<uses-permission>` permission to use the content provider of WordListSQLWithContentProvider.

```
<uses-permission android:name = "com.android.example.wordlistsqllwithcontentprovider.PERMISSION"/>
```

8. Delete the WordListContentProvider class, because the app will access the content provider of WordListSQLWithContentProvider.
9. Delete the WordListOpenHelper class, because your app does not need a database or an open helper of its own.
10. Look at MainActivity and WordListAdapter. Note that the code for inserting, deleting, and updating words remains unchanged, calling the content provider of WordListSQLWithContentProvider using the URIs specified in the Contract class.
11. Run WordListClient.
 - In spite of not having data of its own, WordListClient displays data, which is the data it fetches from the WordListSQLWithContentProvider app's content provider.
12. Insert a few words on WordListClient.
13. Start WordListSQLWithContentProvider.
 - Notice that the word you inserted with WordListClient, displays also in WordListSQLWithContentProvider, because they share one data source.
 - Delete the word in WordListSQLWithContentProvider and notice the word also being deleted from the display of WordListClient. (You may need to scroll to make the change appear.)
14. As you interact with other the other app, changes made by one app are reflected in the other app.

In the previous hat store example, the warehouse owner can update the hat inventory with new red or fancy hats, and the store apps will immediately be able to show these new hats to their customers. And if the red hat store sells all the red fancy hats, the fancy hat store will know that the inventory of fancy red hats is sold out.

Solution code

Android Studio project: [WordListClient](#)

Summary

- Your app can permit other apps to interact with its content provider and get or edit data.
- You can build a client app that instead of providing its own data, fetches data from another app's content provider.
- You can separate management of the data from displaying the data. That is, you can have one app that manages the content provider, and many client apps that use the data provided by the content provider.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Content Providers](#)

Learn more

Developer Documentation:

- [Working with System Permissions](#)
- [Implementing Content Provider Permissions](#)

12.1: 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. Create the base app for WordListLoader
- Task 2: MainActivity: Adding a LoaderManager and LoaderCallbacks
- Task 3: WordListAdapter: Implement setData(), getItemCount(), and onBindViewHolder()
- Summary
- Related concept
- Learn more

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.

Asking a [ContentProvider](#) for data you want to display may take time. If you request data from the content provider from an [Activity](#) (and run it on the UI thread), the app may get blocked long enough to cause a visible delay for the user, and the system may even issue an "Application Not Responding" message. Therefore, you should load data on a separate thread, in the background, and display the results after loading is finished.

To run a query on a separate thread, you use loader that runs asynchronously in the background and reconnects to the [Activity](#) when finished. Specifically, [CursorLoader](#) runs a query in the background, and automatically re-runs it when data associated with the query changes.

You have used an [AsyncTaskLoader](#) in a previous practical. CursorLoader extends AsyncTaskLoader to work with content providers.

At a high level, you need the following pieces to use a loader to display data from a content provider:

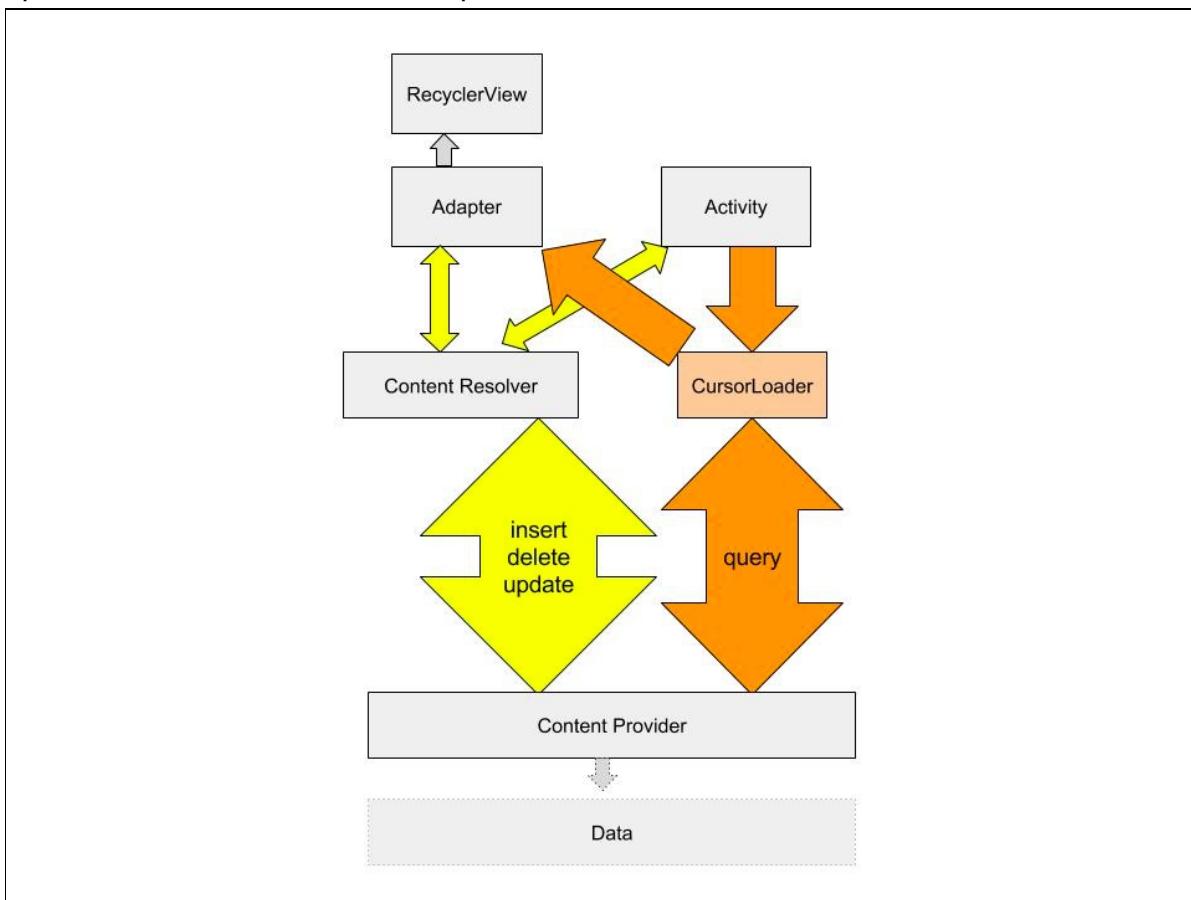
- An [Activity](#) or fragment.
- An instance of the [LoaderManager](#) in the Activity.
- A [CursorLoader](#) to load data backed by a [ContentProvider](#).
- An implementation for [LoaderManager.LoaderCallbacks](#), an abstract callback interface

for the client to interact with the LoaderManager.

- A way of displaying the loader's data, commonly using an adapter. For example, you could display the data in a RecyclerView.

The following diagram shows a complete app architecture with a loader.

- The loader performs querying for items in the background. If the data changes, it automatically gets a new set of data for the adapter.
- The insert, delete, and update operations do not use the loader. However, after the data changes because of an insert, delete, or update operation, the loader fetches the updated data and notifies the adapter.



What you should already KNOW

For this practical you should be able to:

- Display data in a RecyclerView.
- Work with simple Adapters.
- Understand Cursors (see previous practical and concepts).
- Work with AsyncTaskLoader.
- Understand how to work with a Content Providers.

What you will LEARN

You will learn to:

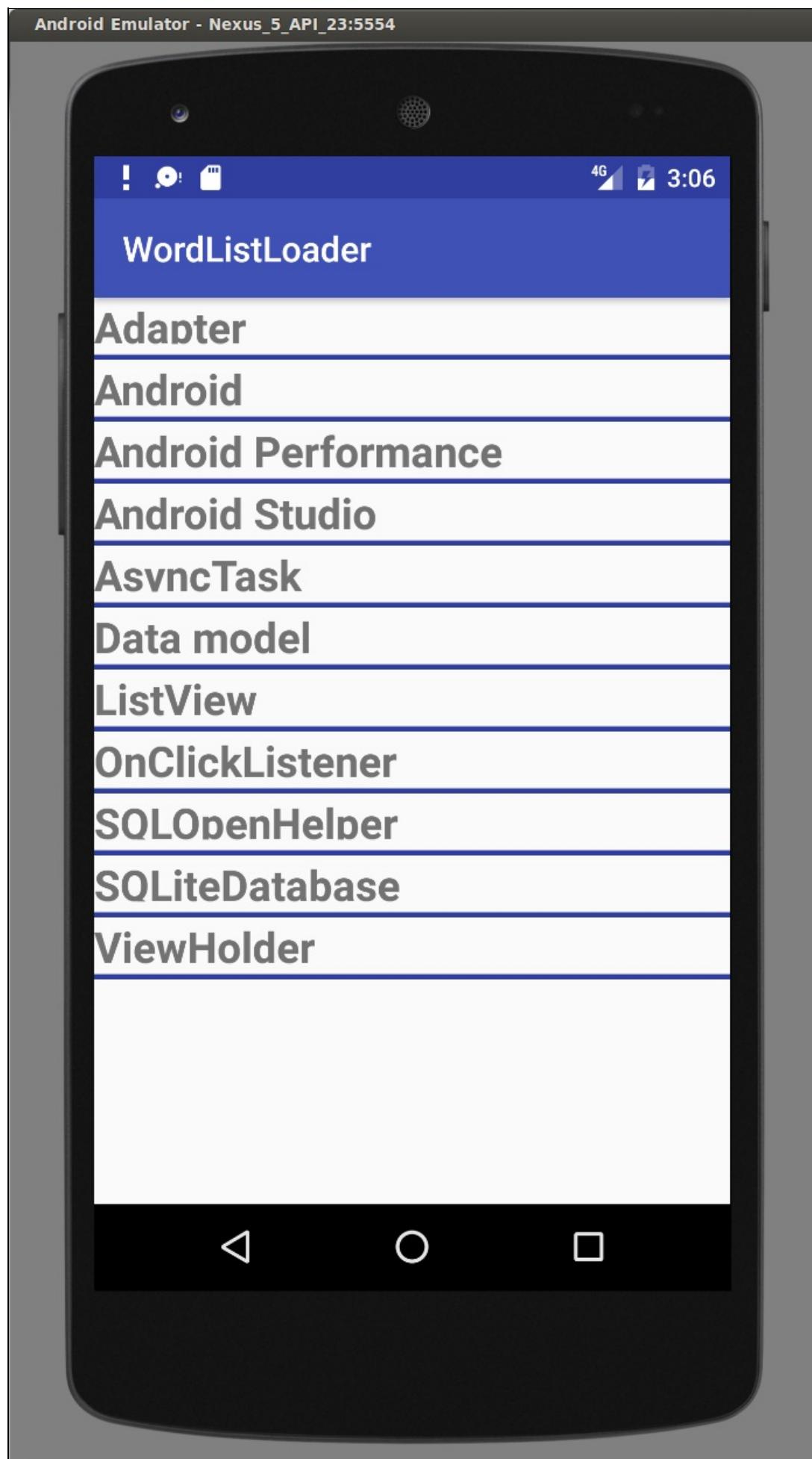
- Load data from a content provider using a CursorLoader.
- Use code from a finished app to quickly build a new app with related functionality.

What you will DO

- You will create a basic app that uses a CursorLoader to query the content provider of WordListSQLWithContentProvider and display the data in a RecyclerView.
- Use WordListClient as a reference for some of the code. In particular, you can reuse the Contract and WordItem Classes, as well as parts of the MainActivity and WordListAdapter classes.
- The app you will create will have a very basic UI. Unlike WordListClient, it will not have insert, delete, or update functionality.

App Overview

Using WordListClient from the previous practical as a source for some of the code, you will create a new app, WordListLoader that loads and displays data from the content provider for WordListSQLWithContentProvider. The following screenshot shows how the finished app will display the words.



IMPORTANT:

- You must install a [WordListWithContentProvider](#) app that shares its content provider so that there is a content provider available for WordListLoader.
- Use the [WordListClient](#) app that you built in the previous practical as a reference and to reuse code.

Task 1. Create the base app for WordListLoader

In this task you will create a project and parts of the app that are not specific to loaders. You need the WordListClient app loaded in Android Studio, so you can copy code from it.

1.1 Create a project with Contract and WordListItem classes and layout files

1. Start Android Studio and load the finished WordListClient app from the previous practical.
2. Create a new project with the Empty Activity template and call it WordListLoader.
3. Add the permission for WordListSQLWithContentProvider's content provider to the Android Manifest.

```
<uses-permission android:name =  
    "com.android.example.wordlistsqlwithcontentprovider.PERMISSION"/>
```

4. Create a new Java class and call it Contract.
5. Copy the Contract class from WordListClient into the new Contract class of WordListLoader. Make sure not to copy the package name.
6. In WordListLoader, create a new Java class and call it WordListItem.
7. Copy the WordItem class from WordListClient into the new WordItem class of WordListLoader.
8. Copy the layout for the recycler view from actity_main.xml from WordListClient to WordListLoader. Remove the floating action button.
9. Create a new layout for WordListItem, wordlist_item.xml.
10. Using wordlist_item.xml from WordListClient as your reference, create a LinearLayout with a single TextView.
 - The id of the TextView must be `android:id="@+id/word"`.
 - Resolve strings, dimensions, and styles that you are reusing. Note that you can copy/paste files between projects. Overwrite the existing XML files in WordListLoader.

- Change the app_name to WordListLoader in strings.xml.

11. At this point, you should see no errors in Android Studio.

1.2 Add a RecyclerView to MainActivity

To display the data, add a RecyclerView to your MainActivity. You can do this on your own, or reuse code from WordListClient.

1. Add the RecyclerView and Coordinator Layout from the support library to your build.gradle file.

```
compile 'com.android.support:recyclerview-v7:24.1.1'  
compile 'com.android.support:design:24.1.1'
```

2. Import the support library versions of RecyclerView and LinearLayoutManager into your MainActivity.

```
import android.support.v7.widget.LinearLayoutManager;  
import android.support.v7.widget.RecyclerView;
```

3. Create a TAG for MainActivity.
4. Create a private variable mRecyclerView for the RecyclerView.
5. Create a private variable mWordListAdapter for the adapter. This will remain red, until you create the adapter class.
6. In onCreate() of MainActivity, create a RecyclerView, create a WordListAdapter, set the adapter on the Recyclerview, and attach a LinearLayoutManager. See WordListClient for sample code.

```
// Create recycler view.  
mRecyclerView = (RecyclerView) findViewById(R.id.recyclerview);  
// Create an adapter and supply the data to be displayed.  
mAdapter = new WordListAdapter(this);  
// Connect the adapter with the recycler view.  
mRecyclerView.setAdapter(mAdapter);  
// Give the recycler view a default layout manager.  
mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
```

7. If you build your app now, only WordListAdapter should be red. The app does not run yet.

1.3 Create WordListAdapter

Use WordListAdapter from WordListClient and the snippets below as a reference for creating this adapter. If you need a refresher, revisit the RecyclerView chapter of this course.

1. Create a new Java class WordListAdapter that extends RecyclerView.Adapter.

```
public class WordListAdapter  
    extends RecyclerView.Adapter<WordListAdapter.WordViewHolder> {}
```

Using WordListAdapter as a reference, add the following:

2. Add an inner ViewHolder class with one TextView, called wordItemView and inflate it from the text view with the id "word".

```
class WordViewHolder extends RecyclerView.ViewHolder {  
    public final TextView wordItemView;  
  
    public WordViewHolder(View itemView) {  
        super(itemView);  
        wordItemView = (TextView) itemView.findViewById(R.id.word);  
    }  
}
```

3. Add a TAG for log messages.

```
private static final String TAG = WordListAdapter.class.getSimpleName();
```

4. Add member variables for the LayoutInflater and the context.

```
private final LayoutInflater mInflater;  
private Context mContext;
```

5. Implement the constructor for WordListAdapter.

```
public WordListAdapter(Context context) {  
    mInflater = LayoutInflater.from(context);  
    this.mContext = context;  
}
```

6. Implement (or copy) the onCreateViewHolder method to inflate a wordlist_item view.

```
@Override  
public WordViewHolder onCreateViewHolder(  
    ViewGroup parent, int viewType) {  
    View mItemView = mInflater.inflate(  
        R.layout.wordlist_item, parent, false);  
    return new WordViewHolder(mItemView);  
}
```

7. Press Alt-Enter on the adapter's class header and "choose implement methods" to

- create method stubs for the `getItemCount()` and `onBindViewHolder()` methods.
8. At this point, there should be no red underlines or words in your code.
 9. Run your app, and it should show a blank activity as shown in the following screenshot, since you haven't loaded any data yet. You will add data in the next task.



Task 2. MainActivity: Adding a LoaderManager and LoaderCallbacks

When you use a loader to load your data for you, you use a loader manager to take care of the details of running the loader.

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.

2.1 Add the Loader Manager

1. Open `MainActivity.java`
2. Extend the class signature to implement `LoaderManager.LoaderCallbacks`. Import 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` from the support library 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, so the second argument is null.
 - And you bind the loader to the current `MainActivity` (`this`).

```
getSupportLoaderManager().initLoader(0, null, this);
```

2.2 Implement `onCreateLoader()`

The `LoaderManager` calls the `onCreateLoader()` method to create the loader, if it does not already exist.

You create a loader by supplying it with a context, and the URI from which to load data—in this case, for content provider of `WordListSQLWithContentProvider`, the URI specified in the Contract.

1. In `onCreateLoader()`, create a `queryUri` and projection. Use the same URI that the content resolver is using to query the content provider. You can find it in the `Contract` and it is used in `WordListClient`.
2. Create and return a new `CursorLoader` from these arguments. Import the `CursorLoader` from the support library.

```
@Override  
public Loader<Cursor> onCreateLoader(int id, Bundle args) {  
    String queryUri = Contract.CONTENT_URI.toString();  
    String[] projection = new String[] {Contract.CONTENT_PATH};  
    return new CursorLoader(this, Uri.parse(queryUri), projection, null, null, null  
);  
}
```

2.3 Implement `onLoadFinished()`

When loading has finished, you need to send the data to the adapter.

1. Call `setData()` in `onLoadFinished()`. The code will turn red, and you will implement it in the next task. The argument for `setData()` is the cursor with "data" returned by the loader when it is finished loading.

```
@Override  
public void onLoadFinished(Loader<Cursor> loader, Cursor data) {  
    mAdapter.setData(data);  
}
```

2.4 Implement `onLoaderReset()`

On resetting the loader, 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);  
}
```

Task 3. `WordListAdapter`: Implement `setData()`, `getItemCount()`, and `onBindViewHolder()`

As your final tasks, you need to implement the `setData()` method referenced above, and implement `onBindViewHolder()` to work with the loader to display the data. Here is how this happens:

- When the loader finishes loading new or changed data in the background, the `onLoadFinished()` method that you implemented in the `MainActivity` is executed.
- `onLoadFinished()` calls `WordListAdapter.setData()`, which updates the `mCursor` variable in the adapter with the loader's latest data and notifies the adapter that the data has changed.
- The adapter updates the UI with the new data, by binding views to the data in `onBindViewHolder()`.

3.1. Implement `setData()`

You need a way to set and store the latest loaded version of the data with the adapter. For this app, the loader returns data as a cursor, so you need to create a Cursor member variable `mCursor` that will always hold the latest data set.

The `setData()` method is called by the loader when it finished loading or is reset, and it needs to update `mCursor`.

1. Create a private member variable of type Cursor. Call it "mCursor" and initialize it to null.
2. Implement the public method `setData()`. It takes a Cursor argument and returns nothing.
3. In the body, set `mCursor` to the passed in Cursor argument and call `notifyDataSetChanged()`, so that the adapter updates the display.

```
public void setData(Cursor cursor) {  
    mCursor = cursor;  
    notifyDataSetChanged();  
}
```

3.2. Implement `getCount()`

Instead of 0, `getCount()` needs to return the number of items in `mCursor`. If `mCursor` is null, return -1.

```
@Override  
public int getCount() {  
    if (mCursor != null) {  
        return mCursor.getCount();  
    } else {  
        return -1;  
    }  
}
```

3.3 Implement `onBindViewHolder()`

In WordListClient, the onBindViewHolder() method uses a content resolver to fetch data from WordListSQLWithContentProvider's content provider. In this app, onBindViewHolder() uses the data provided by the loader and stored in mCursor.

In onBindViewHolder, handle the following situations.

1. If mCursor is null, do nothing, but display a log message. In a real application, you would need to also notify the user in a meaningful way.
2. If mCursor is not null but contains no word, set the text of the TextView to ERROR: NO WORD. Again, in a real application, you would handle this depending on the type of app you have.
3. Otherwise, get the column index for the "word" column (you cannot assume the column is in a fixed location in the cursor's row), and using the index, retrieve the word. Set the text of the text view to the word.

```
@Override  
public void onBindViewHolder(WordViewHolder holder, int position) {  
  
    String word = "";  
  
    if (mCursor != null) {  
        if (mCursor.moveToPosition(position)) {  
            int indexWord = mCursor.getColumnIndex(Contract.WordList.KEY_WORD);  
            word = mCursor.getString(indexWord);  
            holder.wordItemView.setText(word);  
        } else {  
            holder.wordItemView.setText(R.string.error_no_word);  
        }  
    } else {  
        Log.e(TAG, "onBindViewHolder: Cursor is null.");  
    }  
}
```

3.4 Run and test your app

Your WordListLoader app should exactly work the same as the WordListClient app for displaying a list of words. To test your app, do the following.

1. Make sure WordListSQLWithContentProvder is installed on the device, so that your app has a content provider to load from. Otherwise, your app will display a blank text view.
2. Run WordListLoader. You should see the same list of words as in WordListSQLWithContentProvider.

Solution code

Android Studio project: [WordListLoader](#)

Summary

- In this chapter, you learned how to use a loader to load data from a content provider that is not part of your app.

Related concept

The related concept documentation is in [Android Developer Fundamentals: Concepts](#).

- [Loaders](#)

Learn more

- [Loaders](#)
- [Running a query with a CursorLoader](#)
- [CursorLoader](#)

Appendix: Utilities

Table of Contents:

- [Copy and rename a project](#)
- [Delete a project](#)
- [Extract Resources](#)
- [Add support libraries](#)
- [Create images in Asset Studio](#)
- [Compare custom objects](#)
- [Save state of custom objects](#)

This appendix is a collection of tasks you may need to do during development of the apps in the practicals. They are not specific to one practical.

Copy and rename a project

For some 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 existing project (**ExistingProject**), and then rename and refactor the new project's (**NewProject**) components to use the new project's name. (In the instructions below, substitute your actual project names for **ExistingProject** and **NewProject**.

1. Copy the project

1. On your computer's file system (not in Android Studio), make a copy of the **ExistingProject** directory.
2. Rename the copied directory to **NewProject** .

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. Change the file and the package references in your app to the new name, as follows:

1. Start Android Studio
2. Click **Open an existing Android Studio project**.
3. Navigate to the **NewProject** directory, select it, and click **OK**.

4. Select **Build > Clean Project** to remove the auto-generated files.
5. Click the **1:Project** side-tab and choose **Android** from the drop-down menu to see your files in the Project view.
6. Expand **app > java**.
7. Right-click **com.example.android.existingproject** and choose **Refactor > Rename**. This opens the Rename dialog.
8. Change **existingproject** to **newproject**.
9. Check **Search in comments and strings** and **Search for text occurrences** and click **Refactor**.
10. The **Find Refactoring Preview** pane appears, showing the code to be refactored.
11. Click **Do Refactor**.
12. Expand **res > values** and double-click the **strings.xml** file.
13. 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 manually 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.

1. Expand **app > manifests** and double-click **AndroidManifest.xml**.
2. Find the statement below, and if necessary, change the label if 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, delete its folder.

Android Studio also keeps a list of recent projects that you have opened. You can delete a project from the list of recent projects in Android Studio. (Deleting a project from the recent projects list does not affect the actual project files.)

To remove a project from the recent projects list, do one of the following:

- On the Android Studios startup screen screen, click the name of the project and press the delete key.
- Select File > Open Recent > Manage Projects, click the name of the project and press the delete key.

Extract Strings and Dimensions

Extracting Strings

In order for your app to be translatable into multiple languages you must keep all of your string resources in the res/values/strings.xml file.

Creating 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, such as the text attribute of a TextView:

1. 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.

2. Make sure your cursor is in the highlighted text.

3. Press **Alt + Enter** and select **Create string value resource**.

4. Enter your desired string and press **OK** and the string gets added to your strings.xml file.

- You can select any existing, hard-coded string in either XML or Java, press **Alt + Enter**, and select **Extract string resource**.

Accessing string resources:

- In XML, references string resource using the following syntax: `@string/string_name`
- In Java, reference string resources using the following syntax:
`getString(R.string.string_name)`

2. Extract Dimensions

Dimensions should in general not be hard-coded but kept in the dimens.xml file. This allows for you to specify different dimensions using [resource qualifiers](#).

Extract dimensions in the same way as strings (Alt-Enter), and they will be stored in the dimens.xml.

3. Extract Styles

If you have several elements that share attributes, you can create a style in the style.xml file. To learn more about styles, see the [Styles and Themes](#) lesson.

To extract existing attributes into a style, do the following:

1. Place your cursor in the view whose attributes you want to turn into a style.
2. Right click and select **Refactor > Extract > Style**.
3. Name the style and select attributes. If **Launch 'Use Style Where Possible'** **refactoring after the style 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**.

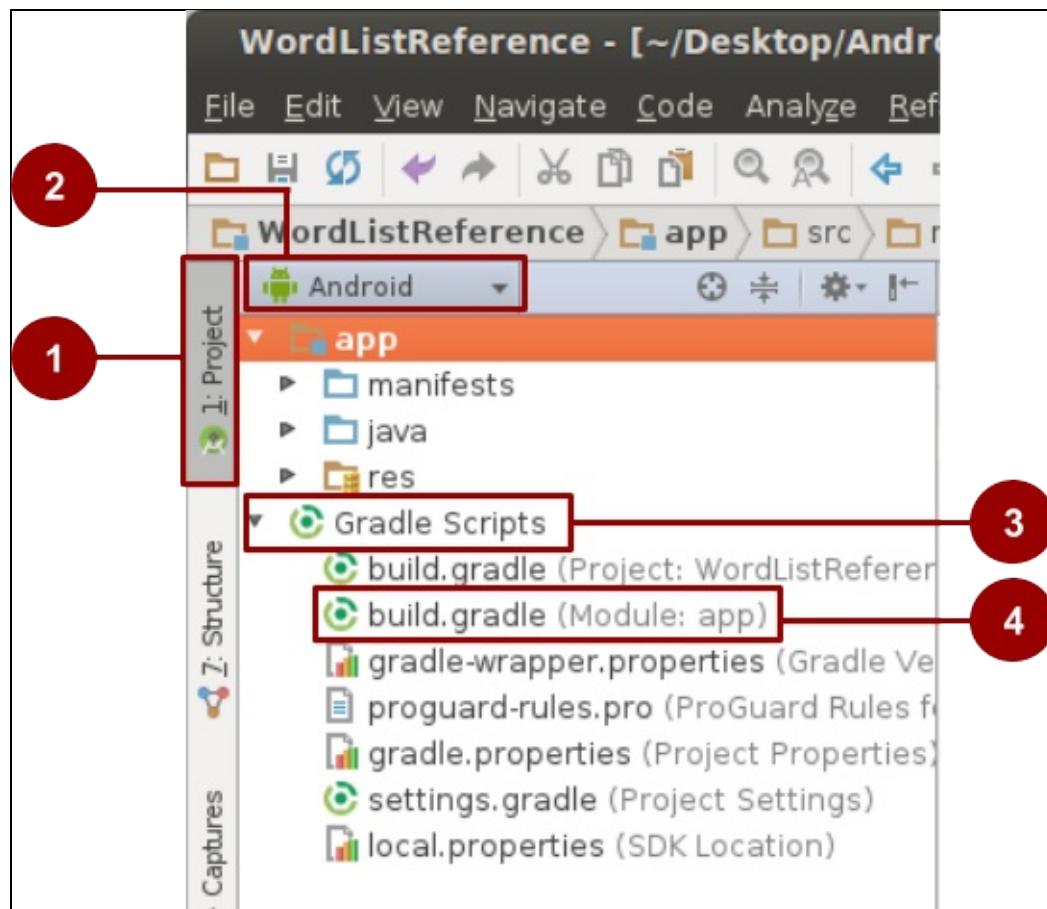
Add Android support libraries to the build file

[Android Support Libraries](#) provide backward-compatible versions of Android framework APIs, additional UI components and a set of useful utilities.

For example, to use the RecyclerView class, which is located in the Android Support package, you must include two dependencies in your project's build.gradle file. The process is the same for other support library components.

Follow these steps and refer to the screenshot below:

1. In Android Studio, in your project, make sure you are in the **Project** pane (1) and in the **Android** view (2).
2. In the hierarchy of files, find the **Gradle Scripts** folder (3).
3. Expand **Gradle Scripts**, if necessary, and open the **build.gradle (Module: app)** file (4).



4. Towards the end of the **build.gradle (Module: app)** file, find the dependencies section.
5. Add these two library 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 similar to this one:

```
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.

The following is an example of the dependencies section of the build.gradle file with support libraries added.

```

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'
}

```

Create images in Asset Studio

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

1. Open your 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.

To add a custom text icon:

1. Change the **Name** 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 "Hello World!" 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, as well as 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, with 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 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.

To add a clipart icon:

Follow the previous steps except:

1. Change the **Name** 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.

To add a custom icon:

Follow the previous steps except:

1. Change the **Name** 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 on your computer.

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.

The [Comparable](#) interface allows you to specify how to compare two objects and determine whether one is bigger, smaller, or the same as the other.

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.

For example, to compare a list of books by publication date:

```
@Override  
public int compareTo(Book book) {  
    if (this.publication == book.publication) { return 0; }  
    else { return this.publication > book.publication ? 1 : -1; }  
}
```

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.