

# Table of Contents

<a href="#">Introduction</a>	1.1
<a href="#">Unit 1: Expand the user experience</a>	1.2
<a href="#">Lesson 1: Fragments</a>	1.2.1
<a href="#">1.1: Creating a Fragment with a UI</a>	1.2.1.1
<a href="#">1.2: Communicating with a Fragment</a>	1.2.1.2
<a href="#">Lesson 2: App widgets</a>	1.2.2
<a href="#">2.1: Building app widgets</a>	1.2.2.1
<a href="#">Lesson 3: Sensors</a>	1.2.3
<a href="#">3.1: Working with sensor data</a>	1.2.3.1
<a href="#">3.2: Working with sensor-based orientation</a>	1.2.3.2
<a href="#">Unit 2: Make your apps fast and small</a>	1.3
<a href="#">Lesson 4: Performance</a>	1.3.1
<a href="#">4.1A: Using the Profile GPU Rendering tool</a>	1.3.1.1
<a href="#">4.1B: Using the Debug GPU Overdraw and Layout Inspector tools</a>	1.3.1.2
<a href="#">4.1C: Using the Systrace and dumpsys tools</a>	1.3.1.3
<a href="#">4.2: Using the Memory Profiler tool</a>	1.3.1.4
<a href="#">4.3: Optimizing network, battery, and image use</a>	1.3.1.5
<a href="#">Unit 3: Make your apps accessible</a>	1.4
<a href="#">Lesson 5: Localization</a>	1.4.1
<a href="#">5.1: Using resources for languages</a>	1.4.1.1
<a href="#">5.2: Using the locale to format information</a>	1.4.1.2
<a href="#">Lesson 6: Accessibility</a>	1.4.2
<a href="#">6.1: Exploring accessibility in Android</a>	1.4.2.1
<a href="#">6.2: Creating accessible apps</a>	1.4.2.2
<a href="#">Unit 4: Add geo features to your apps</a>	1.5
<a href="#">Lesson 7: Location</a>	1.5.1
<a href="#">7.1: Using the device location</a>	1.5.1.1
<a href="#">Lesson 8: Places</a>	1.5.2
<a href="#">8.1: Using the Places API</a>	1.5.2.1
<a href="#">Lesson 9: Mapping</a>	1.5.3

---

<a href="#">9.1: Adding a Google Map to your app</a>	1.5.3.1
<a href="#">Unit 5: Advanced graphics and views</a>	1.6
<a href="#">Lesson 10: Custom views</a>	1.6.1
<a href="#">10.1A: Creating a custom view from a View subclass</a>	1.6.1.1
<a href="#">10.1B: Creating a custom view from scratch</a>	1.6.1.2
<a href="#">Lesson 11: Canvas</a>	1.6.2
<a href="#">11.1A: Creating a simple Canvas object</a>	1.6.2.1
<a href="#">11.1B: Drawing on a Canvas object</a>	1.6.2.2
<a href="#">11.1C: Applying clipping to a Canvas object</a>	1.6.2.3
<a href="#">11.2: Creating a SurfaceView object</a>	1.6.2.4
<a href="#">Lesson 12: Animations</a>	1.6.3
<a href="#">12.1: Creating property animations</a>	1.6.3.1
<a href="#">Appendix</a>	1.7
<a href="#">Appendix: Setup</a>	1.7.1
<a href="#">Appendix: Homework</a>	1.7.2

---

# Advanced Android Development — Practicals

This is the *hands-on practical workbook* for [Advanced Android Development](#), a training course created by the Google Developers Training team. This course builds on the skills you learned in the [Android Developer Fundamentals](#) course.

This course is intended to be taught in a classroom, but all the materials are available online, so if you like to learn by yourself, go ahead!

## Prerequisites

The Advanced Android Development course is intended for experienced developers who have Java programming experience and know the fundamentals of how to build an Android app using the Java programming language. This course assumes you have mastered the topics in Units 1 to 4 of the Android Developer Fundamentals course.

Specifically, this course assumes you know how to:

- Install and use Android Studio
- Run apps from Android Studio on both a device and an emulator
- Create and use `Activity` instances
- Use `View` instances to create your app's user interface
- Enable interaction through click handlers
- Create layouts using the Android Studio layout editor
- Create and use `RecyclerView` and `Adapter` classes
- Run tasks in the background
- Save data in Android shared preferences
- Save data in a local SQL database

## Course materials

The course materials include:

- This practical workbook, which guides you through creating Android apps to practice and perfect the skills you're learning
- A concept reference: [Advanced Android Development—Concepts](#)
- [Slide decks](#) for optional use by instructors
- [Source code in GitHub](#) for apps that you create during the practical exercises

## Get ready for this course

The practicals in this book assume that you are using the latest version of Android Studio. Some of the practicals require at least Android Studio 3.0.

See [Appendix: Setup](#) for details.

## What topics are covered?

### Unit 1: Expand the user experience

#### Lesson 1: Fragments

Create a fragment that has its own UI, and enable your activities to communicate with fragments.

#### Lesson 2: App widgets

Create and update app widgets.

#### Lesson 3: Sensors

Learn how to work with sensor data and build an app that detects and responds to device orientation.

### Unit 2: Make your apps fast and small

#### Lesson 4: Performance

Learn how to detect and analyze your app's performance using tools like these:

- Profile GPU Rendering
- Debug GPU Overdraw and Layout Inspector
- Systrace and dumpsys
- Memory Profiler
- Tools to improve the efficiency of your networking, battery use, and data compression

### Unit 3: Make your apps accessible

#### Lesson 5: Localization

Use resources for different languages, and use the device locale to format information.

#### Lesson 6: Accessibility

Explore accessibility in Android, and learn how to make your app more usable for users who use Google TalkBack.

## Unit 4: Add geo features to your apps

### Lesson 7: Location

Enable your app to detect and update the device's current location.

### Lesson 8: Places

Use the Places API to detect the user's location and offer a "picker" showing local places.

### Lesson 9: Mapping

Add a Google Map to your app.

## Unit 5: Advanced graphics and views

### Lesson 10: Custom views

Create a custom view based on an existing `View` class, then create a custom view from scratch.

### Lesson 11: Canvas

Create a simple `Canvas` object, then draw on the canvas and apply clipping regions to it.

Use a `SurfaceView` object to draw to the screen outside of the main UI thread.

### Lesson 12: Animations

Learn how to use property animations to define an animation to change an object property.

*Developed by the Google Developers Training Team*



*Last updated: October 2017*

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

# 1.1: Creating a Fragment with a UI

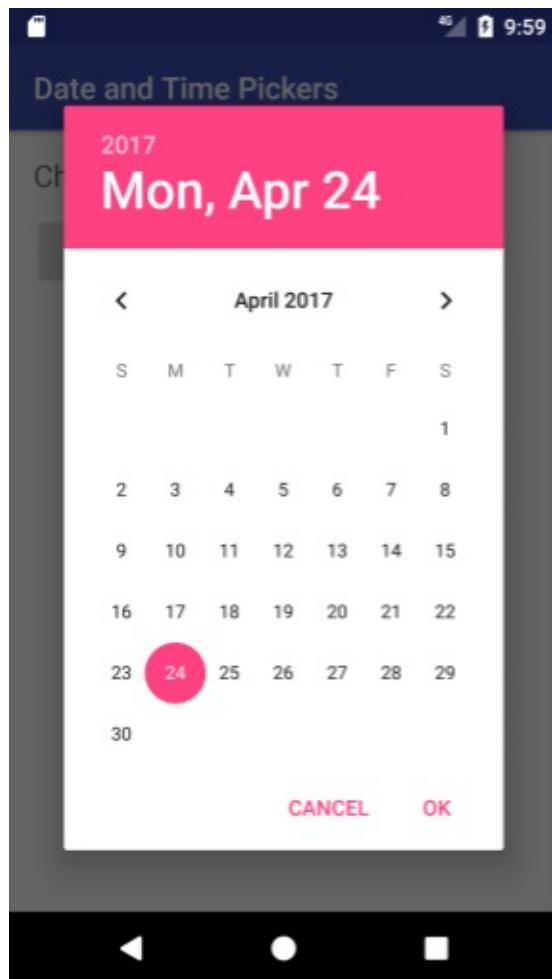
## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Include a fragment for an activity's entire lifecycle
- Task 1 solution code
- Coding challenge
- Task 2. Add a fragment to an activity dynamically
- Task 2 solution code
- Summary
- Related concept
- Learn more

A `Fragment` is a self-contained component with its own user interface (UI) and lifecycle that can be reused in different parts of an app's UI. (A `Fragment` can also be used without a UI, in order to retain values across configuration changes, but this lesson does not cover that usage.)

A `Fragment` can be a *static* part of the UI of an `Activity`, which means that the `Fragment` remains on the screen during the entire lifecycle of the `Activity`. However, the UI of an `Activity` may be more effective if it adds or removes the `Fragment` *dynamically* while the `Activity` is running.

One example of a dynamic `Fragment` is the `DatePicker` object, which is an instance of `DialogFragment`, a subclass of `Fragment`. The date picker displays a dialog window floating on top of its `Activity` window when a user taps a button or an action occurs. The user can



click **OK** or **Cancel** to close the `Fragment`.

This practical introduces the `Fragment` class and shows you how to include a `Fragment` as a static part of a UI, as well as how to use `Fragment` transactions to add, replace, or remove a `Fragment` dynamically.

## What you should already KNOW

You should be able to:

- Create and run apps in Android Studio.
- Use the layout editor to create a UI with a `ConstraintLayout`.
- Inflate the UI layout for an `Activity`.
- Add a set of radio buttons with a listener to a UI.

## What you will LEARN

You will learn how to:

- Create a `Fragment` with an interactive UI.
- Add a `Fragment` to the layout of an `Activity`.

- Add, replace, and remove a `Fragment` while an `Activity` is running.

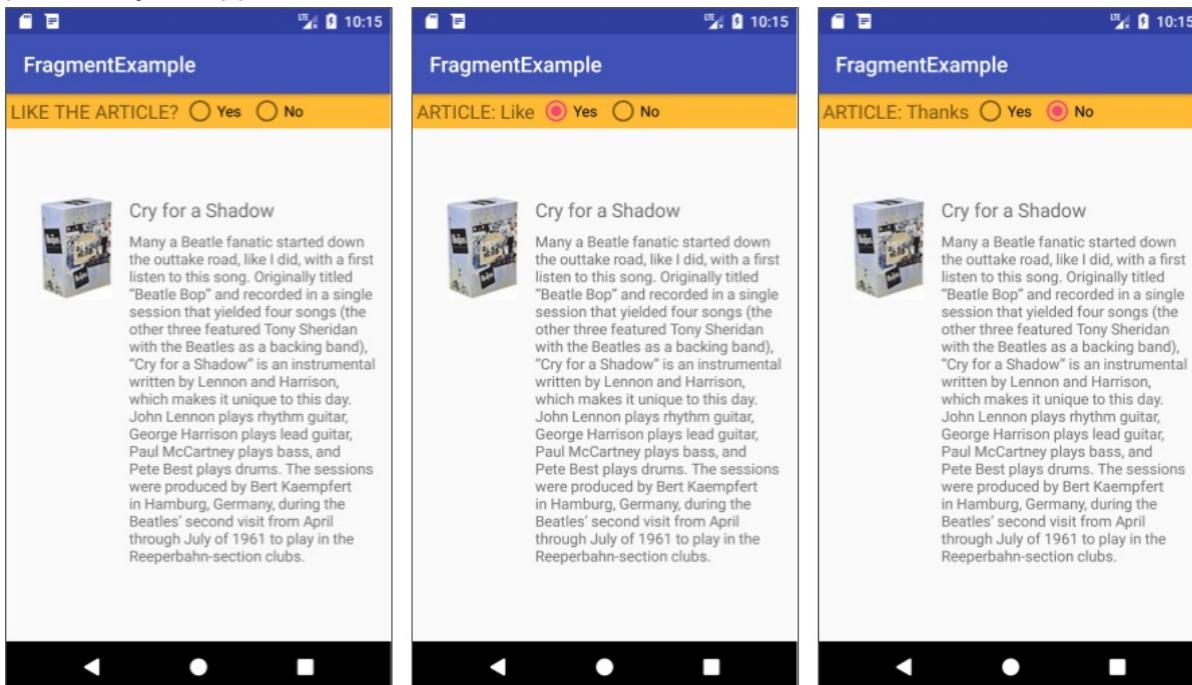
## What you will DO

- Create a `Fragment` to use as a UI element that gives users a "yes" or "no" choice.
- Add interactive elements to the `Fragment` that enable the user to choose "yes" or "no".
- Include the `Fragment` for the duration of an `Activity`.
- Use `Fragment` transactions to add, replace, and remove a `Fragment` while an `Activity` is running.

## App overview

The `FragmentExample1` app shows an image and the title and text of a magazine article. It also shows a `Fragment` that enables users to provide feedback for the article. In this case the feedback is very simple: just "Yes" or "No" to the question "Like the article?" Depending on whether the user gives positive or negative feedback, the app displays an appropriate response.

The `Fragment` is skeletal, but it demonstrates how to create a `Fragment` to use in multiple places in your app's UI.



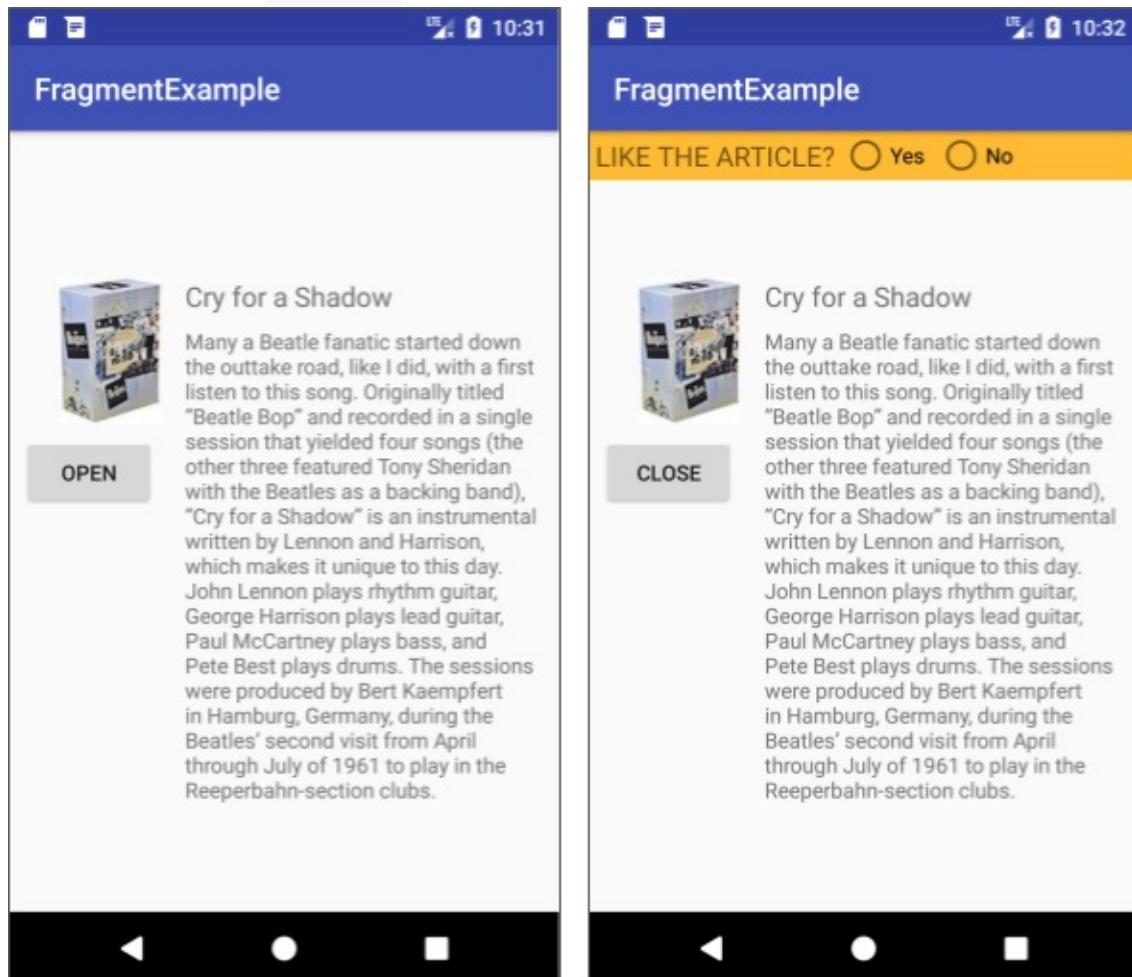
In the first task, you add the `Fragment` *statically* to the `Activity` layout so that it is displayed for the entire duration of the `Activity` lifecycle. The user can interact with the radio buttons in the `Fragment` to choose either "Yes" or "No," as shown in the figure above.

- If the user chooses "Yes," the text in the `Fragment` changes to "Article: Like."

- If the user chooses "No," the text in the `Fragment` changes to "Article: Thanks."

In the second task, in which you create the `FragmentExample2` app, you add the `Fragment` *dynamically* — your code adds, replaces, and removes the `Fragment` while the `Activity` is running. You will change the `Activity` code and layout to do this.

As shown below, the user can tap the **Open** button to show the `Fragment` at the top of the screen. The user can then interact with the UI elements in the `Fragment`. The user taps **Close** to close the `Fragment`.

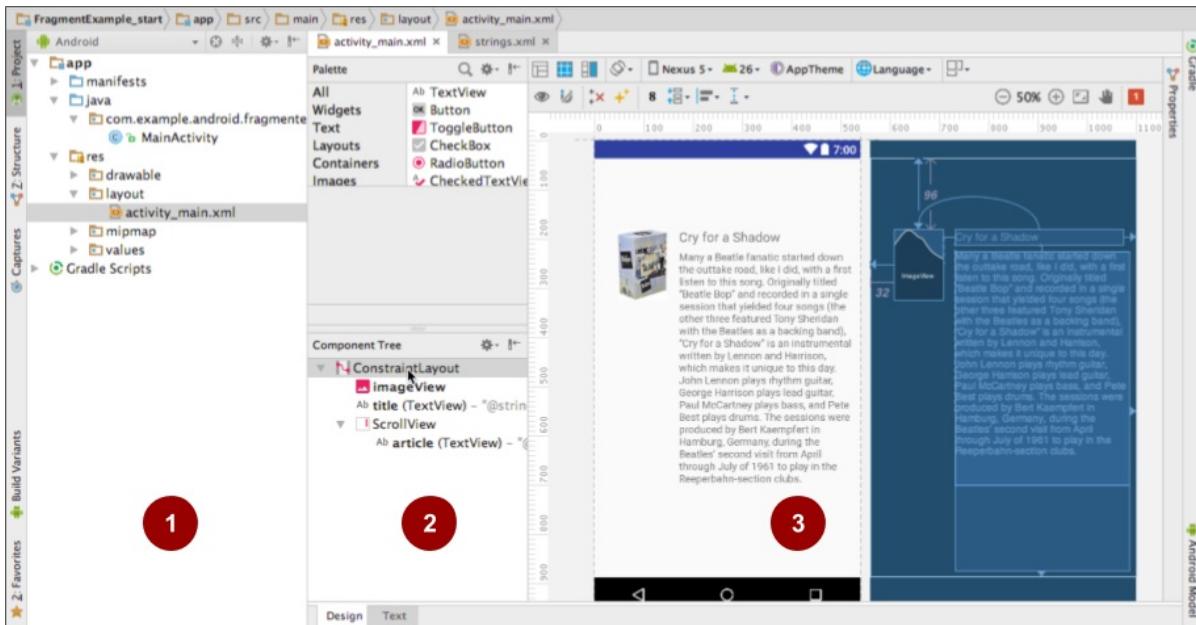


## Task 1. Include a Fragment for the entire Activity lifecycle

In this task you modify a starter app to add a `Fragment` *statically* as a part of the layout of the `Activity`, which means that the `Fragment` is shown during the entire lifecycle of the `Activity`. This is a useful technique for consolidating a set of UI elements (such as radio buttons and text) and user interaction behavior that you can reuse in layouts for other activities.

## 1.1 Open the starter app project

Download the [FragmentExample\\_start](#) Android Studio project. Refactor and rename the project to `FragmentExample1`. (For help with copying projects and refactoring and renaming, see [Copy and rename a project](#).) Explore the app using Android Studio.



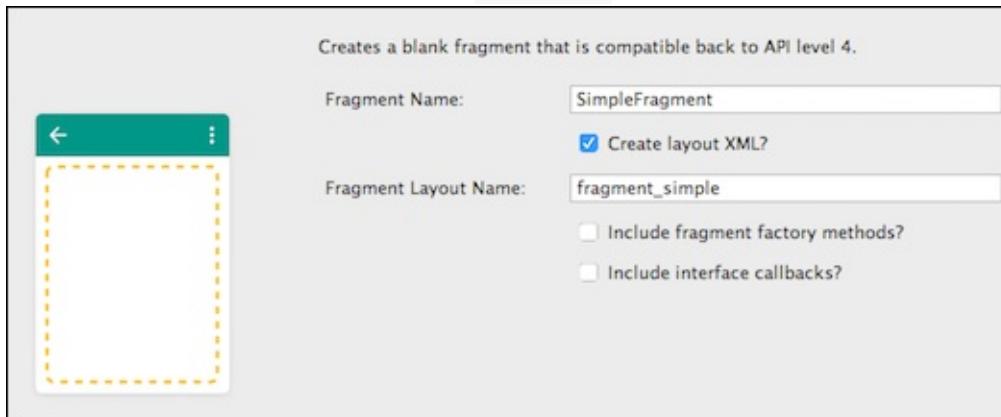
In the above figure:

1. The **Project: Android** pane shows the contents of the project.
2. The **Component Tree** pane for the `activity_main.xml` layout shows the `android:id` value of each UI element.
3. The layout includes image and text elements arranged to provide a space at the top for the app bar and a `Fragment`.

## 1.2 Add a Fragment

1. In **Project: Android** view, expand `app > java` and select `com.example.android.fragmentexample`.
2. Choose **File > New > Fragment > Fragment (Blank)**.
3. In the **Configure Component** dialog, name the `Fragment SimpleFragment`. The **Create layout XML** option should already be selected, and the `Fragment` layout will be filled in as `fragment_simple`.

4. Uncheck the **Include fragment factory methods** and **Include interface callbacks** options. Click **Finish** to create the `Fragment`.



5. Open `SimpleFragment`, and inspect the code:

```
public class SimpleFragment extends Fragment {

    public SimpleFragment() {
        // Required empty public constructor
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
                           ViewGroup container, Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_simple,
                               container, false);
    }
}
```

All subclasses of `Fragment` must include a public no-argument constructor as shown. The Android framework often re-instantiates a `Fragment` object when needed, in particular during state restore. The framework needs to be able to find this constructor so it can instantiate the `Fragment`.

The `Fragment` class uses callback methods that are similar to `Activity` callback methods. For example, `onCreateView()` provides a `LayoutInflater` to inflate the `Fragment` UI from the layout resource `fragment_simple`.

## 1.3 Edit the Fragment's layout

1. Open `fragment_simple.xml`. In the layout editor pane, click **Text** to view the XML.
2. Change the attributes for the "Hello blank fragment" `TextView`:

TextView attribute	Value
android:id	"@+id/fragment_header"
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:textAppearance	"@style/Base.TextAppearance.AppCompat.Medium"
android:padding	"4dp"
android:text	"@string/question_article"

The `@string/question_article` resource is defined in the `strings.xml` file in the starter app as `"LIKE THE ARTICLE?"`.

3. Change the `FrameLayout` root element to `LinearLayout`, and change the following attributes for the `LinearLayout`:

LinearLayout attribute	Value
android:layout_height	"wrap_content"
android:background	"@color/my_fragment_color"
android:orientation	"horizontal"

The `my_fragment_color` value for the `android:background` attribute is already defined in the starter app in `colors.xml`.

4. Within the `LinearLayout`, add the following `RadioGroup` element after the `TextView` element:

```

<RadioGroup
    android:id="@+id/radio_group"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <RadioButton
        android:id="@+id/radio_button_yes"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="8dp"
        android:text="@string/yes" />

    <RadioButton
        android:id="@+id/radio_button_no"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="8dp"
        android:text="@string/no" />

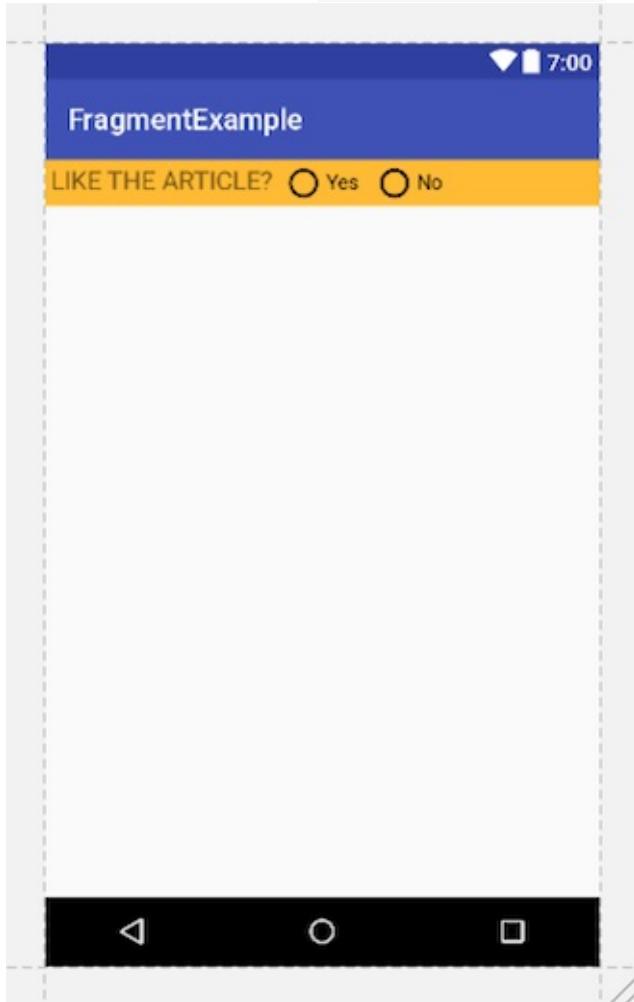
</RadioGroup>

```

The `@string/yes` and `@string/no` resources are defined in the `strings.xml` file in the starter app as `"Yes"` and `"No"`. In addition, the following string resources are also defined in the `strings.xml` file:

```
<string name="yes_message">ARTICLE: Like</string>
<string name="no_message">ARTICLE: Thanks</string>
```

The layout preview for `fragment_simple.xml` should look like the following:



## 1.4 Add a listener for the radio buttons

Add the `RadioGroup.OnCheckedChangeListener` interface to define a callback to be invoked when a radio button is checked or unchecked:

1. Open `simpleFragment` again.
2. Change the code in `onCreateView()` to the code below. This code sets up the `View` and the `RadioGroup`, and returns the `View` (`rootView`). The `View` and `RadioGroup` are declared as `final`, which means they won't change. This is because you will need to access them from within an anonymous inner class (the `OnCheckedChangeListener` for the `RadioGroup`):

```

// Inflate the layout for this fragment.
final View rootView =
    inflater.inflate(R.layout.fragment_simple, container, false);
final RadioGroup radioGroup = rootView.findViewById(R.id.radio_group);

// TODO: Set the radioGroup onCheckedChanged listener.

// Return the View for the fragment's UI.
return rootView;

```

3. Add the following constants to the top of `SimpleFragment` to represent the two states of the radio button choice: 0 = yes and 1 = no:

```

private static final int YES = 0;
private static final int NO = 1;

```

4. Replace the `TODO` comment inside the `onCreateView()` method with the following code to set the radio group listener and change the `textView` (the `fragment_header` in the layout) depending on the radio button choice:

```

radioGroup.setOnCheckedChangeListener(new
    RadioGroup.OnCheckedChangeListener() {
        @Override
        public void onCheckedChanged(RadioGroup group, int checkedId) {
            View radioButton = radioGroup.findViewById(checkedId);
            int index = radioGroup.indexOfChild(radioButton);
            TextView textView =
                rootView.findViewById(R.id.fragment_header);
            switch (index) {
                case YES: // User chose "Yes."
                    textView.setText(R.string.yes_message);
                    break;
                case NO: // User chose "No."
                    textView.setText(R.string.no_message);
                    break;
                default: // No choice made.
                    // Do nothing.
                    break;
            }
        }
    });

```

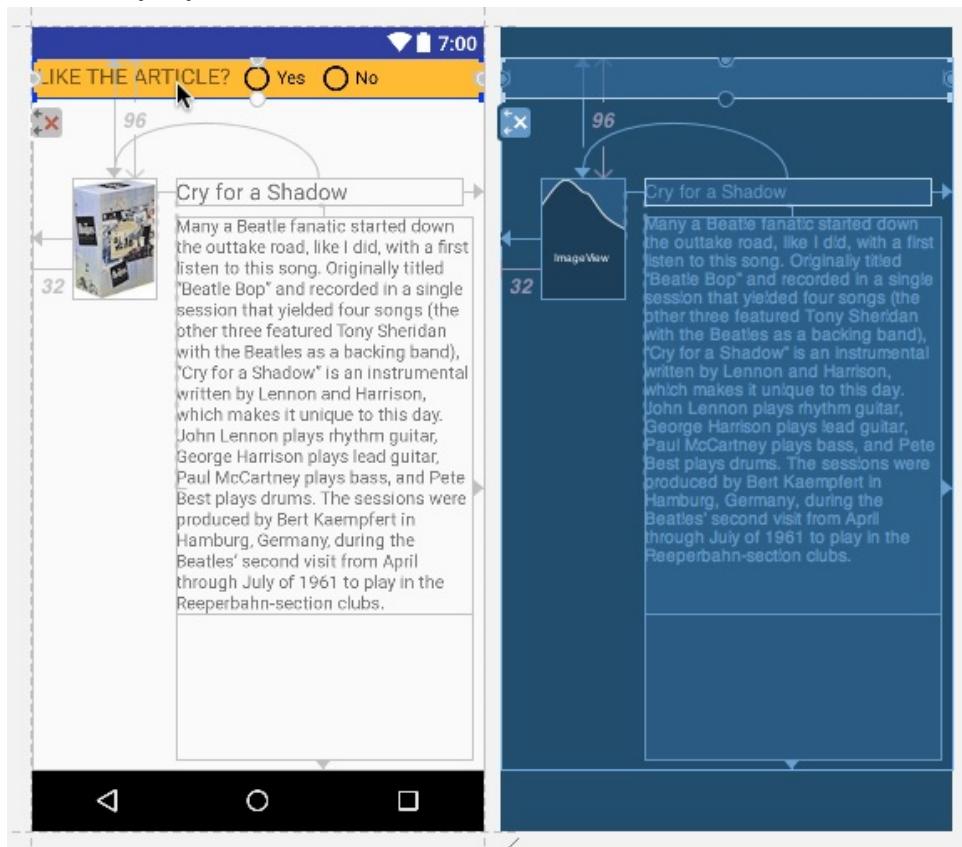
The `oncheckedchanged()` method must be implemented for `RadioGroup.OnCheckedChangeListener`; in this example, the index for the two radio buttons is 0 ("Yes") or 1 ("No"), which display either the `yes_message` text or the `no_message` text.

## 1.5 Add the Fragment statically to the Activity

1. Open `activity_main.xml`.
2. Add the following `<fragment>` element below the `ScrollView` within the `ConstraintLayout` root view. It refers to the `SimpleFragment` you just created:

```
<fragment
    android:id="@+id/fragment"
    android:name="com.example.android.fragmentexample.SimpleFragment"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:layout="@layout/fragment_simple" />
```

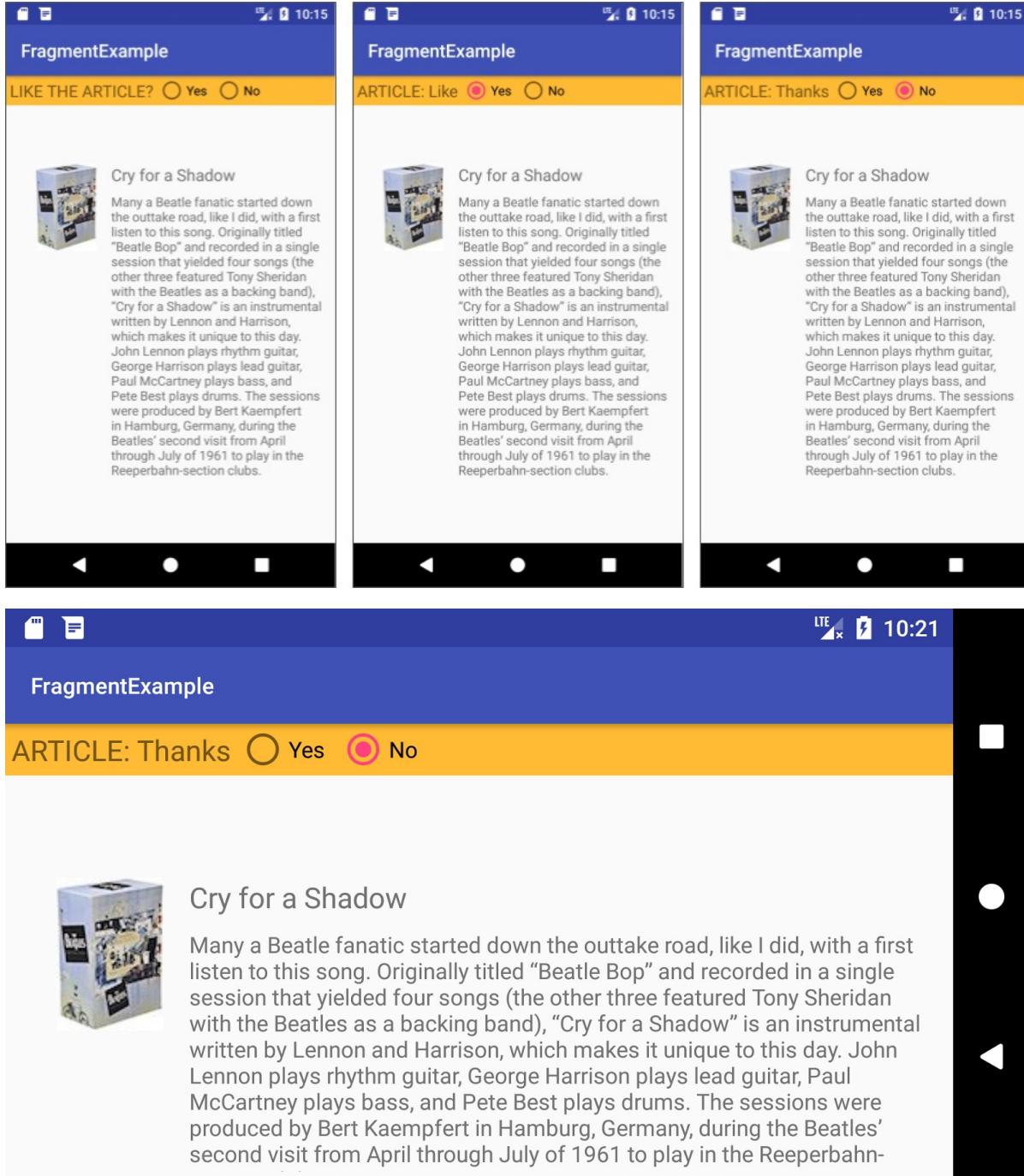
The figure below shows the layout preview, with `SimpleFragment` statically included in the activity layout:



A render error may appear below the preview, because a `<fragment>` element can dynamically include different layouts when the app runs, and the layout editor doesn't know what layout to use for a preview. To see the `Fragment` in the preview, click the `@layout/fragment_simple` link in the error message, and choose `SimpleFragment`.

3. Run the app. The `Fragment` is now included in the `MainActivity` layout, as shown in the figures below.

4. Tap a radio button. The "LIKE THE ARTICLE?" text is replaced by either the `yes_message` text ("ARTICLE: Like") or the `no_message` text ("ARTICLE: Thanks"), depending on which radio button is tapped.
5. After tapping a radio button, change the orientation of your device or emulator from portrait to landscape. Note that the "Yes" or "No" choice is still selected.



Switching the device orientation after choosing "No" demonstrates that a `Fragment` can retain an instance of its data after a configuration change (such as changing the orientation). This feature makes a `Fragment` useful as a UI component, as compared to using separate `Views`. While an `Activity` is destroyed and recreated when a device's configuration changes, a `Fragment` is not destroyed.

## Task 1 solution code

Android Studio project: [FragmentExample1](#)

## Coding challenge

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

**Challenge:** Expand the `Fragment` to include another question ("Like the song?") underneath the first question, so that it appears as shown in the figure below. Add a `RatingBar` so that the user can set a rating for the song, and a `Toast` message that shows the chosen rating.

**Hint:** Use the `OnRatingBarChangeListener` in the `Fragment`, and be sure to include the `android:isIndicator` attribute, set to `false`, for the `RatingBar` in the `Fragment` layout.

This challenge demonstrates how a `Fragment` can handle different kinds of user input.

The screenshot shows a mobile application interface. At the top, there's a blue header bar with the title "FragmentExample". Below it is a yellow navigation bar containing two interactive elements: "ARTICLE: Like" followed by two radio buttons, one pink filled and one grey unfilled, both labeled "Yes" and "No". Next to this is the text "LIKE THE SONG?" followed by a row of five stars, where the first four are pink and the last one is yellow.

The main content area features a small thumbnail image of a Beatles album cover for "Cry for a Shadow" on the left. To its right, the song title "Cry for a Shadow" is displayed in large, bold, dark grey text. Below the title is a detailed paragraph about the song's history and instrumentation. A semi-transparent dark grey oval is overlaid on the bottom right of the text block, containing the text "My Rating: 4.5".

At the very bottom of the screen is a black navigation bar with three white icons: a triangle pointing left, a circle, and a square.

ARTICLE: Like  Yes  No

LIKE THE SONG? ★★★★☆



## Cry for a Shadow

Many a Beatle fanatic started down the outtake road, like I did, with a first listen to this song. Originally titled "Beatle Bop" and recorded in a single session that yielded four songs (the other three featured Tony Sheridan with the Beatles as a backing band), "Cry for a Shadow" is an instrumental written by Lennon and Harrison, which makes it unique to this day. John Lennon plays rhythm guitar, George Harrison plays lead guitar, Paul McCartney plays bass, and Pete Best plays drums. The sessions were produced by Bert Kaempfert in Hamburg, Germany, during the Beatles' second visit from April through July of 1961 to play in the Kneipe in section clubs.

My Rating: 4.5

## Challenge solution code

Android Studio project: [FragmentExample1\\_challenge](#)

## Task 2. Add a Fragment to an Activity dynamically

In this task you learn how to add the same `Fragment` to an `Activity` *dynamically*. The `Fragment` is added only if the user performs an interaction in the `Activity` —in this case, tapping a button.

You will change the [FragmentExample1](#) app to manage the `Fragment` using `FragmentManager` and `FragmentTransaction` statements that can add, remove, and replace a `Fragment`.

### 2.1 Add a ViewGroup for the Fragment

At any time while your `Activity` is running, your code can add a `Fragment` to the `Activity`. All your `Activity` code needs is a `ViewGroup` in the layout as a placeholder for the `Fragment`, such as a `FrameLayout`. Change the `<fragment>` element to `<FrameLayout>` in the main layout. Follow these steps:

1. Copy the [FragmentExample1](#) project, and open the copy in Android Studio. Refactor and rename the project to `FragmentExample2`. (For help with copying projects and refactoring and renaming, see [Copy and rename a project](#).)

If a warning appears to remove the `FragmentExample1` modules, click to remove them.

2. Open the `activity_main.xml` layout, and switch the layout editor to **Text (XML)** view.
3. Change the `<fragment>` element to `<FrameLayout>`, and change the `android:id` to `fragment_container`, as shown below:

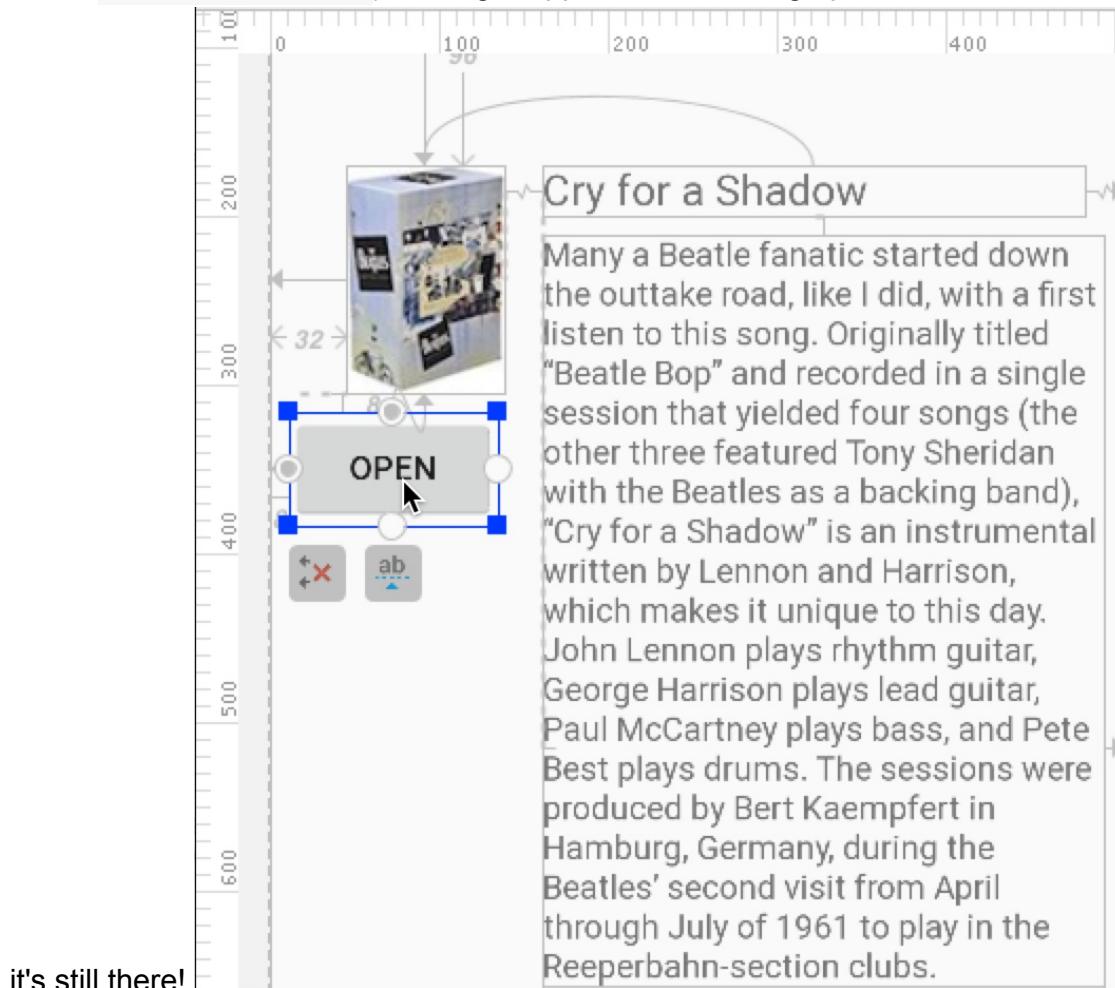
```
<FrameLayout  
    android:id="@+id/fragment_container"
```

### 2.2 Add a button to open and close the Fragment

Users need a way to open and close the `Fragment` from the `Activity`. To keep this example simple, add a `Button` to open the `Fragment`, and if the `Fragment` is open, the same `Button` can close the `Fragment`.

1. Open `activity_main.xml`, click the **Design** tab if it is not already selected, and add a `Button` under the `imageView` element.
2. Constrain the `Button` to the bottom of `imageView` and to the left side of the parent.
3. Open the **Attributes** pane, and add the ID `open_button` and the text "`@string/open`". The `@string/open` and `@string/close` resources are defined in the `strings.xml` file in the starter app as "`OPEN`" and "`CLOSE`".

Note that since you have changed `<fragment>` to `<FrameLayout>`, the `Fragment` (now called `fragment_container`) no longer appears in the design preview—but don't worry,



4. Open `MainActivity`, and declare and initialize the `Button`. Also add a `private boolean` to determine whether the `Fragment` is displayed:

```
private Button mButton;
private boolean isFragmentDisplayed = false;
```

5. Add the following to the `onCreate()` method to initialize the `mButton`:

```
mButton = findViewById(R.id.open_button);
```

## 2.3 Use Fragment transactions

To manage a `Fragment` in your `Activity`, use `FragmentManager`. To perform a `Fragment` transaction in your `Activity` —such as adding, removing, or replacing a `Fragment`—use methods in `FragmentTransaction`.

The best practice for instantiating the `Fragment` in the `Activity` is to provide a `newInstance()` factory method in the `Fragment`. Follow these steps to add the `newInstance()` method to `SimpleFragment` and instantiate the `Fragment` in `MainActivity`. You will also add the `displayFragment()` and `closeFragment()` methods, and use `Fragment` transactions:

1. Open `simpleFragment`, and add the following method for instantiating and returning the `Fragment` to the `Activity`:

```
public static SimpleFragment newInstance() {  
    return new SimpleFragment();  
}
```

2. Open `MainActivity`, and create the following `displayFragment()` method to instantiate and open `SimpleFragment`. It starts by creating an instance of `simpleFragment` by calling the `newInstance()` method in `SimpleFragment`:

```
public void displayFragment() {  
    SimpleFragment simpleFragment = SimpleFragment.newInstance();  
    // TODO: Get the FragmentManager and start a transaction.  
    // TODO: Add the SimpleFragment.  
}
```

3. Replace the `TODO: Get the FragmentManager...` comment in the above code with the following:

```
// Get the FragmentManager and start a transaction.  
FragmentManager fragmentManager = getSupportFragmentManager();  
FragmentTransaction fragmentTransaction = fragmentManager  
    .beginTransaction();
```

To start a transaction, get an instance of `FragmentManager` using `getSupportFragmentManager()`, and then get an instance of `FragmentTransaction` that uses `beginTransaction()`. `Fragment` operations are wrapped into a *transaction* (similar to a bank transaction) so that all the operations finish before the transaction is committed for the final result.

Use `getSupportFragmentManager()` (instead of `getFragmentManager()`) to instantiate the `Fragment` class so your app remains compatible with devices running system versions as low as Android 1.6. (The `getSupportFragmentManager()` method uses the [Support Library](#).)

4. Replace the `TODO: Add the SimpleFragment` comment in the above code with the following:

```
// Add the SimpleFragment.  
fragmentTransaction.add(R.id.fragment_container,  
    simpleFragment).addToBackStack(null).commit();  
// Update the Button text.  
mButton.setText(R.string.close);  
// Set boolean flag to indicate fragment is open.  
isFragmentDisplayed = true;
```

This code adds a new `Fragment` using the `add()` transaction method. The first argument passed to `add()` is the layout resource (`fragment_container`) for the `ViewGroup` in which the `Fragment` should be placed. The second parameter is the `Fragment` (`simpleFragment`) to add.

In addition to the `add()` transaction, the code calls `addToBackStack(null)` in order to add the transaction to a back stack of `Fragment` transactions. This back stack is managed by the `Activity`. It allows the user to return to the previous `Fragment` state by pressing the Back button.

The code then calls `commit()` for the transaction to take effect.

The code also changes the text of the `Button` to "CLOSE" and sets the Boolean `isFragmentDisplayed` to `true` so that you can track the state of the `Fragment`.

5. To close the `Fragment`, add the following `closeFragment()` method to `MainActivity`:

```

public void closeFragment() {
    // Get the FragmentManager.
    FragmentManager fragmentManager = getSupportFragmentManager();
    // Check to see if the fragment is already showing.
    SimpleFragment simpleFragment = (SimpleFragment) fragmentManager
        .findFragmentById(R.id.fragment_container);
    if (simpleFragment != null) {
        // Create and commit the transaction to remove the fragment.
        FragmentTransaction fragmentTransaction =
            fragmentManager.beginTransaction();
        fragmentTransaction.remove(simpleFragment).commit();
    }
    // Update the Button text.
    mButton.setText(R.string.open);
    // Set boolean flag to indicate fragment is closed.
    isFragmentDisplayed = false;
}

```

As with `displayFragment()`, the `closeFragment()` code snippet gets an instance of `FragmentManager` using `getSupportFragmentManager()`, uses `beginTransaction()` to start a series of transactions, and acquires a reference to the `Fragment` using the layout resource (`fragment_container`). It then uses the `remove()` transaction to remove the `Fragment`.

However, before creating this transaction, the code checks to see if the `Fragment` is displayed (not `null`). If the `Fragment` is not displayed, there's nothing to remove.

The code also changes the text of the `Button` to "OPEN" and sets the Boolean `isFragmentDisplayed` to `false` so that you can track the state of the `Fragment`.

## 2.4 Set the Button onClickListener

To take action when the user clicks the `Button`, implement an `onClickListener` for the button in the `onCreate()` method of `MainActivity` in order to open and close the fragment based on the boolean value of `isFragmentDisplayed`. The `onClick()` method will call the `displayFragment()` method to open the `Fragment` if the `Fragment` is not already open; otherwise it calls `closeFragment()`.

You will also add code to save the value of `isFragmentDisplayed` and use it if the configuration changes, such as if the user switches from portrait or landscape orientation.

1. Open `MainActivity`, and add the following to the `onCreate()` method to set the click listener for the Button:

```
// Set the click listener for the button.
mButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (!isFragmentDisplayed) {
            displayFragment();
        } else {
            closeFragment();
        }
    }
});
```

2. To save the boolean value representing the `Fragment` display state, define a key for the `Fragment` state to use in the `savedInstanceState` `Bundle`. Add this member variable to `MainActivity`:

```
static final String STATE_FRAGMENT = "state_of_fragment";
```

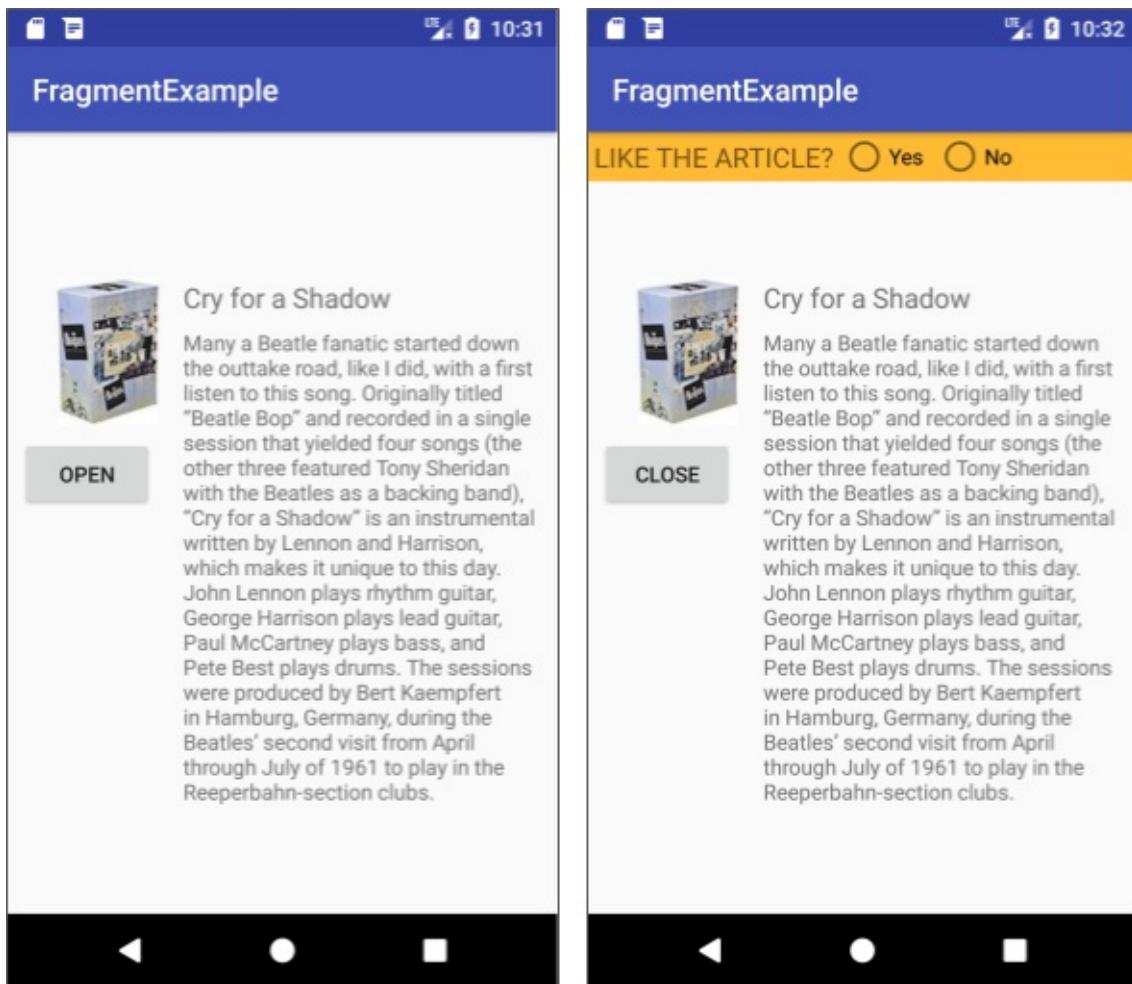
3. Add the following method to `MainActivity` to save the state of the `Fragment` if the configuration changes:

```
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the state of the fragment (true=open, false=closed).
    savedInstanceState.putBoolean(STATE_FRAGMENT, isFragmentDisplayed);
    super.onSaveInstanceState(savedInstanceState);
}
```

4. Go back to the `onCreate()` method and add the following code. It checks to see if the instance state of the `Activity` was saved for some reason, such as a configuration change (the user switching from vertical to horizontal). If the saved instance was not saved, it would be `null`. If the saved instance is *not* `null`, the code retrieves the `Fragment` state from the saved instance, and sets the `Button` text:

```
if (savedInstanceState != null) {
    isFragmentDisplayed =
        savedInstanceState.getBoolean(STATE_FRAGMENT);
    if (isFragmentDisplayed) {
        // If the fragment is displayed, change button to "close".
        mButton.setText(R.string.close);
    }
}
```

5. Run the app. Tapping **Open** adds the `Fragment` and shows the **Close** text in the button. Tapping **Close** removes the `Fragment` and shows the **Open** text in the button. You can switch your device or emulator from vertical to horizontal orientation to see that the buttons and `Fragment` work.



## Task 2 solution code

Android Studio project: [FragmentExample2](#)

## Summary

- To create a blank `Fragment`, expand **app > java** in **Project: Android** view, select the folder containing the Java code for your app, and choose **File > New > Fragment > Fragment (Blank)**.
- The `Fragment` class uses callback methods that are similar to `Activity` callbacks, such as `onCreateView()`, which provides a `LayoutInflater` object to inflate the `Fragment` UI from the layout resource `fragment_simple`.
- To add a `Fragment` *statically* to an `Activity` so that it is displayed for the entire lifecycle of the `Activity`, declare the `Fragment` inside the layout file for the `Activity` using the `<fragment>` tag. You can specify layout attributes for the `Fragment` as if it were a `View`.
- To add a `Fragment` *dynamically* to an `Activity` so that the `Activity` can add, replace,

or remove it, specify a `ViewGroup` inside the layout file for the `Activity` such as a `FrameLayout`.

- When adding a `Fragment` dynamically to an `Activity`, the best practice for creating the fragment is to create the instance with a `newInstance()` method in the `Fragment` itself. Call the `newInstance()` method from the `Activity` to create a new instance.
- To get an instance of `FragmentManager`, use `getSupportFragmentManager()` in order to instantiate the `Fragment` class using the **Support Library** so your app remains compatible with devices running system versions as low as Android 1.6.
- To start a series of `Fragment` transactions, call `beginTransaction()` on a `FragmentTransaction`.
- With `FragmentManager` your code can perform the following `Fragment` transactions while the app runs, using `FragmentTransaction` methods:
  - Add a `Fragment` using `add()`.
  - Remove a `Fragment` using `remove()`.
  - Replace a `Fragment` with another `Fragment` using `replace()`.

## Related concept

The related concept documentation is [Fragments](#).

## Learn more

Android developer documentation:

- [Fragment](#)
- [Fragments](#)
- [FragmentManager](#)
- [FragmentTransaction](#)
- [Creating a Fragment](#)
- [Building a Flexible UI](#)
- [Building a Dynamic UI with Fragments](#)
- [Handling Configuration Changes](#)
- [Supporting Tablets and Handsets](#)

Videos:

- [What the Fragment? \(Google I/O 2016\)](#)
- [Fragment Tricks \(Google I/O '17\)](#)
- [Por que Precisamos de Fragments?](#)



# 1.2: Communicating with a Fragment

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- Apps overview
- Task 1. Communicating with a fragment
- Task 1 solution code
- Task 2. Changing an app to a master/detail layout
- Task 2 solution code
- Summary
- Related concept
- Learn more

An `Activity` hosting a `Fragment` can send data to and receive data from the `Fragment`. A `Fragment` can't communicate directly with another `Fragment`, even within the same `Activity`. The host `Activity` must be used as an intermediary.

This practical demonstrates how to use an `Activity` to communicate with a `Fragment`. It also shows how to use a `Fragment` to implement a two-pane master/detail layout. A master/detail layout is a layout that allows users to view a list of items (the master view) and drill down into each item for more details (the detail view).

## What you should already KNOW

You should be able to:

- Create a `Fragment` with an interactive UI.
- Add a static `Fragment` to the UI layout of an `Activity`.
- Add, replace, and remove a dynamic `Fragment` while an `Activity` is running.

## What you will LEARN

You will learn how to:

- Send data to a `Fragment`.
- Retrieve data from a `Fragment`.

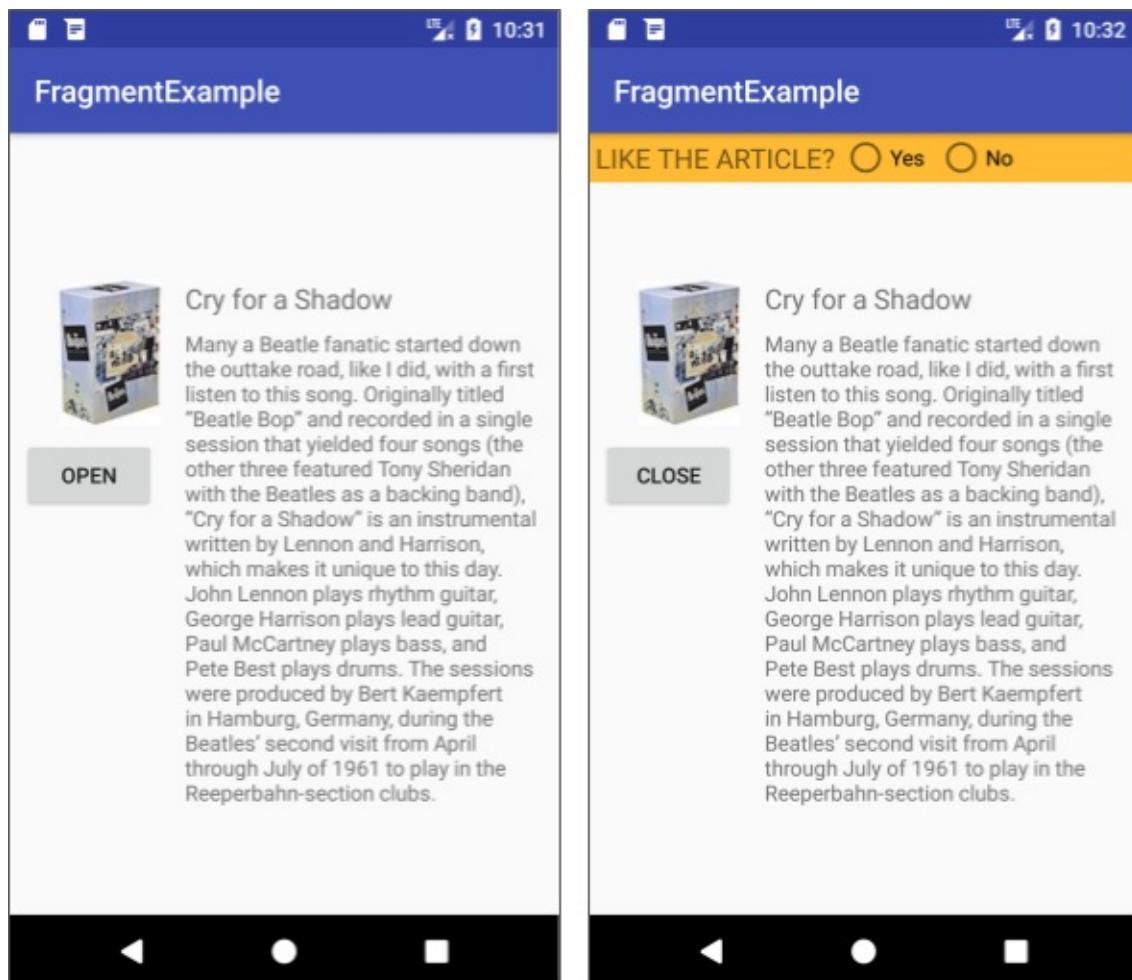
- Implement a master/detail layout for wide screens.

## What you will DO

- Use an argument `Bundle` to send data to a `Fragment`.
- Define a listener interface with a callback method in a `Fragment`.
- Implement the listener for the `Fragment` and a callback to retrieve data from the `Fragment`.
- Move `Activity` code to a `Fragment` for a master/detail layout.

## Apps overview

In [FragmentExample2](#), the app from the lesson on using a `Fragment`, a user can tap the **Open** button to show the `Fragment`, tap a radio button for a "Yes" or "No" choice, and tap **Close** to close the `Fragment`. If the user opens the `Fragment` again, the previous choice is not retained.

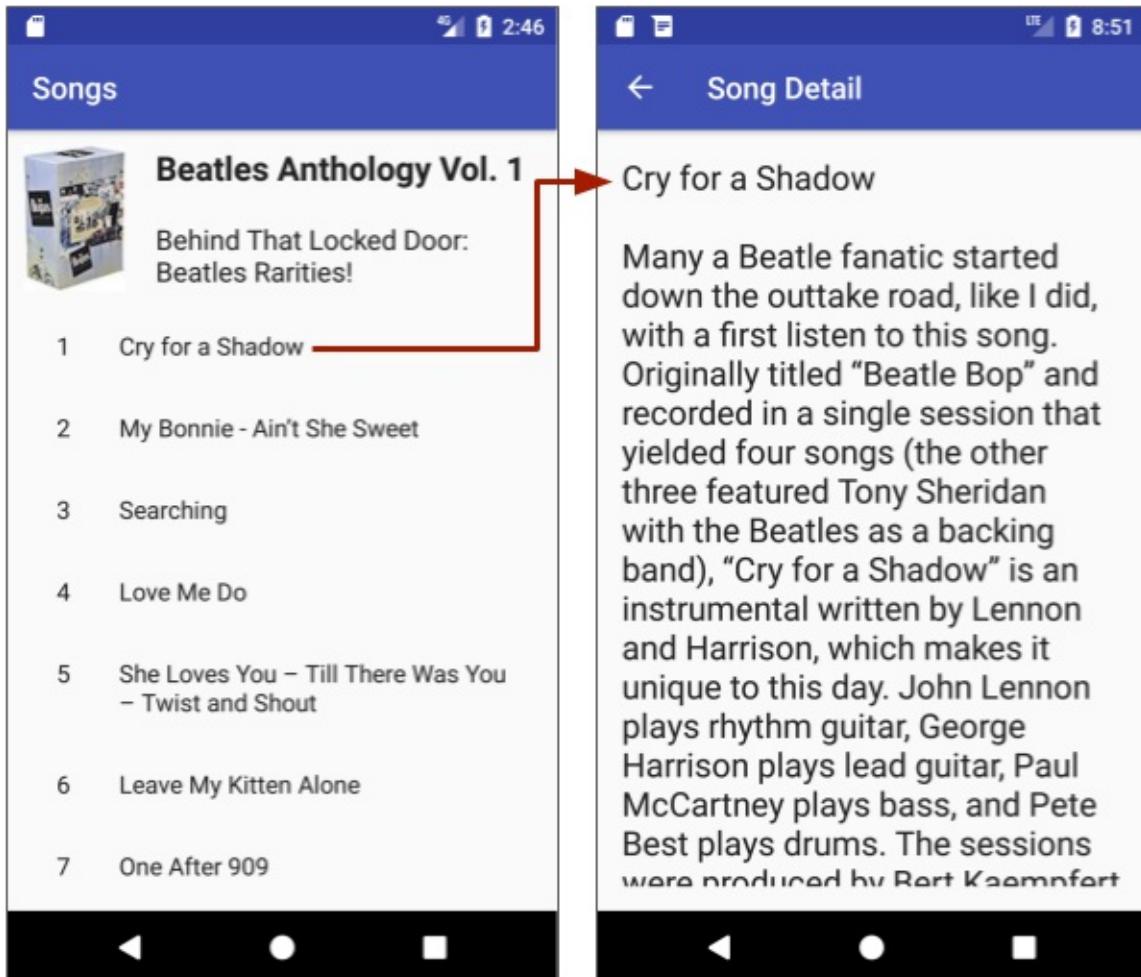


In this lesson you modify the code in the [FragmentExample2](#) app to send the user's choice back to the host `Activity`. When the `Activity` opens a new `Fragment`, the `Activity` can send the user's previous choice to the new `Fragment`. As a result, when the user taps **Open** again, the app shows the previously selected choice.

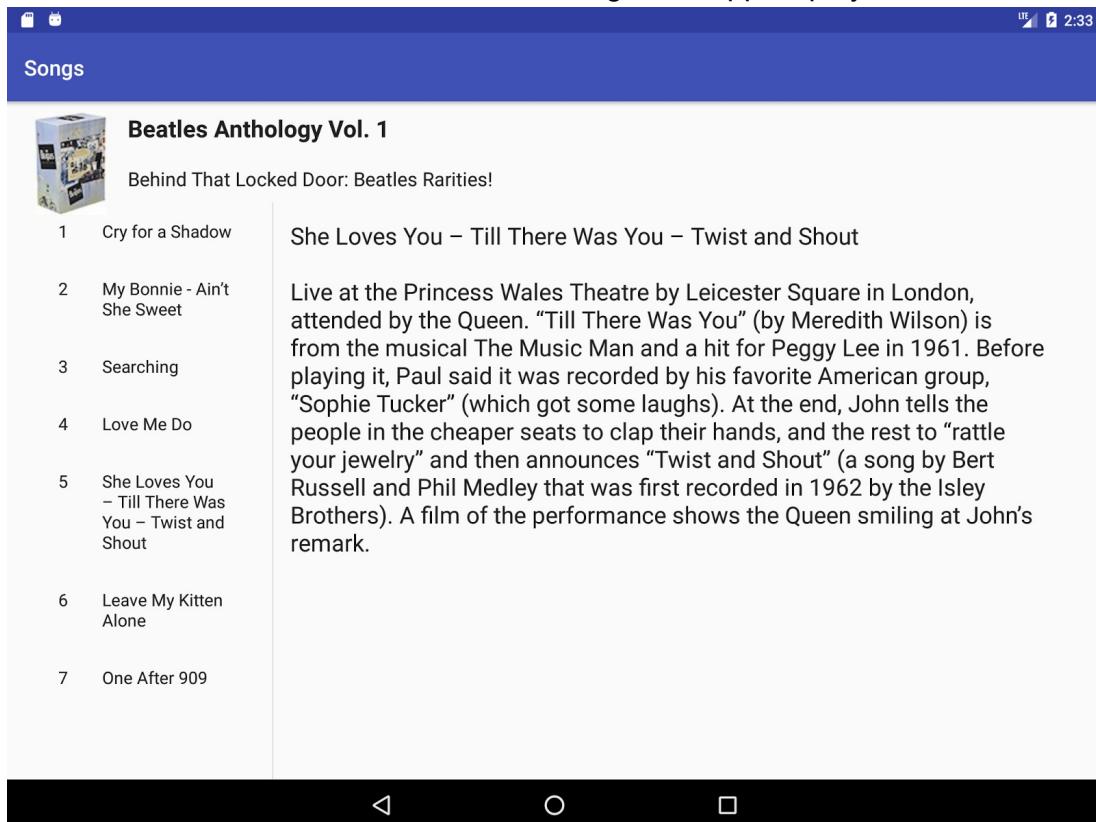
The second app, [SongDetail](#), demonstrates how you can:

- Use a `Fragment` to show a master/detail layout on tablets.
- Provide the `Fragment` with the information it needs to perform tasks.

On a mobile phone screen, the SongDetail app looks like the following figure:



On a tablet in horizontal orientation, the SongDetail app displays a master/detail layout:



## Task 1. Communicating with a fragment

You will modify the [FragmentExample2](#) app from the lesson on creating a `Fragment` with a UI, so that the `Fragment` can tell the host `Activity` which choice ("Yes" or "No") the user made. You will:

- Define a listener interface in the `Fragment` with a callback method to get the user's choice.
- Implement the interface and callback in the host `Activity` to retrieve the user's choice.
- Use the `newInstance()` factory method to provide the user's choice back to the `Fragment` when creating the next `Fragment` instance.

### 1.1 Define a listener interface with a callback

To define a listener interface in the `Fragment`:

1. Copy the [FragmentExample2](#) project in order to preserve it as an intermediate step. Open the new copy in Android Studio, and refactor and rename the new project to `FragmentCommunicate`. (For help with copying projects and refactoring and renaming, see [Copy and rename a project](#).)
2. Open `SimpleFragment`, and add another constant in the `SimpleFragment` class to

represent the third state of the radio button choice, which is 2 if the user has not yet made a choice:

```
private static final int NONE = 2;
```

3. Add a variable for the radio button choice:

```
public int mRadioButtonChoice = NONE;
```

4. Define a listener interface called `OnFragmentInteractionListener`. In it, specify a callback method that you will create, called `onRadioButtonChoice()`:

```
interface OnFragmentInteractionListener {  
    void onRadioButtonChoice(int choice);  
}
```

5. Define a variable for the listener at the top of the `SimpleFragment` class. You will use this variable in `onAttach()` in the next step:

```
OnFragmentInteractionListener mListener;
```

6. Override the `onAttach()` lifecycle method in `SimpleFragment` to capture the host `Activity` interface implementation:

```
@Override  
public void onAttach(Context context) {  
    super.onAttach(context);  
    if (context instanceof OnFragmentInteractionListener) {  
        mListener = (OnFragmentInteractionListener) context;  
    } else {  
        throw new ClassCastException(context.toString()  
            + getResources().getString(R.string.exception_message));  
    }  
}
```

The `onAttach()` method is called as soon as the `Fragment` is associated with the `Activity`. The code makes sure that the host `Activity` has implemented the callback interface. If not, it throws an exception.

The string resource `exception_message` is included in the starter app for the text "must implement `OnFragmentInteractionListener`".

7. In `onCreateView()`, get the `mRadioButtonchoice` by adding code to each `case` of the `switch case` block for the radio buttons:

```

        case YES: // User chose "Yes."
            textView.setText(R.string.yes_message);
            mRadioButtonChoice = YES;
            mListener.onRadioButtonChoice(YES);
            break;
        case NO: // User chose "No."
            textView.setText(R.string.no_message);
            mRadioButtonChoice = NO;
            mListener.onRadioButtonChoice(NO);
            break;
        default: // No choice made.
            mRadioButtonChoice = NONE;
            mListener.onRadioButtonChoice(NONE);
            break;
    }
}

```

## 1.2 Implement the callback in the Activity

To implement the callback:

1. Open `MainActivity` and implement `OnFragmentInteractionListener` in order to receive the data from the `Fragment`:

```

public class MainActivity extends AppCompatActivity
    implements SimpleFragment.OnFragmentInteractionListener {

```

2. In Android Studio, the above code is underlined in red, and a red bulb appears in the left margin. Click the bulb and choose **Implement methods**. Choose `onRadioButtonChoice(choice:int):void`, and click **OK**. An empty `onRadioButtonChoice()` method appears in `MainActivity`.
3. Add a member variable in `MainActivity` for the choice the user makes with the radio buttons, and set it to the default value:

```

private int mRadioButtonChoice = 2; // The default (no choice).

```

4. Add code to the new `onRadioButtonChoice()` method to assign the user's radio button choice from the `Fragment` to `mRadioButtonChoice`. Add a `Toast` to show that the `Activity` has received the data from the `Fragment`:

```

@Override
public void onRadioButtonChoice(int choice) {
    // Keep the radio button choice to pass it back to the fragment.
    mRadioButtonChoice = choice;
    Toast.makeText(this, "Choice is " + Integer.toString(choice),
                  Toast.LENGTH_SHORT).show();
}

```

5. Run the app, tap **Open**, and make a choice. The `Activity` shows the `Toast` message with the choice as an integer (0 is "Yes" and 1 is "No").

LTE 12:15

# FragmentCommunicate

ARTICLE: Like  Yes  No



**CLOSE**

## Cry for a Shadow

Many a Beatle fanatic started down the outtake road, like I did, with a first listen to this song. Originally titled "Beatle Bop" and recorded in a single session that yielded four songs (the other three featured Tony Sheridan with the Beatles as a backing band), "Cry for a Shadow" is an instrumental written by Lennon and Harrison, which makes it unique to this day. John Lennon plays rhythm guitar, George Harrison plays lead guitar, Paul McCartney plays bass, and Pete Best plays drums. The sessions were produced by Bert Kaempfert in Hamburg, Germany, during the Beatles' second visit from April through July of 1961 to play in the Reeperbahn-section clubs.

Choice is 0

◀ ● □

## 1.3 Provide the user's choice to the fragment

To provide the user's previous "Yes" or "No" choice from the host `Activity` to the `Fragment`, pass the choice to the `newInstance()` method in `SimpleFragment` when instantiating `SimpleFragment`. You can set a `Bundle` and use the `Fragment.setArguments(Bundle)` method to supply the construction arguments for the `Fragment`.

Follow these steps:

1. Open `MainActivity`, and change the `newInstance()` statement in `displayFragment()` to the following:

```
SimpleFragment fragment = SimpleFragment.newInstance(mRadioButtonChoice);
```

2. Open `simpleFragment` and add the following constant, which is the key to finding the information in the `Bundle`:

```
private static final String CHOICE = "choice";
```

3. In `SimpleFragment`, change the `newInstance()` method to the following, which uses a `Bundle` and the `setArguments(Bundle)` method to set the arguments before returning the `Fragment`:

```
public static SimpleFragment newInstance(int choice) {  
    SimpleFragment fragment = new SimpleFragment();  
    Bundle arguments = new Bundle();  
    arguments.putInt(CHOICE, choice);  
    fragment.setArguments(arguments);  
    return fragment;  
}
```

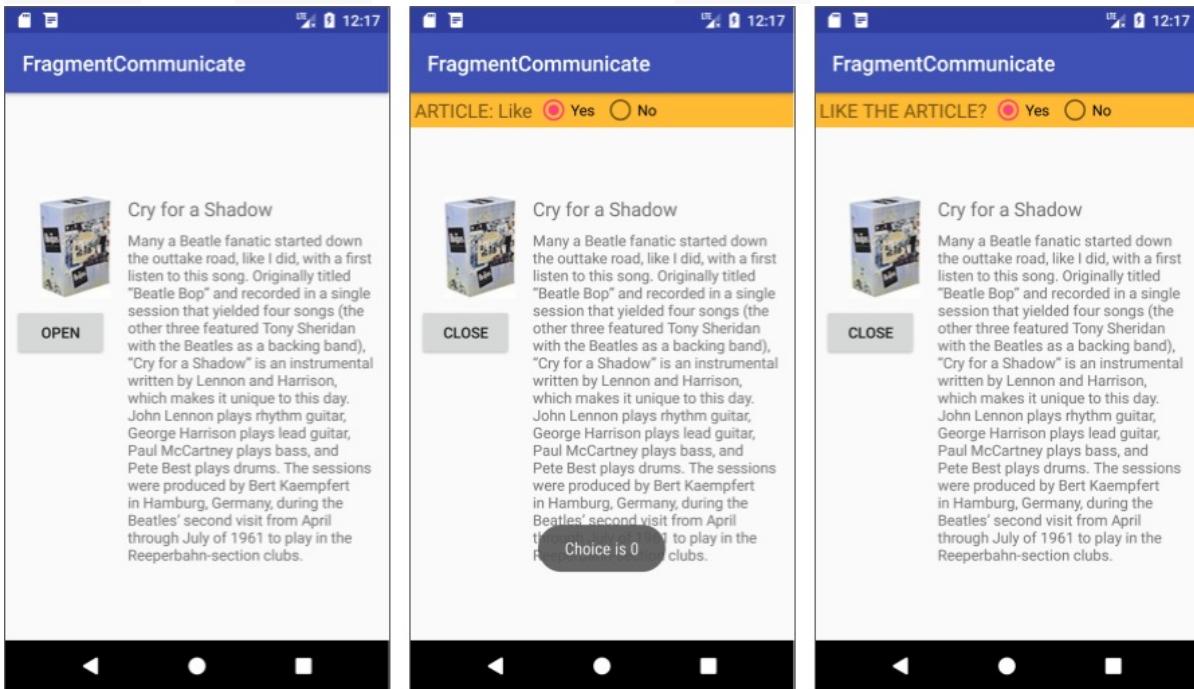
4. Now that a `Bundle` of arguments is available in the `SimpleFragment`, you can add code to the `onCreateView()` method in `SimpleFragment` to get the choice from the `Bundle`. Right before the statement that sets the `radioGroup` `onCheckedChanged` listener, add the following code to retrieve the radio button choice (if a choice was made), and pre-select the radio button.

```

if (getArguments().containsKey(CHOICE)) {
    // A choice was made, so get the choice.
    mRadioButtonChoice = getArguments().getInt(CHOICE);
    // Check the radio button choice.
    if (mRadioButtonChoice != NONE) {
        radioGroup.check
            (radioGroup.getChildAt(mRadioButtonChoice).getId());
    }
}

```

5. Run the app. At first, the app doesn't show the `Fragment` (see the left side of the following figure).
6. Tap **Open** and make a choice such as "Yes" (see the center of the figure below). The choice you made appears in a `Toast`.
7. Tap **Close** to close the `Fragment`.
8. Tap **Open** to reopen the `Fragment`. The `Fragment` appears with the choice already made (see the right side of the figure). Your app has communicated the choice from the `Fragment` to the `Activity`, and then back to the `Fragment`.



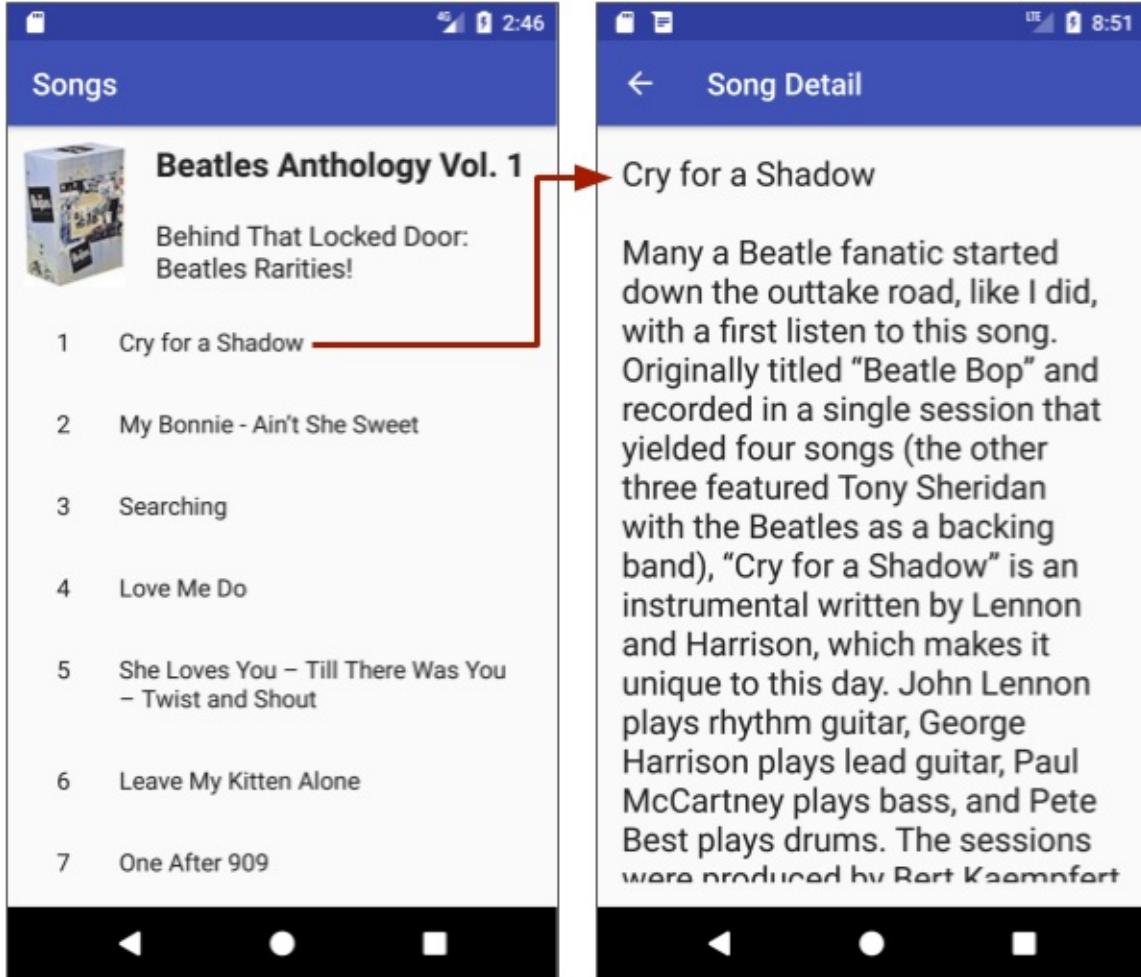
## Task 1 solution code

Android Studio project: [FragmentCommunicate](#)

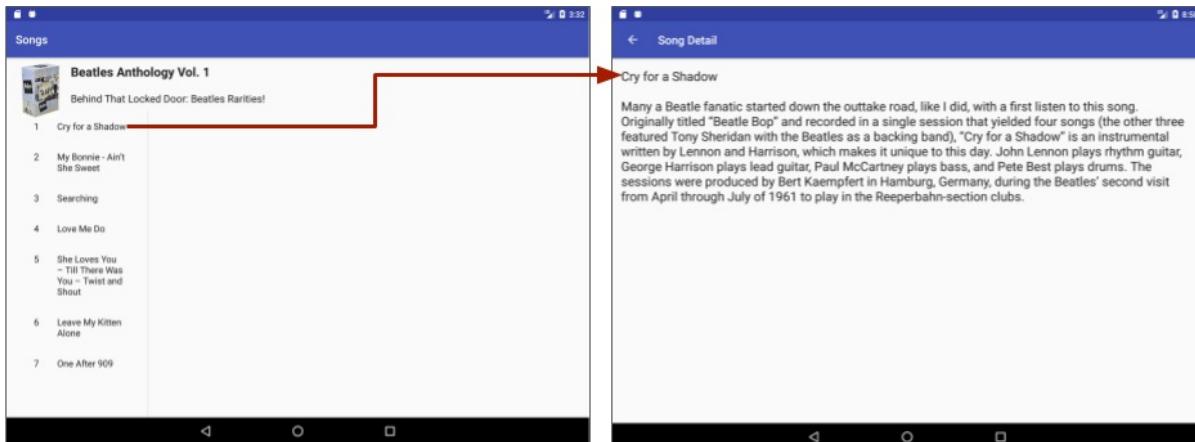
## Task 2. Changing an app to a master/detail layout

This task demonstrates how you can use a `Fragment` to implement a two-pane master/detail layout for a horizontal tablet display. It also shows how to take code from an `Activity` and encapsulate it within a `Fragment`, thereby simplifying the `Activity`.

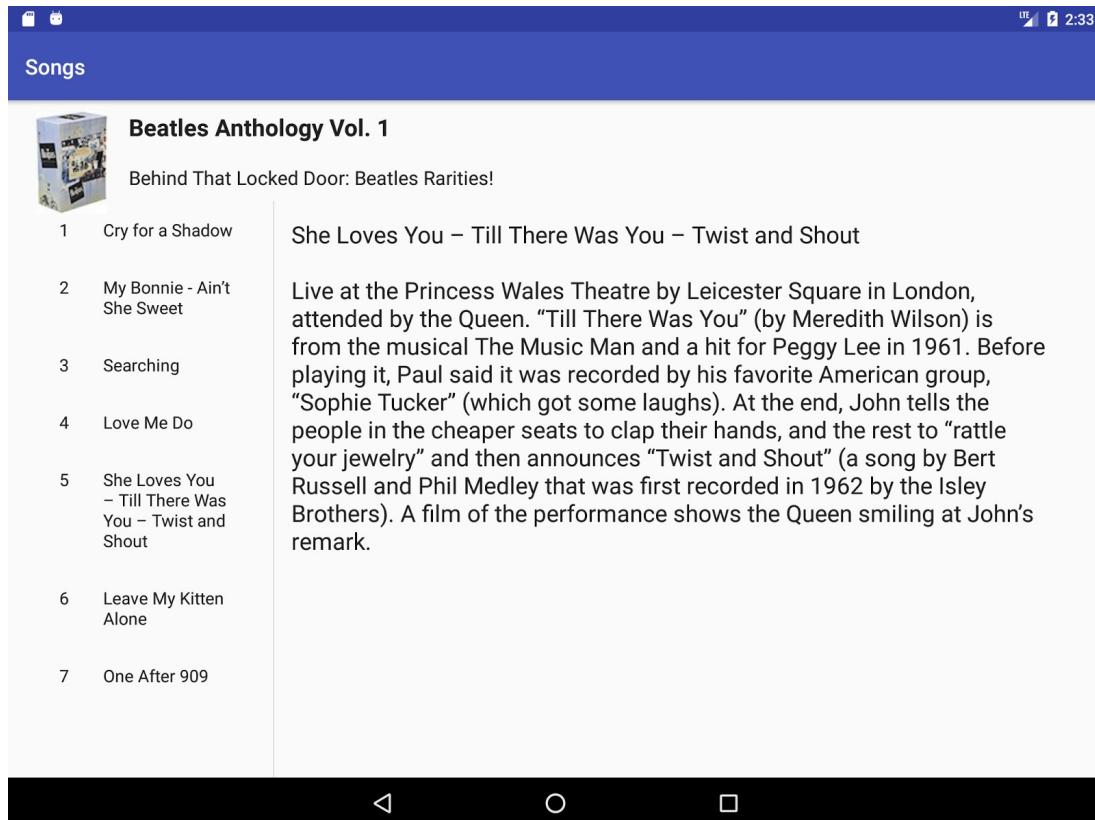
In this task you use a starter app called `SongDetail_start` that displays song titles that the user can tap to see song details.



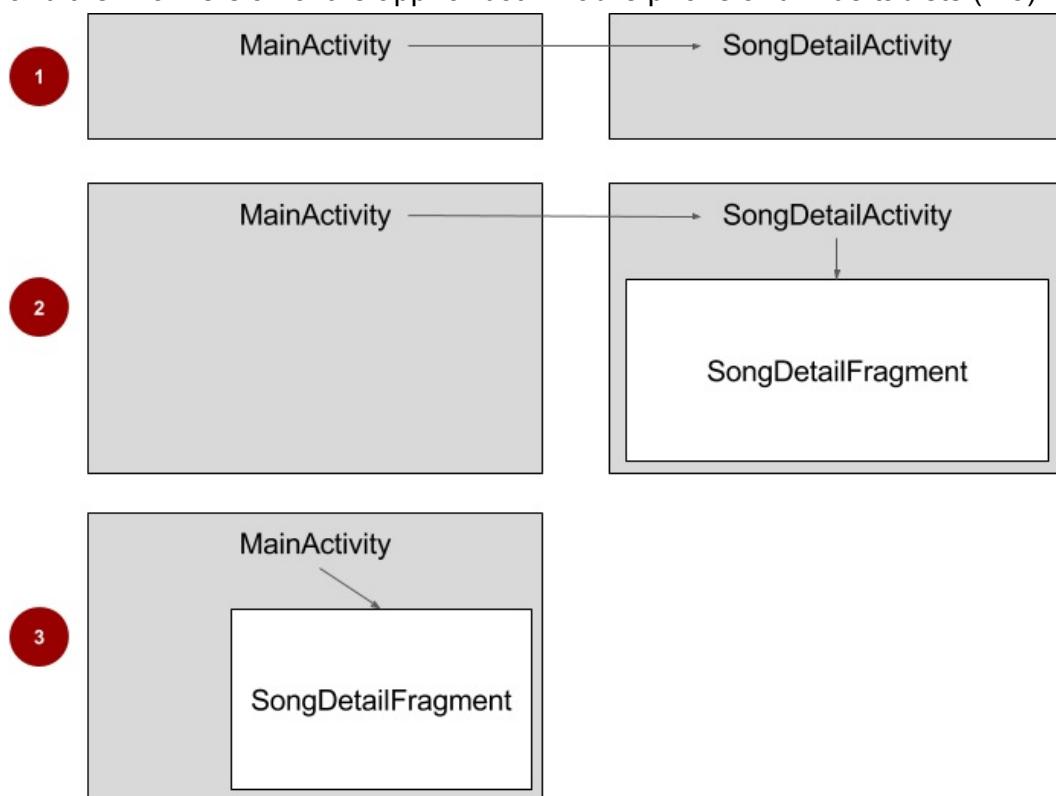
On a tablet, the app doesn't take advantage of the full screen size, as shown in the following figure:



When set to a horizontal orientation, a tablet device is wide enough to show information in a master/detail layout. You will modify the app to show a master/detail layout if the device is wide enough, with the song list as the master, and the `Fragment` as the detail, as shown in the following figure.



The following diagram shows the difference in the code for the `SongDetail` starter app (1), and the final version of the app for both mobile phone and wide tablets (2-3).



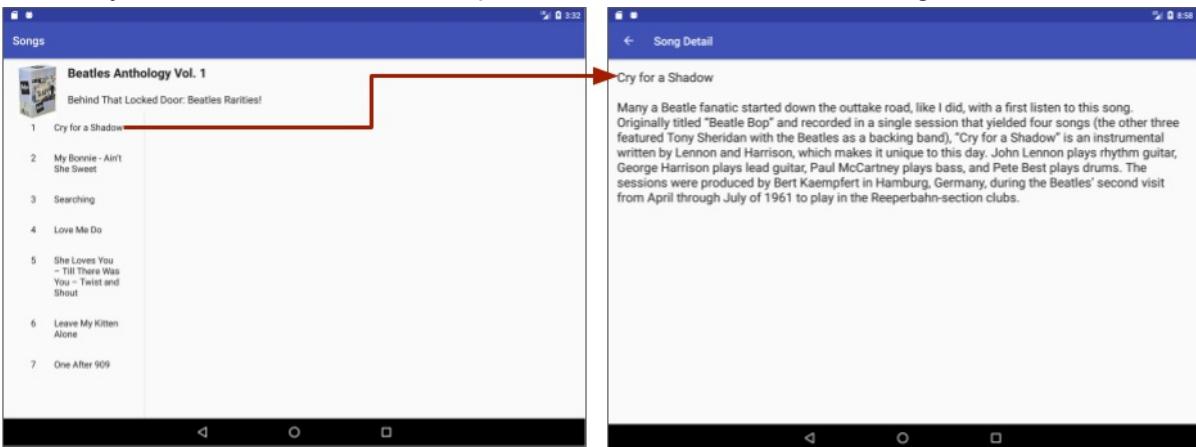
In the above figure:

1. Phone or tablet: The `SongDetail_start` app displays the song details in a vertical layout in `SongDetailActivity`, which is called from `MainActivity`.
2. Phone or small screen: The final version of `SongDetail` displays the song details in `SongDetailFragment`. `MainActivity` calls `SongDetailActivity`, which then hosts the `Fragment` in a vertical layout.
3. Tablet or larger screen in horizontal orientation: If the screen is wide enough for the master/detail layout, the final version of `SongDetail` displays the song details in `SongDetailFragment`. `MainActivity` hosts the `Fragment` directly.

## 2.1 Examine the starter app layout

To save time, download the [SongDetail\\_start](#) starter app, which has been prepared with data, layouts, and a `RecyclerView`.

1. Open the [SongDetail\\_start](#) app in Android Studio, and rename and refactor the project to `SongDetail` (for help with copying projects and refactoring and renaming, see "Copy and rename a project").
2. Run the app on a tablet or a tablet emulator in horizontal orientation. For instructions on using the emulator, see [Run Apps on the Android Emulator](#). The starter app uses the same layout for tablets and mobile phones—it doesn't take advantage of a wide screen.



3. Examine the layouts. Although you don't need to change them, you will reference the `android:id` values in your code.

The `song_list.xml` layout is included within `activity_song_list.xml` to define the layout of the song list. You can expand it to show:

- `song_list.xml` as the default for any screen size.
- `song_list.xml (w900dp)` for devices with screens that have a width of 900dp or larger. It differs from `song_list.xml` because it includes a `FrameLayout` with an id of `song_detail_container` for displaying the `Fragment` on a wide screen.

The `activity_song_detail.xml` layout for `SongDetailActivity` includes `song_detail.xml`. Provided is a `FrameLayout` with the same id of `song_detail_container` for displaying the `Fragment` on a screen that is not wide.

The following layouts are also provided, which you don't have to change:

- `song_detail.xml` : Included within `activity_song_detail.xml` to define the layout of the `TextView` for the detailed song information.
- `activity_song_list.xml` : Layout for `MainActivity`. This layout includes `song_list.xml`.
- `song_list_content.xml` : Item layout for the `RecyclerView` adapter.

## 2.2 Examine the starter app code

Open `SongDetailActivity` and find the code in the `onCreate()` method that displays the song detail:

```
// ...
// This activity displays the detail. In a real-world scenario,
// get the data from a content repository.
mSong = SongUtils.SONG_ITEMS.get
    (getIntent().getIntExtra(SongUtils.SONG_ID_KEY, 0));

// Show the detail information in a TextView.
if (mSong != null) {
    ((TextView) findViewById(R.id.song_detail))
        .setText(mSong.details);
}
// ...
```

In the next step you will add a new `Fragment`, and copy the `if (mSong != null)` block with `setText()` to the new `Fragment`, so that the `Fragment` controls how the song detail is displayed.

The `SongUtils.java` class in the `content` folder creates an array of fixed entries for the song title and song detail information. You can modify this class to refer to different types of data. However, in a real-world production app, you would most likely get data from a repository or server, rather than hardcoding it in the app.

## 2.3 Add the fragment

Add a new blank `Fragment`, and move code from `SongDetailActivity` to the `Fragment`, so that the `Fragment` can take over the job of displaying the song detail.

1. Select the app package name within `java` in the **Project: Android** view, add a new

- Fragment (Blank)**, and name the `Fragment` **SongDetailFragment**. Uncheck the **Include fragment factory methods** and **Include interface callbacks** options.
- Open `SongDetailActivity`, and **Edit > Cut** the `mSong` variable declaration from the Activity. Some of the code in `SongDetailActivity` that relies on it will be underlined in red, but you will replace that code in subsequent steps.

```
public SongUtils.Song mSong;
```

- Open `SongDetailFragment`, and **Edit > Paste** the above declaration at the top of the class.
- In `SongDetailFragment`, remove all code in the `onCreateView()` method and change it to inflate the `song_detail.xml` layout:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                        Bundle savedInstanceState) {
    View rootView =
        inflater.inflate(R.layout.song_detail, container, false);
    // TODO: Show the detail information in a TextView.
    return rootView;
}
```

- To use the song detail in the `Fragment`, replace the `TODO` comment with the `if (mSong != null)` block from `SongDetailActivity`, which includes the `setText()` method to show the detail information in the `song_detail` `TextView`. You need to add `rootView` to use the `findViewById()` method; otherwise, the block is the same as the one formerly used in `SongDetailActivity`:

```
if (mSong != null) {
    ((TextView) rootView.findViewById(R.id.song_detail))
        .setText(mSong.details);
}
```

## 2.4 Check if the screen is wide enough for a two-pane layout

`MainActivity` in the starter app provides the data to a second `Activity` (`SongDetailActivity`) to display the song detail on a separate `Activity` display. To change the app to provide data for the `Fragment`, you will change the code that displays the song detail.

- If the display is wide enough for a two-pane layout, `MainActivity` will host the `Fragment`, and send the position of the selected song in the list directly to the `Fragment`.

- If the screen is *not* wide enough for a two-pane layout, `MainActivity` will use an intent with extra data—the position of the selected song—to start `SongDetailActivity`. `SongDetailActivity` will then host the `Fragment`, and send the position of the selected song to the `Fragment`.

In other words, the `Fragment` will take over the job of displaying the song detail. Therefore, your code needs to host the `Fragment` in `MainActivity` if the screen is wide enough for a two-pane display, or in `SongDetailActivity` if the screen is not wide enough.

Open `MainActivity`, and follow these steps:

- To serve as a check for the size of the screen, add a `private boolean` to the `MainActivity` class called `mTwoPane`:

```
private boolean mTwoPane = false;
```

- Add the following to the end of the `MainActivity onCreate()` method:

```
if (findViewById(R.id.song_detail_container) != null) {
    mTwoPane = true;
}
```

The above code checks for the screen size *and* orientation. The `song_detail_container` view for `MainActivity` will be present only if the screen's *width* is 900dp or larger, because it is defined only in the `song_list.xml (w900dp)` layout, not in the default `song_list.xml` layout for smaller screen sizes. If this view is present, then the `Activity` should be in two-pane mode.

If a tablet is set to portrait orientation, its width will most likely be lower than 900dp, and so it will not show a two-pane layout. If the tablet is set to horizontal orientation and its width is 900dp or larger, it will show a two-pane layout.

## 2.5 Use the fragment to show song detail

The `Fragment` needs to know which song title the user selected. To use the same best practice for creating an instance of a `Fragment`, as in the previous exercises, create a `newInstance()` factory method in the `Fragment`.

In the `newInstance()` method you can set a `Bundle` and use the `Fragment.setArguments(Bundle)` method to supply the construction arguments for the `Fragment`. In a following step, you will use the `Fragment.getArguments()` method in the `Fragment` to get the arguments supplied by `setArguments(Bundle)`.

- Open `SongDetailFragment`, and add the following method to it:

```
public static SongDetailFragment newInstance (int selectedSong) {  
    SongDetailFragment fragment = new SongDetailFragment();  
    // Set the bundle arguments for the fragment.  
    Bundle arguments = new Bundle();  
    arguments.putInt(SongUtils.SONG_ID_KEY, selectedSong);  
    fragment.setArguments(arguments);  
    return fragment;  
}
```

The above method receives the `selectedSong` (the integer position of the song title in the list), and creates the `arguments` `Bundle` with `SONG_ID_KEY` and `selectedSong`. It then uses `setArguments(arguments)` to set the arguments for the `Fragment`, and returns the `Fragment`.

2. Open `MainActivity`, and find the `onBindViewHolder()` method that implements a listener with `setOnClickListener()`. When the user taps a song title, the starter app code starts `SongDetailActivity` using an intent with extra data (the position of the selected song in the list):

```
holder.mView.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        Context context = v.getContext();  
        Intent intent = new Intent(context, SongDetailActivity.class);  
        intent.putExtra(SongUtils.SONG_ID_KEY,  
                        holder.getAdapterPosition());  
        context.startActivity(intent);  
    }  
});
```

This code sends the extra data—`SongUtils.SONG_ID_KEY` and `holder.getAdapterPosition()`—that `SongDetailActivity` uses to show the correct detail for the tapped song title. You will have to send this data to the new `Fragment`.

3. Change the code within the `onClick()` method to create a new instance of the `Fragment` for two-pane display, or to use the intent (as before) to launch the second Activity if *not* a two-pane display.

```

if (mTwoPane) {
    int selectedSong = holder.getAdapterPosition();
    SongDetailFragment fragment =
        SongDetailFragment.newInstance(selectedSong);
    getSupportFragmentManager().beginTransaction()
        .replace(R.id.song_detail_container, fragment)
        .addToBackStack(null)
        .commit();
} else {
    Context context = v.getContext();
    Intent intent = new Intent(context, SongDetailActivity.class);
    intent.putExtra(SongUtils.SONG_ID_KEY,
        holder.getAdapterPosition());
    context.startActivity(intent);
}

```

If `mTwoPane` is true, the code gets the selected song position (`selectedSong`) in the song title list, and passes it to the new instance of `SongDetailFragment` using the `newInstance()` method in the `Fragment`. It then uses `getSupportFragmentManager()` with a `replace` transaction to show a new version of the `Fragment`.

The transaction code for managing a `Fragment` should be familiar, as you performed such operations in a previous lesson. By replacing the `Fragment`, you can refresh with new data a `Fragment` that is already running.

If `mTwoPane` is false, the code does exactly the same thing it did in the starter app: it starts `SongDetailActivity` with an intent and `SONG_ID_KEY` and `holder.getAdapterPosition()` as extra data.

4. Open `SongDetailActivity`, and find the code in the `onCreate()` method that no longer works due to the removal of the `mSong` declaration. In the next step you will replace it.

```

// This activity displays the detail. In a real-world scenario,
// get the data from a content repository.
mSong = SongUtils.SONG_ITEMS.get
    (getIntent().getIntExtra(SongUtils.SONG_ID_KEY, 0));
// Show the detail information in a TextView.
if (mSong != null) {
    ((TextView) findViewById(R.id.song_detail))
        .setText(mSong.details);
}

```

Previously you cut the `if (mSong != null)` block that followed the above code and pasted it into the `Fragment`, so that the `Fragment` could display the song detail. You can now replace the above code in the next step so that `SongDetailActivity` will use the `Fragment` to display the song detail.

5. Replace the above code in `onCreate()` with the following code. It first checks if `savedInstanceState` is `null`, which means the `Activity` started but its state was not saved. If it is `null`, it creates an instance of the `Fragment`, passing it the `selectedSong`. (If `savedInstanceState` is *not* `null`, the `Activity` state has been saved—such as when the screen is rotated. In such cases, you don't need to add the `Fragment`.)

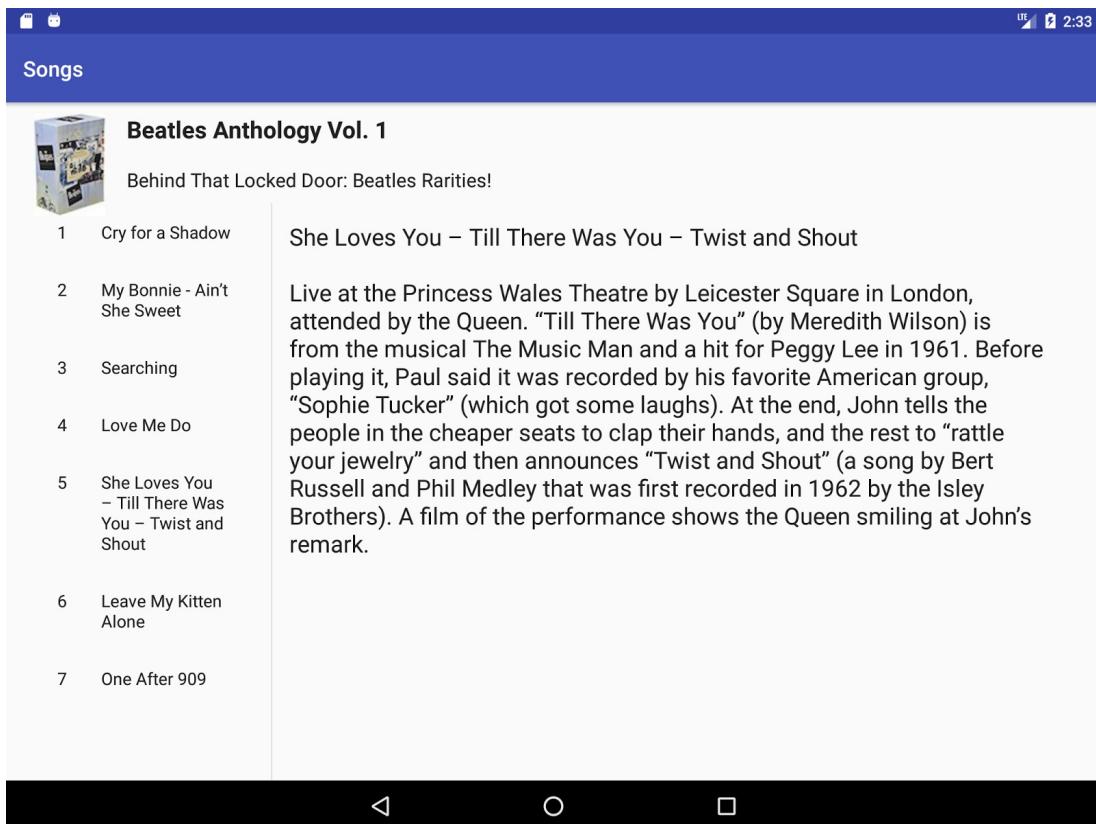
```
if (savedInstanceState == null) {
    int selectedSong =
        getIntent().getIntExtra(SongUtils.SONG_ID_KEY, 0);
    SongDetailFragment fragment =
        SongDetailFragment.newInstance(selectedSong);
    getSupportFragmentManager().beginTransaction()
        .add(R.id.song_detail_container, fragment)
        .commit();
}
```

The code first gets the selected song title position from the intent extra data. It then creates an instance of the `Fragment` and adds it to the `Activity` using a `Fragment` transaction. `SongDetailActivity` will now use the `SongDetailFragment` to display the detail.

6. To set up the data in the `Fragment`, open `SongDetailFragment` and add the entire `onCreate()` method *before* the `onCreateView()` method. The `getArguments()` method in the `onCreate()` method gets the arguments supplied to the `Fragment` using `setArguments(Bundle)`.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (getArguments().containsKey(SongUtils.SONG_ID_KEY)) {
        // Load the content specified by the fragment arguments.
        mSong = SongUtils.SONG_ITEMS.get(getArguments()
            .getInt(SongUtils.SONG_ID_KEY));
    }
}
```

7. Run the app on a mobile phone or phone emulator. It should look the same as it did before. (Refer to the figure [at the beginning of this task](#).)  
 8. Run the app on a tablet or tablet emulator in horizontal orientation. It should display the master/detail layout as shown in the following figure.



## Task 2 solution code

Android Studio project: [SongDetail](#)

## Summary

Adding a `Fragment` dynamically:

- When adding a `Fragment` dynamically to an `Activity`, the best practice for representing the `Fragment` as an instance in the `Activity` is to create the instance with a `newInstance()` factory method in the `Fragment`.
- The `newInstance()` method can set a `Bundle` and use `setArguments(Bundle)` to supply the construction arguments for the `Fragment`.
- Call the `newInstance()` method from the `Activity` to create a new instance, and pass the specific data you need for this `Bundle`.

Fragment lifecycle:

- The system calls `onAttach()` when a `Fragment` is first associated with an `Activity`. Use `onAttach()` to initialize essential components of the `Fragment`, such as a listener.
- The system calls `onCreate()` when creating a `Fragment`. Use `onCreate()` to initialize components of the `Fragment` that you want to retain when the `Fragment` is paused or

stopped, then resumed.

- The system calls `onCreateView()` to draw a `Fragment` UI for the first time. To draw a UI for your `Fragment`, you must return the root `View` of your `Fragment` layout from this method. You can return `null` if the `Fragment` does not provide a UI.
- When a `Fragment` is in the active or resumed state, it can access the host `Activity` instance with `getActivity()` and easily perform tasks such as finding a `view` in the `Activity` layout.

Calling `Fragment` methods and saving its state:

- The host `Activity` can call methods in a `Fragment` by acquiring a reference to the `Fragment` from `FragmentManager`, using `findFragmentById()`.
- Save the `Fragment` state during the `onSaveInstanceState()` callback and restore it during either `onCreate()`, `onCreateView()`, or `onActivityCreated()`.

To communicate from the host `Activity` to a `Fragment`, use a `Bundle` and the following:

- `setArguments(Bundle)` : Supply the construction arguments for a `Fragment`. The arguments are retained across the `Fragment` lifecycle.
- `getArguments()` : Return the arguments supplied to `setArguments(Bundle)`, if any.

To have a `Fragment` communicate to its host `Activity`, declare an interface in the `Fragment`, and implement it in the `Activity`.

- The interface in the `Fragment` defines a callback method to communicate to its host `Activity`.
- The host `Activity` implements the callback method.

## Related concept

The related concept documentation is [Fragment lifecycle and communications](#).

## Learn more

Android developer documentation:

- [Fragment](#)
- [Fragments](#)
- [FragmentManager](#)
- [FragmentTransaction](#)
- [Creating a Fragment](#)
- [Communicating with Other Fragments](#)

- The Activity Lifecycle
- Building a Flexible UI
- Building a Dynamic UI with Fragments
- Handling Configuration Changes
- Tasks and Back Stack

Videos:

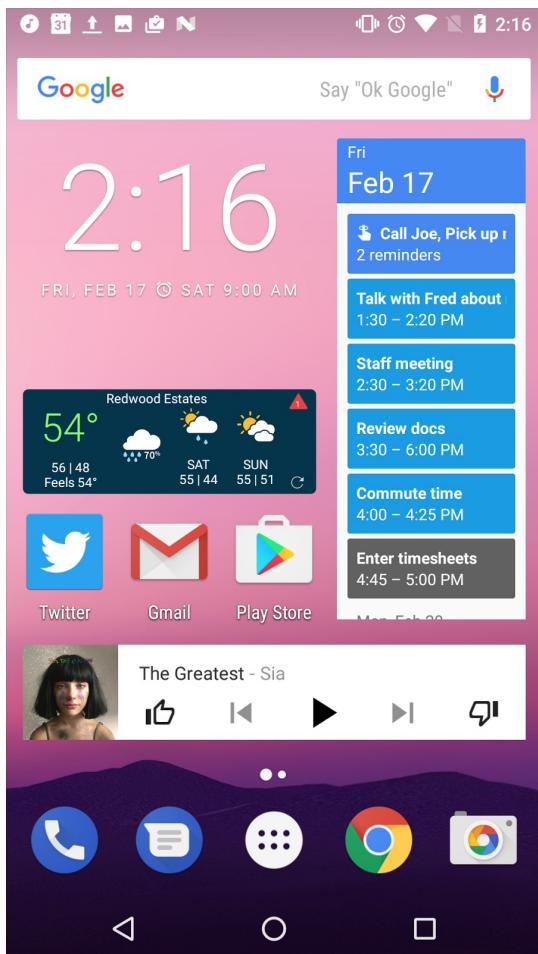
- [What the Fragment? \(Google I/O 2016\)](#)
- [Fragment Tricks \(Google I/O '17\)](#)
- [Por que Precisamos de Fragments?](#)

## 2.1: Building app widgets

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Set up the app widget project](#)
- [Task 2. Create the app widget layout](#)
- [Task 3. Add app widget updates and actions](#)
- [Solution code](#)
- [Coding challenge](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

An *app widget* is a miniature app view that appears on the Android home screen and can be updated periodically with new data. App widgets display small amounts of information or perform simple functions such as showing the time, summarizing the day's calendar events,



or controlling music playback.

App widgets are add-ons for an existing app and are made available to users when the app is installed on a device. Your app can have multiple widgets. You can't create a stand-alone app widget without an associated app.

Users can place widgets on any home screen panel from the widget picker. To access the widget picker, touch & hold any blank space on the home screen, and then choose **Widgets**. Touch & hold any widget to place it on the home screen. You can also touch & hold an already-placed widget to move or resize it, if it is resizeable.

**Note:** As of Android 5.0, widgets can only be placed on the Android home screen. Previous versions of Android (4.2/API 17 to 4.4/API 19) also allowed widgets to appear on the lock screen (keyguard). Although the app widget tools and APIs still occasionally mention lock screen widgets, that functionality is deprecated. This chapter discusses only the home screen widgets.

App widgets that display data can be updated periodically to refresh that data, either by the system or by the widget's associated app, through a broadcast intent. An app widget is a broadcast receiver that accepts those intents.

App widgets can also perform actions when tapped, such as launching their associated app. You can create click handlers to perform actions for the widget as a whole, or for any view of the widget layout such as a button.

In this practical, you build a simple widget that displays data, updates itself periodically, and includes a button to refresh the data on request.

**Note:** The term "widget" in Android also commonly refers to the user interface elements (views) you use to build an app, such as buttons and checkboxes. In this chapter all instances of the word widget refers to app widgets.

## What you should already KNOW

You should be familiar with:

- Creating, building, and running apps in Android Studio.
- Sending and receiving broadcast intents.
- Building and sending pending intents.

## What you will LEARN

You will learn how to:

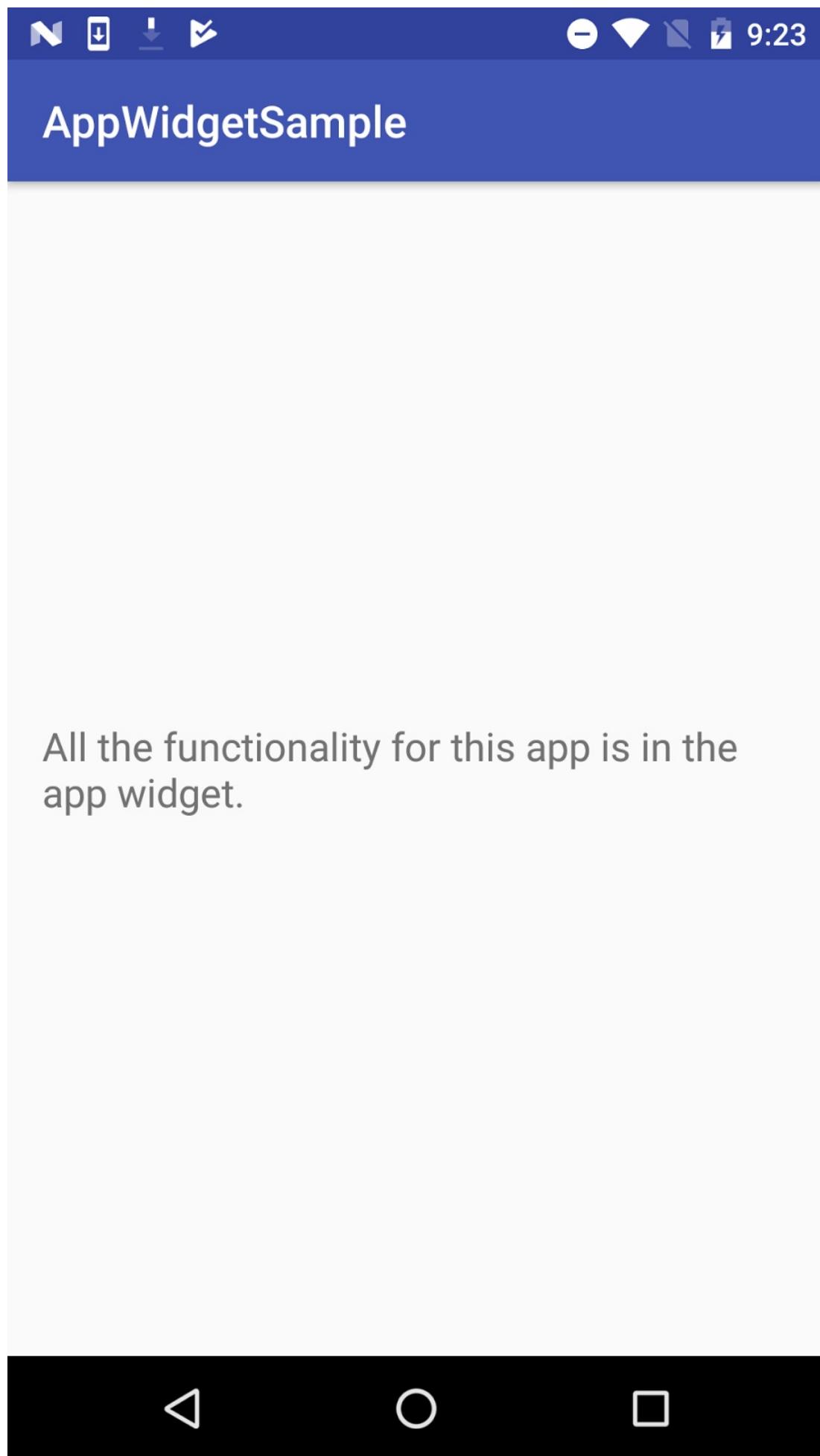
- Identify app widgets, and understand the key parts of an app widget.
- Add an app widget to your project with Android Studio.
- Understand the mechanics of app widget updates, and how to receive and handle update intents for your app widget.
- Implement app widget actions when an element of an app widget is tapped.

## What you will DO

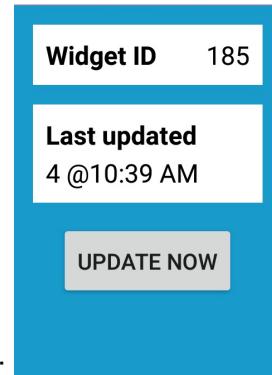
- Add an app widget to a starter project with Android Studio.
- Implement an app widget layout.
- Implement app widget updates, and learn the difference between periodic and requested updates.

## App overview

The AppWidgetSample app demonstrates a simple app widget. Because the app demonstrates app widgets, the app's main activity is minimal, with one explanatory text view:



All the functionality for this app is in the app widget.



The actual app widget has two panels for information and a button:

The panels display:

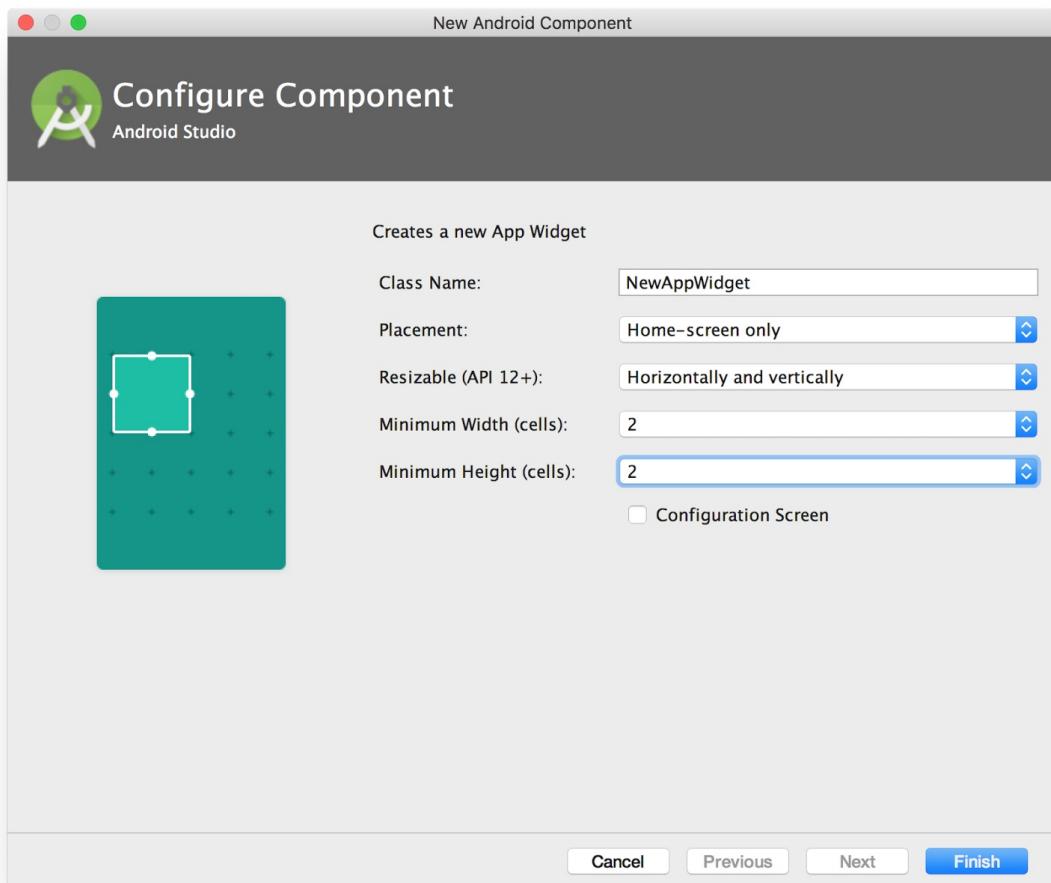
- The app widget ID. The user can place multiple widget instances on their home screen. An internal ID identifies each widget.
- How many times the widget has been updated, and the last update time.
- An **Update now** button, to request an immediate update.

## Task 1. Set up the app widget project

In this task, you open the starter app for the project, add a skeleton app widget to that app with Android Studio, and explore the files for that new app widget.

### 1.1 Build and run the app project

1. Create a new Android project. Call it `AppWidgetSample` and use the Empty activity template.
2. Open `res/layout/activity_main.xml`.
3. Change the `android:text` attribute of the `TextView` to read "All the functionality for this app is in the app widget." Extract the string.
4. Select **File > New > Widget > AppWidget**. Leave the widget name as `NewAppWidget`. Set the minimum width and minimum height to 2 cells. Leave all the other options as they are, and click **Finish**.



Android Studio generates all the template files you need for adding an app widget to your app. You explore those files in the next section.

5. Build and run the project in Android Studio.
6. When the app appears, tap the home button.
7. Touch & hold any empty space on the home screen, then tap **Widgets**. A list of available widgets appears. This screen is sometimes called the "widget picker."
8. Scroll down to **AppWidgetSample** section, and touch & hold the blue **EXAMPLE** widget. Place it in any empty spot on any home screen, and release.

The new widget appears on the home screen two cells wide and one cell high, the default widget size you defined when you created the widget. This sample widget doesn't do anything other than display the word EXAMPLE on a blue background.



9. Add another widget. You can add multiple widgets from the same app on your home screen. This functionality is mostly useful for widgets that can be configured to display customized information. For example, different weather widgets could display the weather in different locations.
10. Touch & hold either of the EXAMPLE widgets. Resize handles appear on the edges of the widget. You can now move the widget or change the size of the widget to take up more than the allotted cells on the home screen.
11. To remove an EXAMPLE widget from the device, touch & hold the widget and drag it to the word **Remove** at the top of the screen. Removing a widget only removes that particular widget instance. Neither the second widget nor the app are removed.

## 1.2 Explore the AppWidgetSample files

The skeleton EXAMPLE widget that Android Studio created for you generates many files in your project. In this task you explore the files.

1. Open `res/xml/new_app_widget_info.xml`.

This XML configuration file is usually called the *provider-info file*. The provider-info file defines several properties of your app widget, including the widget's layout file, default size, configuration activity (if any), preview image, and periodic update frequency (how often the widget updates itself, in milliseconds).

2. Open `res/layout/new_app_widget.xml`.

This file defines the layout of your app widget. App widget layouts are based on `RemoteViews` elements, rather than the normal `view` hierarchy, although you define them in XML the same way. Remote views provide a separate view hierarchy that can be displayed outside an app. Remote views include a limited subset of the available Android layouts and views.

3. Open `res/drawable/example_appwidget_preview.png`. This drawable is a PNG image that provides a preview of the widget itself. The preview image appears on the Android widget picker screen when the user installs your widget. The default preview image you get when you create the app widget is a blue rectangle with the word EXAMPLE. If there's no preview image, the icon for your app is used instead. The resource for the preview image is specified in the provider-info file.

4. Open `java/values/NewAppWidget.java`.

This file is the *widget provider*, the Java file that defines the behavior for your widget. The key task for a widget provider is to handle widget update intents. App widgets extend the `AppWidgetProvider` class, which in turn extends `BroadcastReceiver`.

5. Open `res/values/dimens.xml`.

This default dimensions file includes a value for the widget padding of 8 dp. App widgets look best with a little extra space around the edges so that the widgets do not display edge-to-edge on the user's home screen. Before Android 4.0 (API 14), your layout needed to include this padding. After API 14 the system adds the margin for you.

6. Open `res/values/dimens.xml(2)/dimen.xml (v14)`.

This dimensions file is used for Android API versions 14 and higher. The value of `widget-margin` in this file is 0 dp, because the system adds the margin for you.

7. Open `manifests/AndroidManifest.xml`.

In the `AndroidManifest.xml` file, the widget provider is defined as a `BroadcastReceiver` with the `<receiver>` tag, because the `AppWidgetProvider` class extends `BroadcastReceiver`. The definition also includes an intent filter with an action of `android.appwidget.action.APPWIDGET_UPDATE`, which indicates that this app widget listens to app widget update broadcast intents. Note the `android:resource` attribute of the `meta-data` tag, which specifies the widget provider-info file (`new_app_widget_info`).

## 1.3 Explore the provider-info file

Some metadata about the app widget is defined in the widget provider-info file. In this step, you explore that file and some of the metadata it contains.

1. Open `res/xml/new_app_widget_info.xml`.
2. Note the `android:initialLayout` attribute.

This attribute defines the layout resource that your app widget will use, in this case the `@layout/new_app_widget` layout file.

**Note:** The default provider-info file also includes an `android:initialKeyguardLayout` attribute. This attribute enables you to provide a different layout for keyguard (lock screen) widgets. Previous versions of Android (4.2/API 17 to 4.4/API 19) allowed widgets to appear on the lock screen. Although the app widget tools and APIs still occasionally mention lock screen widgets, that functionality is deprecated.

3. Note the `android:minHeight` and `android:minWidth` attributes.

These attributes define the minimum initial size of the widget, in dp. When you defined your widget in Android Studio to be 2 cells wide by 2 high, Android Studio fills in these values in the provider-info file. When your widget is added to a user's home screen, it is stretched both horizontally and vertically to occupy as many grid cells as satisfy the `minWidth` and `minHeight` values.

The rule for how many dp fit into a grid cell is based on the equation  $70 \times \text{grid\_size} - 30$ , where `grid_size` is the number of cells you want your widget to take up. Generally speaking, you can use this table to determine what your `minWidth` and `minHeight` should be:

# of cells (columns or rows)	<code>minWidth</code> or <code>minHeight</code>
1	40 dp
2	110 dp
3	180 dp
4	250 dp

## Task 2. Create the app widget layout

In this task, you learn how home screen cells translate to actual widget dimensions. You create the widget layout views in the layout editor, and you edit the widget provider to build and display the layout.

### 2.1 Add the widget layout

1. Open `res/layout/new_app_widget.xml`.
2. Note the value for `android:padding` in the top-level `RelativeLayout` element. This padding value is defined in the `dimens.xml` files (`@dimen/widget_margin`), and varies depending on the version of Android on which the widget is running.
3. Delete the existing `TextView` element. Add a `LinearLayout` element inside the `RelativeLayout` element with these attributes:

Attribute	Value
<code>android:id</code>	" <code>@+id/section_id</code> "
<code>android:layout_width</code>	" <code>match_parent</code> "
<code>android:layout_height</code>	" <code>wrap_content</code> "
<code>android:layout_alignParentLeft</code>	" <code>true</code> "
<code>android:layout_alignParentStart</code>	" <code>true</code> "
<code>android:layout_alignParentTop</code>	" <code>true</code> "
<code>android:orientation</code>	" <code>horizontal</code> "
<code>style</code>	" <code>@style/AppWidgetSection</code> "

This linear layout provides the appearance of a light-colored panel on top of a grey-blue background. The `AppWidgetSection` style does not yet exist and appears in red in Android Studio. (You add it later.)

4. Inside the `LinearLayout`, add a `TextView` with these attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/appwidget_id_label"</code>
<code>android:layout_width</code>	<code>"0dp"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:layout_weight</code>	<code>"2"</code>
<code>android:text</code>	<code>"Widget ID"</code>
<code>style</code>	<code>"@style/AppWidgetLabel"</code>

Extract the string for the text. This text view is the label for the widget ID. The `AppWidgetLabel` style does not yet exist.

5. Add a second `TextView` below the first one, and give it these attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/appwidget_id"</code>
<code>android:layout_width</code>	<code>"0dp"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:layout_weight</code>	<code>"1"</code>
<code>android:text</code>	<code>"XX"</code>
<code>style</code>	<code>"@style/AppWidgetText"</code>

You do not need to extract the string for the text in this text view, because the string is replaced with the actual ID when the app widget runs. As with the previous views, the `AppWidgetText` style is not yet defined.

6. Open `res/values/styles.xml`. Add the following code below the `AppTheme` styles to define `AppWidgetSection`, `AppWidgetLabel`, and `AppWidgetText`:

```

<style name="AppWidgetSection" parent="@android:style/Widget">
    <item name="android:padding">8dp</item>
    <item name="android:layout_marginTop">12dp</item>
    <item name="android:layout_marginLeft">12dp</item>
    <item name="android:layout_marginRight">12dp</item>
    <item name="android:background">&#64;android:color/white</item>
</style>

<style name="AppWidgetLabel" parent="AppWidgetText">
    <item name="android:textStyle">bold</item>
</style>

<style name="AppWidgetText" parent="Base.TextAppearance.AppCompat.Subhead">
    <item name="android:textColor">&#64;android:color/black</item>
</style>

```

7. Return to the app widget's layout file. Click the **Design** tab to examine the layout in the design editor. By default, Android Studio assumes that you are designing a layout for a regular activity, and it displays a default activity "skin" around your layout. Android Studio does not provide a preview for app widget designs.

**TIP:** To simulate a simple widget design, choose **Android Wear Square** from the device menu and resize the layout to be approximately 110 dp wide and 110 dp tall (the values of `minWidth` and `minHeight` in the provider-info file).

## 2.2 Build the widget views in the widget provider

The widget-provider class is a subclass of `AppWidgetProvider`. You must implement the `onUpdate()` method for every app widget. This method is called the first time the widget runs and again each time the widget receives an update request (a broadcast intent).

Implementing a widget update typically involves these tasks:

- Retrieve any new data that the app widget needs to display.
- Build a `RemoteViews` object from the app's context and the app widget's layout file.
- Update any views within the app widget's layout with new data.
- Tell the app widget manager to redisplay the widget with the new remote views.

Unlike activities, where you only inflate the layout once and then modify it in place as new data appears, the entire app widget layout must be reconstructed and redisplayed each time the widget receives an update intent.

1. Open `java/values/NewAppWidget.java`.
2. Scroll down to the `onUpdate()` method, and examine the method parameters.

The `onUpdate()` method is called with several arguments including the context, the app widget manager, and an array of integers that contains all the available app widget IDs.

Every app widget that the user adds to the home screen gets a unique internal ID that identifies that app widget. Each time you get an update request in your provider class, you must update all app widget instances by iterating over that array of IDs.

The template code that Android Studio defines for your widget provider's `onUpdate()` method iterates over that array of app widget IDs and calls the `updateAppWidget()` helper method.

```
@Override  
public void onUpdate(Context context,  
    AppWidgetManager appWidgetManager, int[] appWidgetIds) {  
    // There may be multiple widgets active, so update all of them  
    for (int appWidgetId : appWidgetIds) {  
        updateAppWidget(context, appWidgetManager, appWidgetId);  
    }  
}
```

When you use this template code for an app widget, you do not need to modify the actual `onUpdate()` method. Use the `updateAppWidget()` helper method to update each individual widget.

3. In the `updateAppWidget()` method, delete the line that gets the app widget text:

```
CharSequence widgetText =  
    context.getString(R.string.appwidget_text);
```

The template code for the app widget includes this string. You won't use it for this app widget.

4. Modify the arguments to the `views.setTextViewText()` method to update the `R.id.appwidget_id` view with the actual `appWidgetId`.

```
views.setTextViewText(R.id.appwidget_id, String.valueOf(appWidgetId));
```

5. Delete the `onEnabled()` and `onDisabled()` method stubs.

You would use `onEnabled()` to perform initial setup for a widget (such as opening a new database) when *the first instance* is initially added to the user's home screen. Even if the user adds multiple widgets, this method is only called once. Use `onDisabled()`,

correspondingly, to clean up any resources that were created in `onEnabled()` once *the last instance* of that widget is removed. You won't use either of these methods for this app, so you can delete them.

6. Build and run the app. When the app launches, go to the device's Home screen.

Delete the existing EXAMPLE widget from the home screen, and add a new widget. The preview for your app widget in the widget picker still uses an image that represents the EXAMPLE widget. When you place the new widget on a home screen, the widget should show the new layout with the internal widget ID. Note that the current height of the widget leaves a lot of space below the panel for the ID. You'll add more panels soon.



**Note:** Because app widgets are updated independently from their associated app, sometimes when you make changes to an app widget in Android Studio those changes do not show up in existing apps. When testing widgets make sure to remove all existing widgets before adding new ones.

7. Add a second copy of the app widget to the home screen. Note that each widget has its own widget ID.

## Task 3. Add app widget updates and actions

The data your app widget contains can be updated in two ways:

- The widget can update itself at regular intervals. You can define the interval in the widget's provider-info file.
- The widget's associated app can request a widget update explicitly.

In both these cases the app widget manager sends a broadcast intent with the action `ACTION_APPWIDGET_UPDATE`. Your app widget-provider class receives that intent, and calls the `onUpdate()` method.

### 3.1 Handle periodic updates

In this task, you add a second panel to the app widget that indicates how many times the widget has been updated, and the last update time.

1. In the widget layout file, add a second `LinearLayout` element just after the first, and give it these attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/section_update"</code>
<code>android:layout_width</code>	<code>"match_parent"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:layout_alignParentLeft</code>	<code>"true"</code>
<code>android:layout_alignParentStart</code>	<code>"true"</code>
<code>android:layout_below</code>	<code>"@+id/section_id"</code>
<code>android:orientation</code>	<code>"vertical"</code>
<code>style</code>	<code>"@style/AppWidgetSection"</code>

2. Inside the new `LinearLayout`, add a `TextView` with these attributes, and extract the string:

Attribute	Value
<code>android:id</code>	<code>"@+id/appwidget_update_label"</code>
<code>android:layout_width</code>	<code>"match_parent"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:layout_marginBottom</code>	<code>"2dp"</code>
<code>android:text</code>	<code>"Last Updated"</code>
<code>style</code>	<code>"@style/AppWidgetLabel"</code>

3. Add a second `TextView` after the first one and give it these attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/appwidget_update"</code>
<code>android:layout_width</code>	<code>"match_parent"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:layout_weight</code>	<code>"1"</code>
<code>android:text</code>	<code>"%1\$d @%2\$s"</code>
<code>style</code>	<code>"@style/AppWidgetText"</code>

Extract the string in `android:text` and give it the name `date_count_format`. The odd characters in this text string are placeholder formatting code. Parts of this string will be replaced in the Java code for your app, with the formatting codes filled in with numeric values. In this case the formatting code has four parts:

- `%1` : The first placeholder.
- `$d` : The format for the first placeholder value. In this case, a decimal number.
- `%2` : The second placeholder.
- `$s` : The format for the second placeholder value (a string).

The parts of the string that are not placeholders (here, just the `@` sign) are passed through to the new string. You can find out more about placeholders and formatting codes in the [Formatter](#) documentation.

**TIP:** To see the new widget in Android Studio's **Design** tab, you may need to enlarge the layout by dragging the lower-right corner downward.

4. In the app widget provider-info file (`res/xml/new_app_widget_info.xml`), change the `android:minHeight` attribute to `180dp`.

```
    android:minHeight="180dp"
```

When you add more content to the layout, the default size of the widget in cells on the home screen also needs to change. This iteration of the app widget is 3 cells high by 2 wide, which means `minHeight` is now 180 dp and `minWidth` remains 110 dp.

5. Also in the provider-info file, change `android:updatePeriodMillis` to `1800000`.

The `android:updatePeriodMillis` attribute defines how often the app widget is updated. The default update interval is 86,400,000 milliseconds (24 hours). 1,800,000 milliseconds is a 30 minute interval. If you set `updatePeriodMillis` to less than 1,800,000 milliseconds, the app widget manager only sends update requests every 30 minutes. Because updates use system resources, even if the associated app is not running, you should generally avoid frequent app widget updates.

6. In the app widget provider (`NewAppWidget.java`), add static variables to the top of the class for shared preferences. You'll use shared preferences to keep track of the current update count for the widget.

```
private static final String mSharedPrefFile =
    "com.example.android.appwidgetsample";
private static final String COUNT_KEY = "count";
```

7. At the top of the `updateAppWidget()` method, get the value of the update count from the shared preferences, and increment that value.

```
SharedPreferences prefs = context.getSharedPreferences(
    mSharedPrefFile, 0);
int count = prefs.getInt(COUNT_KEY + appWidgetId, 0);
count++;
```

The key to get the current count out of the shared preferences includes both the static key `COUNT_KEY` and the current app widget ID. Each app widget may have a different update count so each app widget needs its own entry in the shared preferences.

8. Get the current time and format it as a short string (`DateFormat.SHORT`):

```
String dateString =
    DateFormat.getTimeInstance(DateFormat.SHORT).format(new Date());
```

You will need to import both the `DateFormat` (`java.text.DateFormat`) and `Date` (`java.util.Date`) classes.

9. After updating the text view for `appwidget_id`, add a line to update the `appwidget_update` text view. This code gets the date format string from the resources and substitutes the formatting codes in the string with the actual values for the number of updates (`count`) and the current update time (`dateString`):

```
views.setTextViewText(R.id.appwidget_update,
    context.getResources().getString(
        R.string.date_count_format, count, dateString));
```

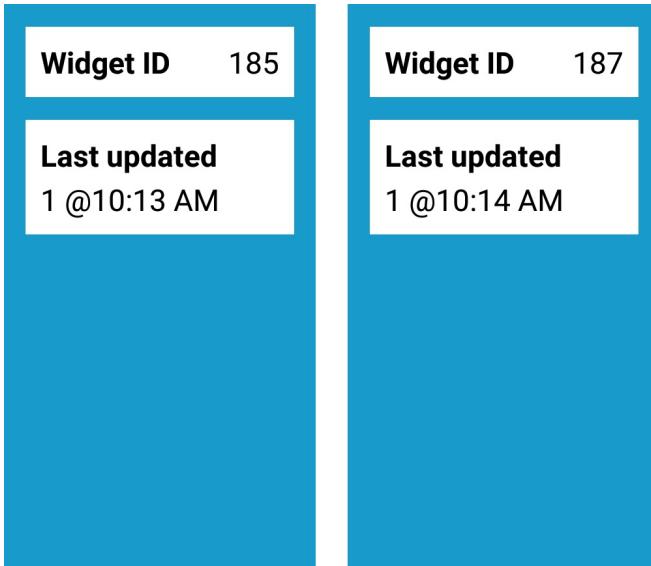
10. After constructing the `RemoteViews` object and before requesting the update from the app widget manager, put the current update count back into shared preferences:

```
SharedPreferences.Editor prefEditor = prefs.edit();
prefEditor.putInt(COUNT_KEY + appWidgetId, count);
prefEditor.apply();
```

As with the earlier lines where you retrieved the count from the shared preferences, here you store the count with a key that includes `COUNT_KEY` and the app widget ID, to differentiate between different counts.

11. Compile and run the app. As before, make sure you remove any existing widgets.

12. Add two widgets to the home screen, at least one minute apart. The widgets will have the same update count (1) but different update times.



At each half an hour interval after you add an app widget to the home screen, the Android app widget manager sends an update broadcast intent. Your widget provider accepts that intent, increments the count and updates the time for each widget. You could wait half an hour to see the widgets update itself, but in the next section you add a button that manually triggers an update.

The automatic update interval is timed from the *first* instance of the widget you placed on the home screen. Once that first widget receives an update request, then all the widget instances are updated at the same time.

## 3.2 Add an update button

In this last task, you add a button to the widget that explicitly requests a widget update with a broadcast intent. With this button you can update your widgets on request without waiting for the widget manager to get around to updating your widgets.

1. In the widget layout file, add a `Button` element just below the second `LinearLayout`, and give it these attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/button_update"</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/section_update"</code>
<code>android:layout_centerHorizontal</code>	<code>"true"</code>
<code>android:text</code>	<code>"Update now"</code>
<code>style</code>	<code>"@style/AppWidgetButton"</code>

Again, you may need to enlarge the widget in the **Design** tab.

2. In `styles.xml`, add this style for the button:

```
<style name="AppWidgetButton" parent="Base.Widget.AppCompat.Button">
    <item name="android:layout_marginTop">12dp</item>
</style>
```

3. In your `AppWidgetProvider` class, in the `updateAppWidget()` method, create an intent and set that intent's action to `AppWidgetManager.ACTION_APPWIDGET_UPDATE`. Add these lines just after you save the count to the shared preferences, and before the final call to `appWidgetManager.updateAppWidget()`.

```
Intent intentUpdate = new Intent(context, NewAppWidget.class);
intentUpdate.setAction(AppWidgetManager.ACTION_APPWIDGET_UPDATE);
```

The new intent is an explicit intent with the widget-provider class (`NewAppWidget.class`) as the target component.

4. After the lines to create the intent, create an array of integers with only one element: the current app widget ID.

```
int[] idArray = new int[]{appWidgetId};
```

5. Add an intent extra with the key `AppWidgetManager.EXTRA_APPWIDGET_IDS`, and the array you just created.

```
intentUpdate.putExtra(AppWidgetManager.EXTRA_APPWIDGET_IDS, idArray);
```

The intent needs an array of app widget IDs to update. In this case there's only the current widget ID, but that ID still needs to be wrapped in an array.

6. Use the `PendingIntent.getBroadcast()` method to wrap the intent as a pending intent that will perform a broadcast.

```
PendingIntent pendingUpdate = PendingIntent.getBroadcast(
    context, appWidgetId, intentUpdate,
    PendingIntent.FLAG_UPDATE_CURRENT);
```

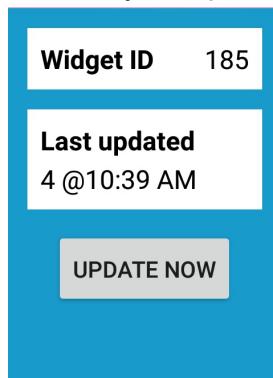
You use a pending intent here because the app widget manager sends the broadcast intent on your behalf.

- Set the `onClick` listener for the button to send the pending intent. Specifically for this action, the `RemoteViews` class provides a shortcut method called `setOnClickPendingIntent()`.

```
views.setOnClickListener(R.id.button_update, pendingUpdate);
```

**TIP:** In this step, a single view (the button) sends a pending intent. To have the entire widget send a pending intent, give an ID to the top-level widget layout view. Specify that ID as the first argument in the `setOnClickPendingIntent()` method.

- Compile and run the app. Remove all existing app widgets from the home screen, and add a new widget. When you tap the **Update now** button, both the update count and



the time update.

- Add a second widget. Confirm that tapping the **Update now** button in one widget only updates that particular widget and not the other widget.

## Solution code

Android Studio project: [AppWidgetSample](#)

## Coding challenge

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

**Challenge:** Update the widget preview image with a screenshot of the actual app widget.

**TIP:** The Android emulator includes an app called "Widget Preview" that helps create a preview image.

## Summary

- An *app widget* is a miniature app view that appears on the Android home screen. App

Widgets can be updated periodically with new data. To add app widgets to your app in Android Studio, use **File > New > Widget > AppWidget**. Android Studio generates all the template files you need for your app widget.

- The app widget provider-info file is in `res/xml/`. This file defines several properties of your app widget, including its layout file, default size, configuration activity (if any), preview image, and periodic update frequency.
- The app widget provider is a Java class that extends `AppWidgetProvider`, and implements the behavior for your widget—primarily handling managing widget update requests. The `AppWidgetProvider` class in turn inherits from `BroadcastReceiver`. Because widgets are broadcast receivers, the widget provider is defined as a broadcast receiver in the `AndroidManifest.xml` file with the `<receiver>` tag.
- App widget layouts are based on *remote views*, which are view hierarchies that can be displayed outside an app. Remote views provide a limited subset of the available Android layouts and views.
- When placed on the home screen, app widgets take up a certain number of cells on a grid. The cells correspond to a specific minimum width and height defined in the widget provider file. The rule for how many dp fit into a grid cell is based on the equation  $70 \times \text{grid\_size} - 30$ , where `grid_size` is the number of cells you want your widget to take up. In general, cells correspond to these dp values:

# of columns or rows	<code>minWidth</code> or <code>minHeight</code>
1	40 dp
2	110 dp
3	180 dp
4	250 dp

- App widgets can receive periodic requests to be updated through a broadcast intent. An app widget provider is a broadcast receiver which accepts those intents. To update an app widget, implement the `onUpdate()` method in your widget provider.
- The user may have multiple instances of your widget installed. The `onUpdate()` method should update all the available widgets by iterating over an array of widget IDs.
- The app widget layout is updated for new data in the `onUpdate()` method by rebuilding the widget's layout views (`RemoteViews`), and passing that `RemoteViews` object to the app widget manager.
- App widget actions are pending intents. Use the `onClickPendingIntent()` method to attach an app widget action to a view.
- Widget updates can be requested with a pending intent whose intent has the action `AppWidgetManager.ACTION_APPWIDGET_UPDATE`. An extra, `AppWidgetManager.EXTRA_APPWIDGET_IDS`, contains an array of app widget IDs to update.

## Related concept

The related concept documentation is in [App Widgets](#).

## Learn more

- [App Widgets](#)
- [App widget design guidelines](#)
- [Determining a size for your widget](#)
- [AppWidgetProvider class](#)
- [AppWidgetProviderInfo class](#)
- [AppWidgetManager class](#)
- [BroadcastReceiver class](#)
- [RemoteViews class](#)

# 3.1: Working with sensor data

## Contents:

- [Introduction](#)
- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. List the available sensors](#)
- [Task 2. Get sensor data](#)
- [Solution code](#)
- [Coding challenge](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

Many Android-powered devices include built-in sensors that measure motion, orientation, and environmental conditions such as ambient light or temperature. These sensors can provide data to your app with high precision and accuracy. Sensors can be used to monitor three-dimensional device movement or positioning, or to monitor changes in the environment near a device, such as changes to temperature or humidity. For example, a game might track readings from a device's accelerometer sensor to infer complex user gestures and motions, such as tilt, shake, or rotation.

In this practical you learn about the Android sensor framework, which is used to find the available sensors on a device and retrieve data from those sensors.

The device camera, fingerprint sensor, microphone, and GPS (location) sensor all have their own APIs and are not considered part of the Android sensor framework.

## What you should already KNOW

You should be familiar with:

- Creating, building, and running apps in Android Studio.
- Running and testing apps with the Android emulator.

## What you will LEARN

You will learn how to:

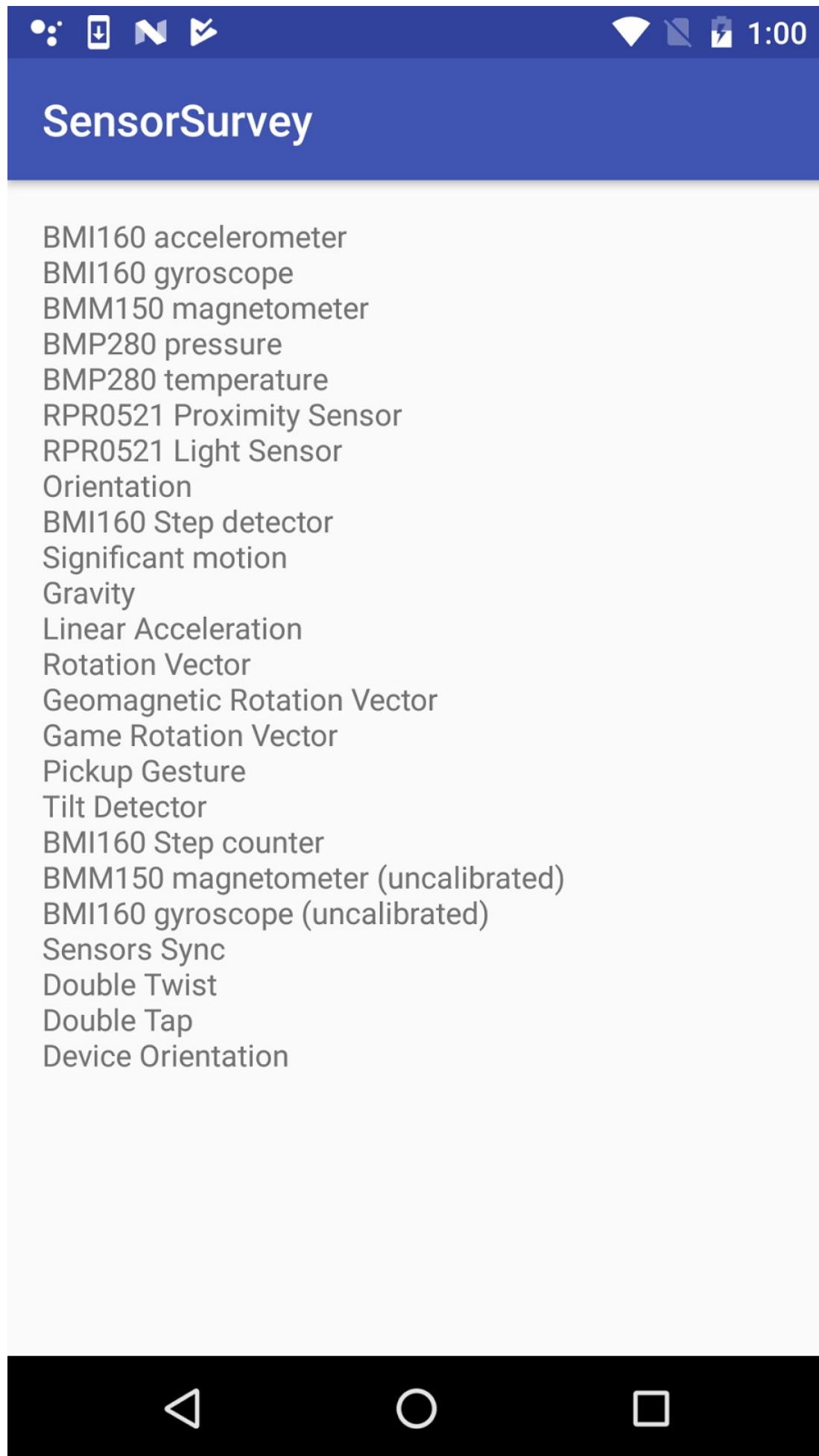
- Query the sensor manager for available sensors, and retrieve information about specific sensors.
- Register listeners for sensor data.
- React to incoming sensor data.

## What you will DO

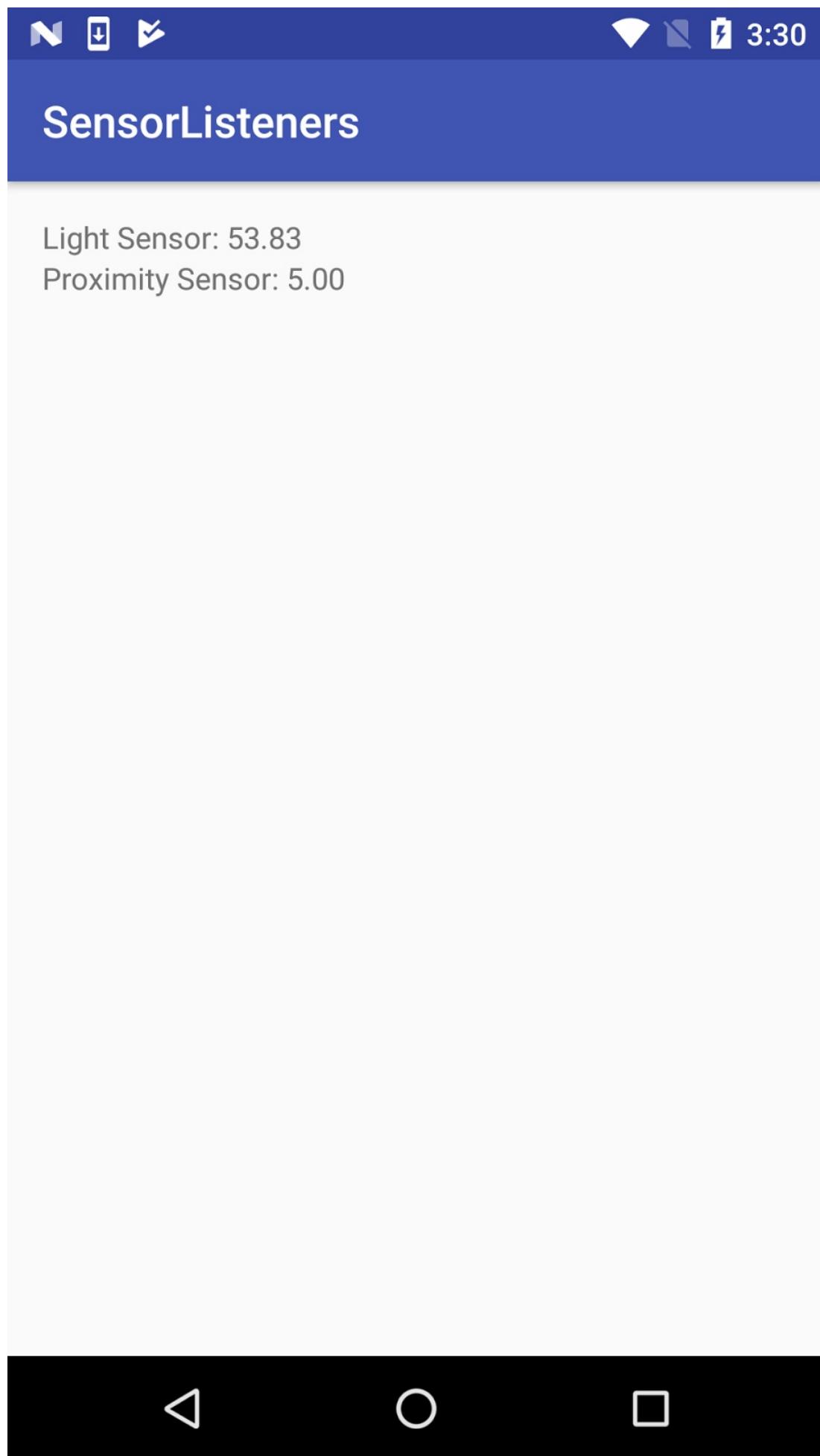
- Create an app that lists the available device sensors.
- Run the app on a device and on the emulator to view sensors.
- Create a second app that gets data from the light and proximity sensors, and displays that data.
- Interact with the device and note the changes in sensor data.
- Run the app in the emulator and learn about the emulator's virtual sensors.

## App overview

You will build two apps in this practical. The first app lists the available sensors on the device or emulator. The list of sensors is scrollable, if it is too big to fit the screen.



The second app, modified from the first, gets data from the ambient light and proximity sensors, and displays that data. Light and proximity sensors are some of the most common Android device sensors.



# Task 1. List the available sensors

In this task, you build a simple app that queries the sensor manager for the list of sensors available on the device.

## 1.1 Build the app

1. Create a new Android project. Call it SensorSurvey and use the Empty activity template.
2. Open `res/layout/activity_main.xml`.
3. Add a margin of 16 dp to the constraint layout.

```
    android:layout_margin="16dp"
```

4. Delete the existing `TextView`.
5. Add a `ScrollView` element inside the constraint layout. Give it these attributes:

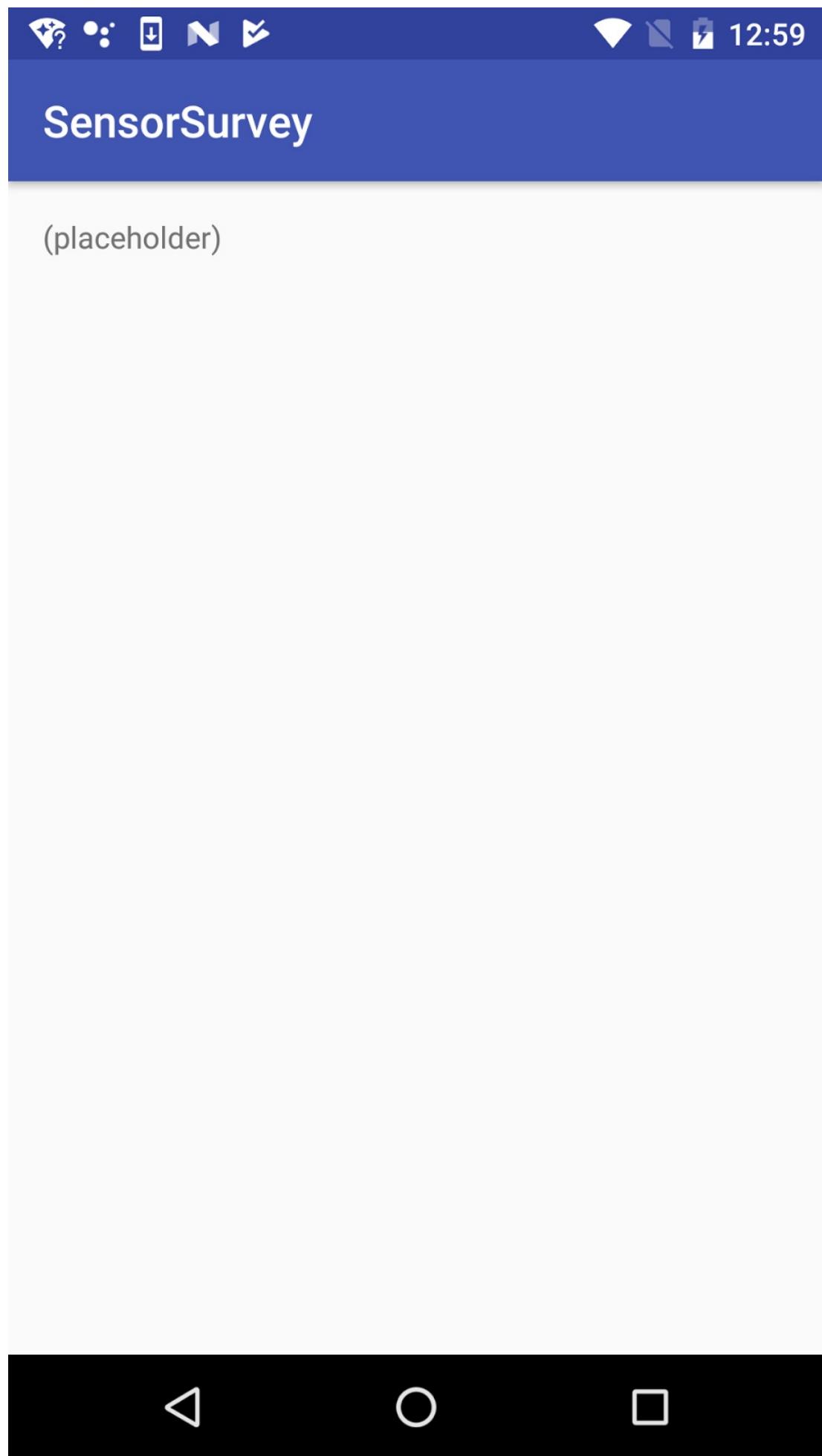
Attribute	Value
<code>android:layout_width</code>	"match_parent"
<code>android:layout_height</code>	"match_parent"
<code>app:layout_constraintBottom_toBottomOf</code>	"parent"
<code>app:layout_constraintTop_toTopOf</code>	"parent"
<code>app:layout_constraintLeft_toLeftOf</code>	"parent"
<code>app:layout_constraintRight_toRightOf</code>	"parent"

The `ScrollView` is here to allow the list of sensors to scroll if it is longer than the screen.

6. Add a `TextView` element inside the `ScrollView` and give it these attributes:

Attribute	Value
<code>android:id</code>	"@+id/sensor_list"
<code>android:layout_width</code>	"wrap_content"
<code>android:layout_height</code>	"wrap_content"
<code>android:text</code>	"(placeholder)"

This `TextView` holds the list of sensors. The placeholder text is replaced at runtime by the actual sensor list. The layout for your app should look like this screenshot:



7. Open `MainActivity` and add a variable at the top of the class to hold an instance of `SensorManager`:

```
private SensorManager mSensorManager;
```

The sensor manager is a system service that lets you access the device sensors.

8. In the `onCreate()` method, below the `setContentView()` method, get an instance of the sensor manager from system services, and assign it to the `mSensorManager` variable:

```
mSensorManager =  
(SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

9. Get the list of all sensors from the sensor manager. Store the list in a `List` object whose values are of type `Sensor`:

```
List<Sensor> sensorList =  
mSensorManager.getSensorList(Sensor.TYPE_ALL);
```

The `Sensor` class represents an individual sensor and defines constants for the available sensor types. The `Sensor.TYPE_ALL` constant indicates all the available sensors.

10. Iterate over the list of sensors. For each sensor, get that sensor's official name with the `getName()` method, and append that name to the `sensorText` string. Each line of the sensor list is separated by the value of the `line.separator` property, typically a newline character:

```
String sensorText = "";  
for (Sensor currentSensor : sensorList ) {  
    sensorText.append(currentSensor.getName()).append(  
        System.getProperty("line.separator"));  
}
```

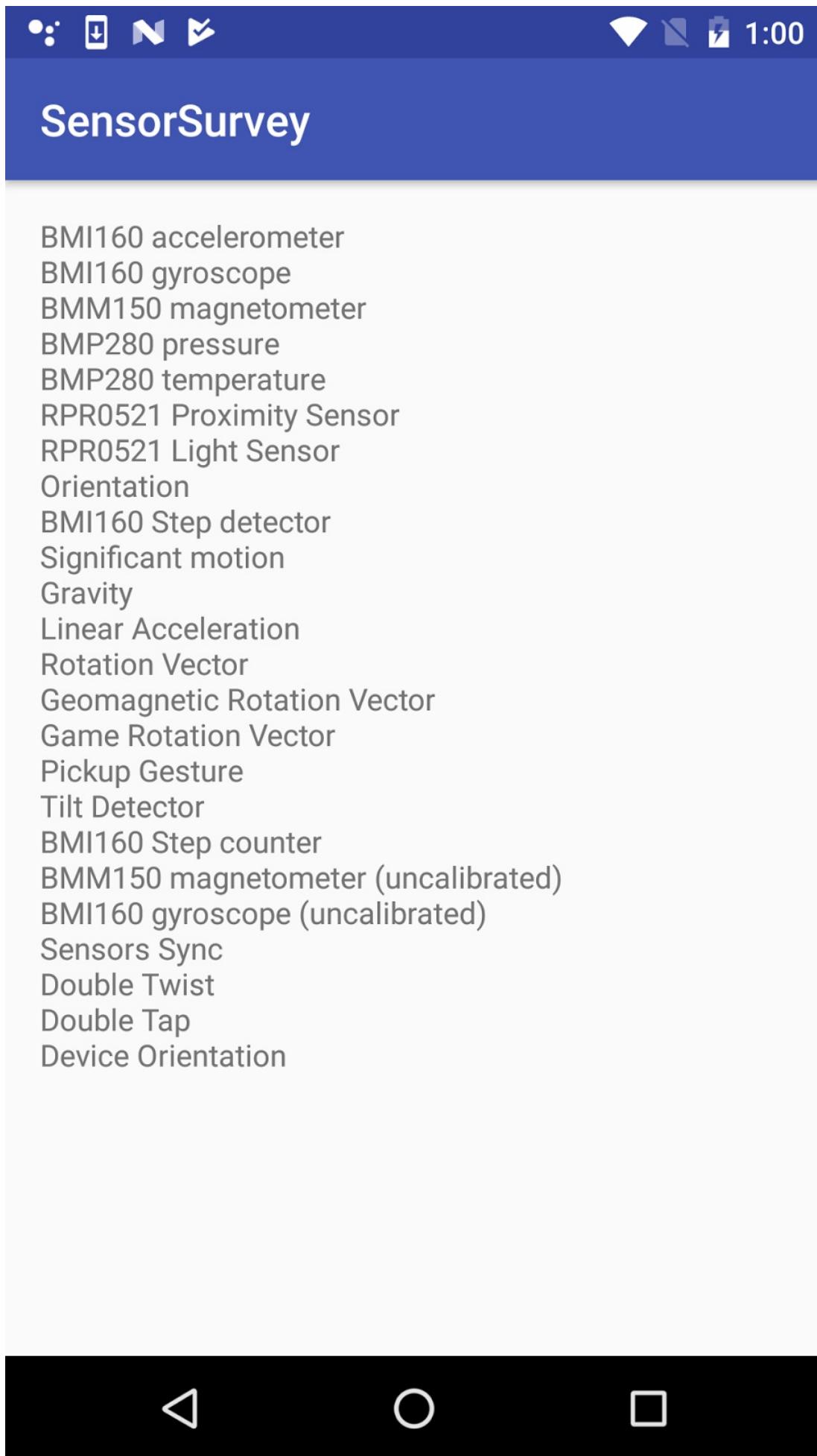
11. Get a reference to the `TextView` for the sensor list, and update the text of that view with the string containing the list of sensors:

```
TextView sensorTextView = (TextView) findViewById(R.id.sensor_list);  
sensorTextView.setText(sensorText);
```

## 1.2 Run the app on a device and in the emulator

Different Android devices have different sensors available, which means the SensorSurvey app shows different results for each device. In addition, the Android emulator includes a small set of simulated sensors.

1. Run the app on a physical device. The output of the app looks something like this screenshot:



In this list, lines that begin with a letter/number code represent physical hardware in the device. The letters and numbers indicate sensor manufacturers and model numbers. In most devices the accelerometer, gyroscope, and magnetometer are physical sensors.

Lines without letter/number codes are virtual or composite sensors, that is, sensors that are simulated in software. These sensors use the data from one or more physical sensors. So, for example, the gravity sensor may use data from the accelerometer, gyroscope, and magnetometer to provide the direction and magnitude of gravity in the device's coordinate system.

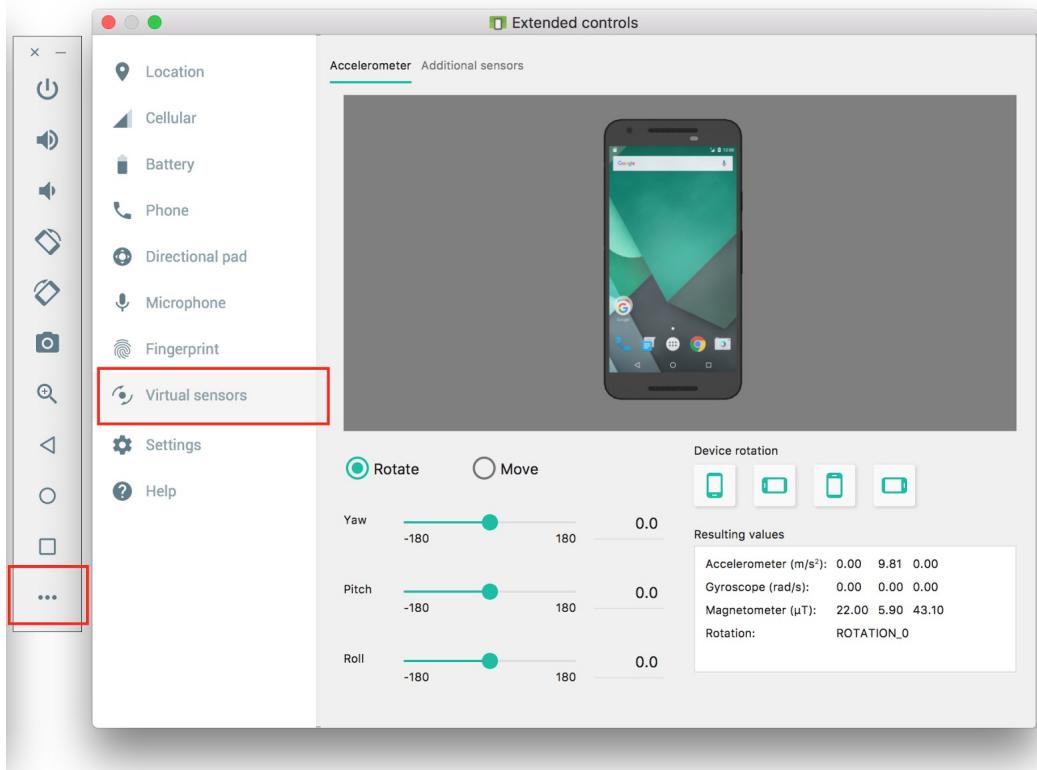
2. Run the app in an emulator. The output of the app looks something like this screenshot:



Because the Android emulator is a simulated device, all the available sensors are virtual sensors. "Goldfish" is the name of the emulator's Linux kernel.

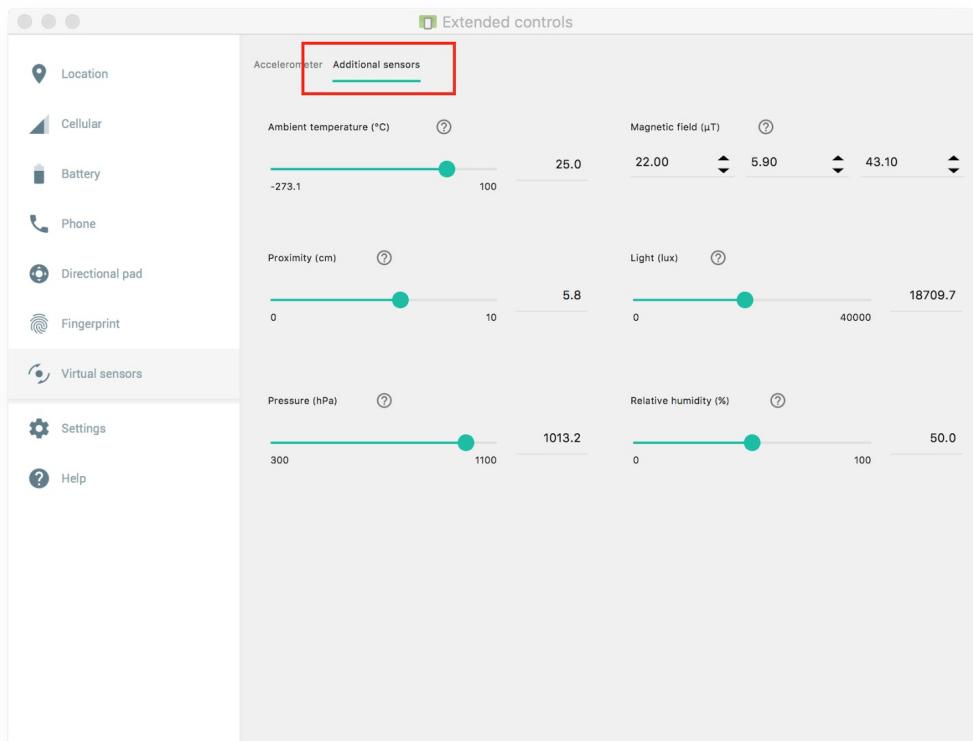
3. Click the **More** button (three horizontal dots) on the emulator's control panel. The **Extended Controls** window appears.

#### 4. Click **Virtual Sensors**.



This window shows the settings and current values for the emulator's virtual sensors. Drag the image of the device to simulate motion and acceleration with the accelerometer. Dragging the device image may also rotate the main emulator window.

#### 5. Click the **Additional Sensors** tab.



This tab shows the other available virtual sensors for the emulator, including the light, temperature, and proximity sensors. You use more of these sensors in the next task.

## Task 2. Get sensor data

The Android sensor framework provides the ability for your app to register for and react to changes in sensor data. In this task you modify your existing app to listen to and report values from the proximity and light sensors.

- The light sensor measures ambient light in `lux`, a standard unit of illumination. The light sensor typically is used to automatically adjust screen brightness.
- The proximity sensor measures when the device is close to another object. The proximity sensor is often used to turn off touch events on a phone's screen when you answer a phone call, so that touching your phone to your face does not accidentally launch apps or otherwise interfere with the device's operation.

### 2.1 Modify the layout

1. Open `res/layout/activity_main.xml`.
2. Delete the `ScrollView` and `TextView` elements from the previous app.
3. Add a `TextView` and give it the attributes in the following table. Extract the string into a resource called `"label_light"`. This text view will print the current value from the light sensor.

Attribute	Value
<code>android:id</code>	<code>"@+id/label_light"</code>
<code>android:layout_width</code>	<code>"wrap_content"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:text</code>	<code>"Light Sensor: %1\$.2f"</code>
<code>app:layout_constraintLeft_toLeftOf</code>	<code>"parent"</code>
<code>app:layout_constraintTop_toBottomOf</code>	<code>"parent"</code>

The `%1$.2f` part of the text string is a placeholder code. This code will be replaced in the Java code for your app with the placeholder filled in with an actual numeric value. In this case the placeholder code has three parts:

- `%1` : The first placeholder. You could include multiple placeholders in the same string with `%2` , `%3` , and so on.
- `.2` : The number format. In this case, `.2` indicates that the value should be formatted with only two digits after the decimal point.
- `f` : Indicates that the value to display is a floating-point number. Use `s` for string

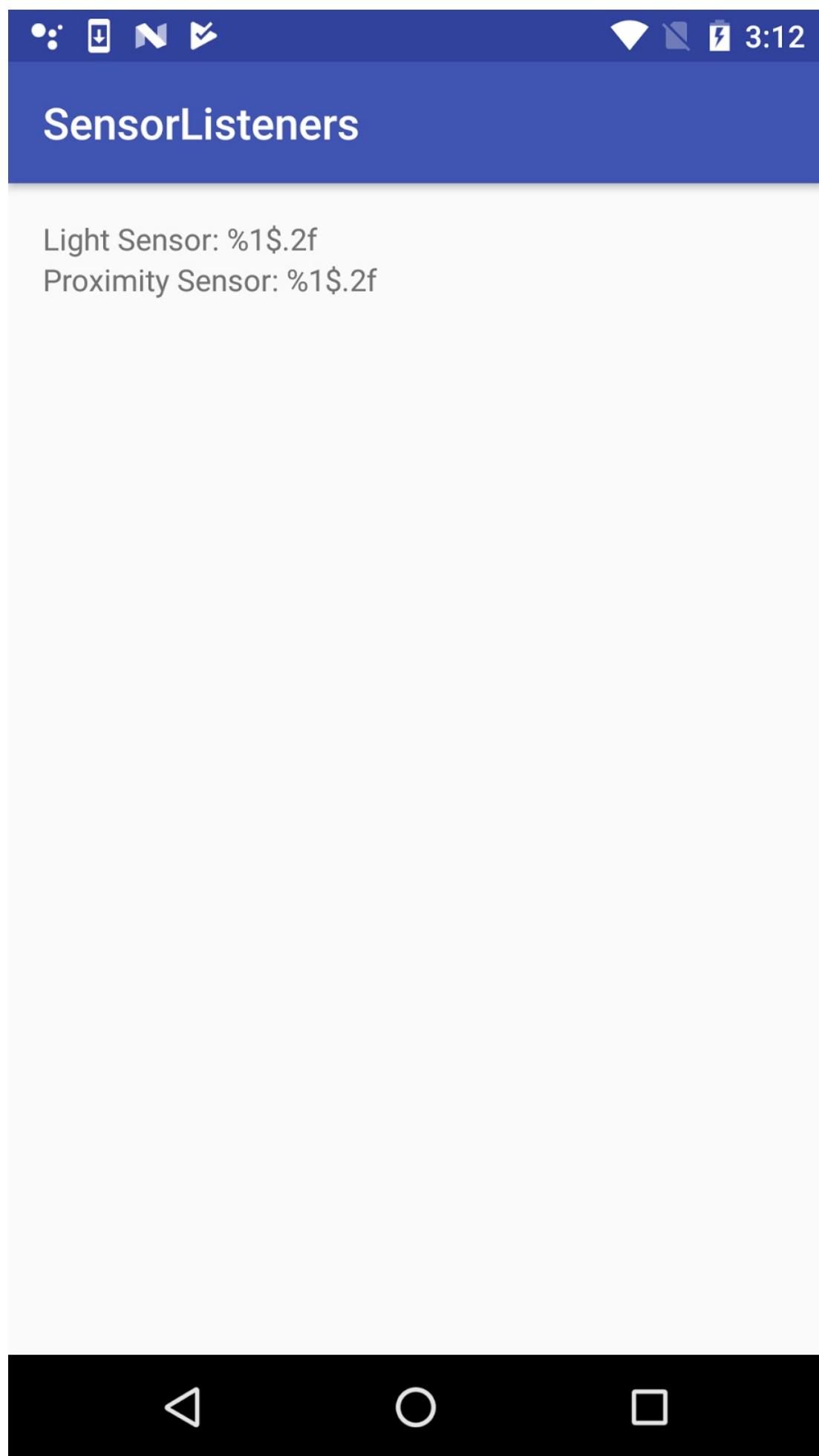
values and `d` for decimal values.

The part of the string that is not made up of placeholders ( "Light Sensor: " ) is passed through to the new string. You can find out more about placeholders and formatting codes in the [Formatter](#) documentation.

4. Copy and paste the `TextView` element. Change the attributes in the following table. Extract the string into a resource called `"label_proximity"`. This text view will print values from the proximity sensor.

Attribute	Value
<code>android:id</code>	<code>"@+id/label_proximity"</code>
<code>android:text</code>	<code>"Proximity Sensor: %1\$.2f"</code>
<code>app:layout_constraintTop_toBottomOf</code>	<code>"@+id/label_light"</code>

The layout for your app should look like this screenshot:



5. Open `res/values/strings.xml` and add this line:

```
<string name="error_no_sensor">No sensor</string>
```

You'll use this message in the next task when you test if a sensor is available.

## 2.2 Get the sensors

In this task, you modify the activity's `onCreate()` method to gain access to the light and proximity sensors.

1. Open `MainActivity` and add private member variables at the top of the class to hold `Sensor` objects for the light and proximity sensors. Also add private member variables to hold the `TextView` objects from the layout:

```
// Individual light and proximity sensors.  
private Sensor mSensorProximity;  
private Sensor mSensorLight;  
  
// TextViews to display current sensor values  
private TextView mTextSensorLight;  
private TextView mTextSensorProximity;
```

2. In the `onCreate()` method, delete all the existing code after the line to get the sensor manager.
3. Add code to `onCreate()` to get the two `TextView` views and assign them to their respective variables:

```
mTextSensorLight = (TextView) findViewById(R.id.label_light);  
mTextSensorProximity = (TextView) findViewById(R.id.label_proximity);
```

4. Get instances of the default light and proximity sensors. These will be instances of the `Sensor` class. Assign them to their respective variables:

```
mSensorProximity =  
    mSensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);  
mSensorLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
```

The `getDefaultSensor()` method is used to query the sensor manager for sensors of a given type. The sensor types are defined by the `Sensor` class. If there is no sensor available for the given type, the `getDefaultSensor()` method returns `null`.

5. Get the error string you defined earlier from the `strings.xml` resource:

```
String error = getResources().getString(R.string.error_no_sensor);
```

6. Test that there is an available light sensor. If the sensor is not available (that is, if `getDefaultSensor()` returned `null`), set the display text for the light sensor's `TextView` to the error string.

```
if (mSensorLight == null) {  
    mTextSensorLight.setText(sensor_error);  
}
```

Different devices have different sensors, so it is important that your app check that a sensor exists before using the sensor. If a sensor is not available, your app should turn off features that use that sensor and provide helpful information to the user. If your app's functionality relies on a sensor that is not available, your app should provide a message and gracefully quit. Do not assume that any device will have any given sensor.

7. Test for the existence of the proximity sensor.

```
if (mSensorProximity == null) {  
    mTextSensorProximity.setText(sensor_error);  
}
```

## 2.3 Listen for new sensor data

When sensor data changes, the Android sensor framework generates an event (a `SensorEvent`) for that new data. Your app can register listeners for these events, then handle the new sensor data in an `onSensorChanged()` callback. All of these tasks are part of the `SensorEventListener` interface.

In this task, you register listeners for changes to the light and proximity sensors. You process new data from those sensors and display that data in the app layout.

1. At the top of the class, modify the class signature to implement the `SensorEventListener` interface.

```
public class MainActivity  
    extends AppCompatActivity implements SensorEventListener {
```

2. Click the red light bulb icon, select "implement methods," and select all methods.

The `SensorEventListener` interface includes two callback methods that enable your app to handle sensor events:

- `onSensorChanged()` : Called when new sensor data is available. You will use this callback most often to handle new sensor data in your app.
- `onAccuracyChanged()` : Called if the sensor's accuracy changes, so your app can react to that change. Most sensors, including the light and proximity sensors, do not report accuracy changes. In this app, you leave `onAccuracyChanged()` empty.

3. Override the `onstart()` activity lifecycle method to register your sensor listeners.

Listening to incoming sensor data uses device power and consumes battery life. Don't register your listeners in `onCreate()`, as that would cause the sensors to be on and sending data (using device power) even when your app was not in the foreground. Use the `onStart()` and `onStop()` methods to register and unregister your sensor listeners.

```
@Override
protected void onStart() {
    super.onStart();

    if (mSensorProximity != null) {
        mSensorManager.registerListener(this, mSensorProximity,
            SensorManager.SENSOR_DELAY_NORMAL);
    }
    if (mSensorLight != null) {
        mSensorManager.registerListener(this, mSensorLight,
            SensorManager.SENSOR_DELAY_NORMAL);
    }
}
```

**Note:** The `onStart()` and `onStop()` methods are preferred over `onResume()` and `onPause()` to register and unregister listeners. As of Android 7.0 (API 24), apps can run in multi-window mode (split-screen or picture-in-picture mode). Apps running in this mode are paused, but still visible on screen. Use `onStart()` and `onStop()` to ensure that sensors continue running even if the app is in multi-window mode.

Each sensor that your app uses needs its own listener, and you should make sure that those sensors exist before you register a listener for them. Use the `registerListener()` method from the `SensorManager` to register a listener. This method takes three arguments:

- An app or activity `context`. You can use the current activity (`this`) as the context.
- The `Sensor` object to listen to.
- A delay constant from the `SensorManager` class. The delay constant indicates how quickly new data is reported from the sensor. Sensors can report a lot of data very quickly, but more reported data means that the device consumes more power. Make sure that your listener is registered with the minimum amount of new data it needs. In this example you use the slowest value (`SensorManager.SENSOR_DELAY_NORMAL`). For more data-intensive apps such as

games, you may need a faster rate such as `SENSOR_DELAY_GAME` or `SENSOR_DELAY_FASTEST`.

4. Implement the `onStop()` lifecycle method to unregister your sensor listeners when the app pauses:

```
@Override  
protected void onStop() {  
    super.onStop();  
    mSensorManager.unregisterListener(this);  
}
```

A single call to the `SensorManager.unregisterListener()` method unregisters all the registered listeners. Unregistering the sensor listeners in the `onStop()` method prevents the device from using power when the app is not visible.

5. In the `onSensorChanged()` method, get the sensor type.

```
int sensorType = sensorEvent.sensor.getType();
```

The `onSensorChanged()` method is called with a `SensorEvent` object. The `SensorEvent` object includes important properties of the event, such as which sensor is reporting new data, and the new data values. Use the `sensor` property of the `SensorEvent` to get a `Sensor` object, and then use `getType()` to get the type of that sensor. Sensor types are defined as constants in the `Sensor` class, for example, `Sensor.TYPE_LIGHT`.

6. Also in `onSensorChanged()`, get the sensor value.

```
float currentValue = sensorEvent.values[0];
```

The sensor event stores the new data from the sensor in the `values` array. Depending on the sensor type, this array may contain a single piece of data or a multidimensional array full of data. For example, the accelerometer reports data for the `x`-axis, `y`-axis, and `z`-axis for every change in the `values[0]`, `values[1]`, and `values[2]` positions. Both the light and proximity sensors only report one value, in `values[0]`.

7. Add a `switch` statement for the `sensorType` variable. Add a `case` for `Sensor.TYPE_LIGHT` to indicate that the event was triggered by the light sensor.

```
switch (sensorType) {  
    // Event came from the light sensor.  
    case Sensor.TYPE_LIGHT:  
        // Handle light sensor  
        break;  
    default:  
        // do nothing  
}
```

8. Inside the light sensor `case`, get the template string from the resources, and update the value in the light sensor's `TextView`.

```
mTextSensorLight.setText(getResources().getString(  
    R.string.label_light, currentValue));
```

When you defined this `TextView` in the layout, the original string resource included a placeholder code, like this:

```
Light Sensor: %1$.2f
```

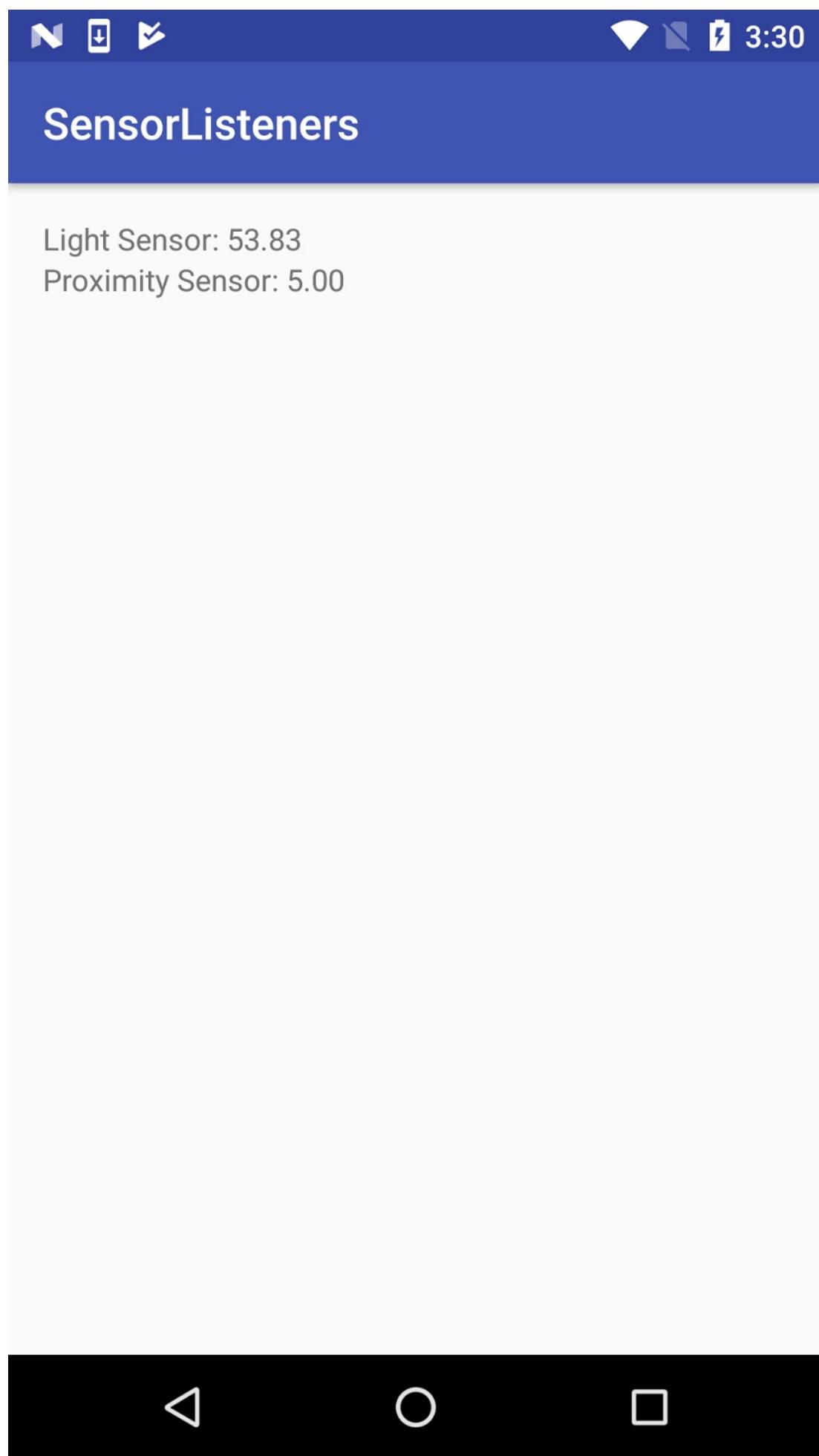
When you call `getString()` to get the string from the resources, you include values to substitute into the string where the placeholder codes are. The part of the string that is not made up of placeholders ("Light Sensor: ") is passed through to the new string.

9. Add a second `case` for the proximity sensor (`Sensor.TYPE_PROXIMITY`).

```
case Sensor.TYPE_PROXIMITY:  
    mTextSensorProximity.setText(getResources().getString(  
        R.string.label_proximity, currentValue));  
    break;
```

## 2.4 Run the app on a device and in the emulator

1. Run the app on a physical device. The output of the app looks something like this screenshot:



2. Move the device towards a light source, or shine a flashlight on it. Move the device away from the light or cover the device with your hand. Note how the light sensor reports changes in the light level.

**TIP:** The light sensor is often placed on the top right of the device's screen.

The light sensor's value is generally measured in lux, a standard unit of illumination. However, the lux value that a sensor reports may differ across different devices, and the maximum may vary as well. If your app requires a specific range of values for the light sensor, you must translate the raw sensor data into something your app can use.

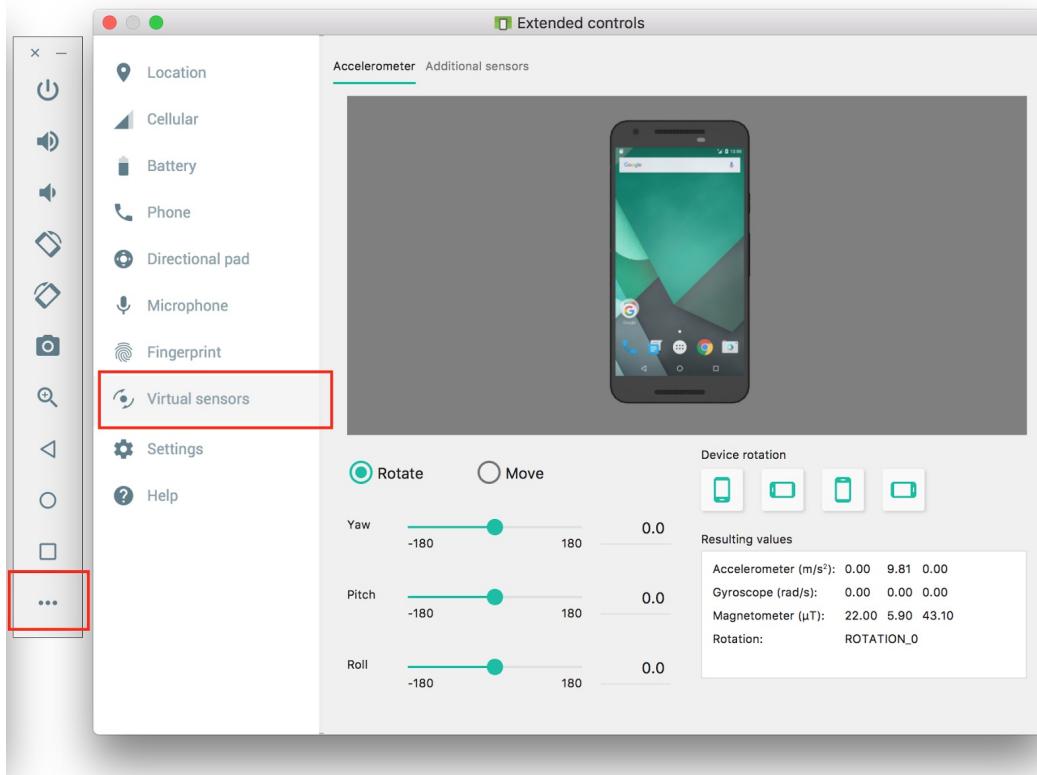
3. Move your hand toward the device, and then move it away again. Note how the proximity sensor reports values indicating "near" and "far." Depending on how the proximity sensor is implemented, you may get a range of values, or you may get just two values (for example, 0 and 5) to represent near and far.

**TIP:** The proximity sensor is often a virtual sensor that gets its data from the light sensor. For that reason, covering the light sensor may produce changes to the proximity value.

As with the light sensor, the sensor data for the proximity sensor can vary from device to device. Proximity values may be a range between a minimum and a maximum. More often there are only two proximity values, one to indicate "near," and one to indicate "far." All these values may vary across devices.

4. Run the app in an emulator, and click the **More** button (three horizontal dots) on the emulator's control panel to bring up the **Extended controls** window.

- Click **Virtual sensors**, and then click the **Additional sensors** tab.



The sliders in this window enable you to simulate changes to sensor data that would normally come from the hardware sensors. Changes in this window generate sensor events in the emulator that your app can respond to.

- Move the sliders for the light and proximity sensors and observe that the values in the app change as well.

## Solution code

Android Studio projects:

- [SensorSurvey](#)
- [SensorListeners](#)

## Coding challenge

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

**Challenge:** Modify the SensorListeners app such that:

- The background color of the app changes in response to the light level.

**TIP:** You can use `getWindow().getDecorView().setBackgroundColor()` to set the app's background color.

- Place an `ImageView` or `Drawable` in the layout. Make the image larger or smaller based on the value that the app receives from the proximity sensor.

## Summary

- The Android sensor framework provides access to data coming from a set of device sensors. These sensors include accelerometers, gyroscopes, magnetometers, barometers, humidity sensors, light sensors, proximity sensors, and so on.
- The `SensorManager` service lets your app access and list sensors and listen for sensor events (`SensorEvent`). The sensor manager is a system service you can request with `getSystemService()`.
- The `Sensor` class represents a specific sensor and contains methods to indicate the properties and capabilities of a given sensor. It also provides constants for sensor types, which define how the sensors behave and what data they provide.
- Use `getSensorList(Sensor.TYPE_ALL)` to get a list of all the available sensors.
- Use `getDefaultSensor()` with a sensor type to gain access to a particular sensor as a `Sensor` object.
- Sensors provide data through a series of sensor events. A `SensorEvent` object includes information about the sensor that generated it, the time, and new data. The data a sensor provides depends on the sensor type. Simple sensors such as light and proximity sensors report only one data value, whereas motion sensors such as the accelerometer provide multidimensional arrays of data for each event.
- Your app uses sensor listeners to receive sensor data. Implement the `SensorEventListener` interface to listen for sensor events.
- Use the `onSensorChanged()` method to handle individual sensor events. From the `SensorEvent` object passed into that method, you can get the sensor that generated the event and the new data.
- Register the sensor listeners in the `onResume()` lifecycle method, and unregister them in `onPause()`. Doing this prevents your app from drawing system resources when your app is not in the foreground.
- Use the `registerListener()` method to listen to sensor events. Listener registration includes both the type of sensor your app is interested in, and the rate at which your app prefers to receive data. A higher data rate provides more data events, but uses more system resources.
- Use the `unregisterListener()` method to stop listening to sensor events.

## Related concept

The related concept documentation is in [Sensor Basics](#).

## Learn more

Android developer documentation:

- [Sensors Overview](#)

Android API Reference:

- [Sensor](#)
- [SensorEvent](#)
- [SensorManager](#)
- [SensorEventListener](#)

## 3.2: Working with sensor-based orientation

### Contents:

- [Introduction](#)
- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Build the TiltSpot app](#)
- [Task 2. Add the spots](#)
- [Task 3. Handle activity rotation](#)
- [Solution code](#)
- [Coding challenge](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

The Android platform provides several sensors that enable your app to monitor the motion or position of a device, in addition to other sensors such as the light sensor.

Motion sensors such as the accelerometer or gyroscope are useful for monitoring device movement such as tilt, shake, rotation, or swing. Position sensors are useful for determining a device's physical position in relation to the Earth. For example, you can use a device's geomagnetic field sensor to determine its position relative to the magnetic north pole.

A common use of motion and position sensors, especially for games, is to determine the orientation of the device, that is, the device's bearing (north/south/east/west) and tilt. For example, a driving game could allow the user to control acceleration with a forward tilt or backward tilt, and control steering with a left tilt or right tilt.

Early versions of Android included an explicit sensor type for orientation ([`Sensor.TYPE\_ORIENTATION`](#)). The orientation sensor was software-only, and it combined data from other sensors to determine bearing and tilt for the device. Because of problems with the accuracy of the algorithm, this sensor type was deprecated in API 8 and may be unavailable in current devices. The recommended way to determine device orientation involves using both the accelerometer and geomagnetic field sensor and several methods in the [`SensorManager`](#) class. These sensors are common, even on older devices. This is the process you learn in this practical.

# What you should already KNOW

You should be familiar with:

- Creating, building, and running apps in Android Studio.
- Using the Android sensor framework to gain access to available sensors on the device, and to register and unregister listeners for those sensors.
- Using the `onSensorChanged()` method from the `SensorEventListener` interface to handle changes to sensor data.

# What you will LEARN

- What the accelerometer and magnetometer sensors do.
- The differences between the device-coordinate system and the Earth coordinate system, and which sensors use which systems.
- Orientation angles (azimuth, pitch, roll), and how they relate to other coordinate systems.
- How to use methods from the sensor manager to get the device orientation angles.
- How activity rotation (portrait or landscape) affects sensor input.

# What you will DO

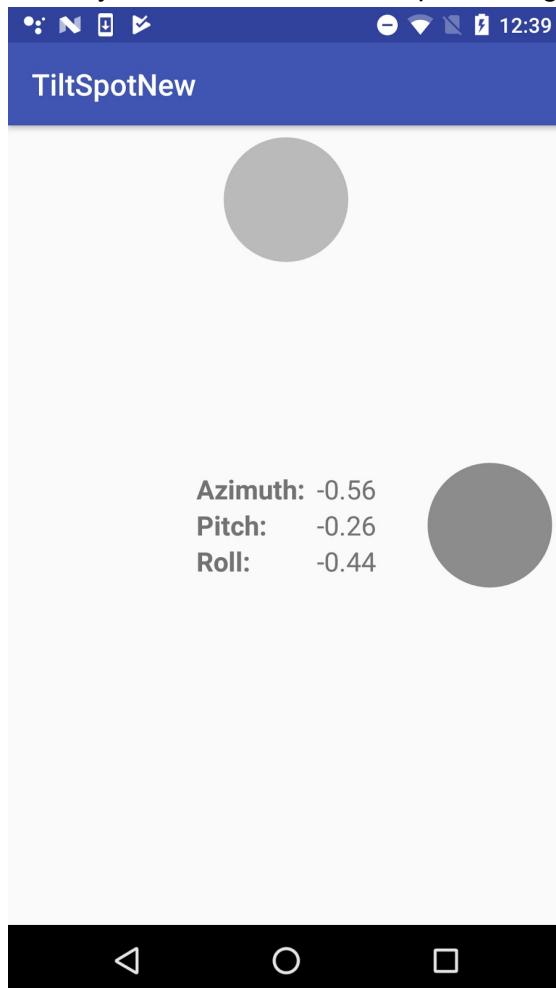
- Download and explore a starter app.
- Get the orientation angles from the accelerometer and magnetometer, and update text views in the activity to display those values.
- Change the color of a shape drawable to indicate which edge of the device is tilted up.
- Handle changes to the sensor data when the device is rotated from portrait to landscape.

# App overview

The TiltSpot app displays the device orientation angles as numbers and as colored spots along the four edges of the device screen. There are three components to device orientation:

- *Azimuth* : The direction (north/south/east/west) the device is pointing. 0 is magnetic north.
- *Pitch* : The top-to-bottom tilt of the device. 0 is flat.
- *Roll* : The left-to-right tilt of the device. 0 is flat.

When you tilt the device, the spots along the edges that are tilted up become darker.



## Task 1. Build the TiltSpot app

In this task you download and open the starter app for the project and explore the layout and activity code. Then you implement the `onSensorChanged()` method to get data from the sensors, convert that data into orientation angles, and report updates to sensor data in several text views.

### 1.1 Download and explore the starter app

1. Download the [TiltSpot\\_start](#) app and open it in Android Studio.
2. Open `res/layout/activity_main.xml`.

The initial layout for the TiltSpot app includes several text views to display the device orientation angles (azimuth, pitch, and roll)—you learn more about how these angles work later in the practical. All those textviews are nested inside their own constraint

layout to center them both horizontally and vertically within in the activity. You need the nested constraint layout because later in the practical you add the spots around the edges of the screen and around this inner text view.

3. Open `MainActivity`.

`MainActivity` in this starter app contains much of the skeleton code for managing sensors and sensor data as you learned about in the last practical.

1. Examine the `oncreate()` method.

This method gets an instance of the `SensorManager` service, and then uses the `getDefaultSensor()` method to retrieve specific sensors. In this app those sensors are the accelerometer (`Sensor.TYPE_ACCELEROMETER`) and the magnetometer (`Sensor.TYPE_MAGNETIC_FIELD`).

The *accelerometer* measures acceleration forces on the device; that is, it measures how fast the device is accelerating, and in which direction. Acceleration force includes the force of gravity. The accelerometer is sensitive, so even when you think you're holding the device still or leaving it motionless on a table, the accelerometer is recording minute forces, either from gravity or from the environment. This makes the data generated by the accelerometer very "noisy."

The *magnetometer*, also known as the *geomagnetic field sensor*, measures the strength of magnetic fields around the device, including Earth's magnetic field. You can use the magnetometer to find the device's position with respect to the external world. However, magnetic fields can also be generated by other devices in the vicinity, by external factors such as your location on Earth (because the magnetic field is weaker toward the equator), or even by solar winds.

Neither the accelerometer nor the magnetometer alone can determine device tilt or orientation. However, the Android sensor framework can combine data from both sensors to get a fairly accurate device orientation—accurate enough for the purposes of this app, at least.

2. At the top of `oncreate()`, note this line:

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
```

This line locks the activity in portrait mode, to prevent the app from automatically rotating the activity as you tilt the device. Activity rotation and sensor data can interact in unexpected ways. Later in the practical, you explicitly handle sensor data changes in your app in response to activity rotation, and remove this line.

3. Examine the `onstart()` and `onStop()` methods. The `onstart()` method registers the listeners for the accelerometer and magnetometer, and the `onStop()` method unregisters them.
4. Examine `onSensorChanged()` and `onAccuracyChanged()`. These are the methods from the `SensorEventListener` interface that you have to implement. The `onAccuracyChanged()` method is empty because it is unused in this class. You implement `onSensorChanged()` in the next task.

## 1.2 Get sensor data and calculate orientation angles

In this task you implement the `onSensorChanged()` method to get raw sensor data, use methods from the `SensorManager` to convert that data to device orientation angles, and update the text views with those values.

1. Open `MainActivity`.
2. Add member variables to hold copies of the accelerometer and magnetometer data.

```
private float[] mAccelerometerData = new float[3];
private float[] mMagnetometerData = new float[3];
```

When a sensor event occurs, both the accelerometer and the magnetometer produce arrays of floating-point values representing points on the `x`-axis, `y`-axis, and `z`-axis of the device's coordinate system. You will combine the data from both these sensors, and over several calls to `onSensorChanged()`, so you need to retain a copy of this data each time it changes.

3. Scroll down to the `onSensorChanged()` method. Add a line to get the sensor type from the sensor event object:

```
int sensorType = sensorEvent.sensor.getType();
```

4. Add tests for the accelerometer and magnetometer sensor types, and clone the event data into the appropriate member variables:

```
switch (sensorType) {  
    case Sensor.TYPE_ACCELEROMETER:  
        mAccelerometerData = sensorEvent.values.clone();  
        break;  
    case Sensor.TYPE_MAGNETIC_FIELD:  
        mMagnetometerData = sensorEvent.values.clone();  
        break;  
    default:  
        return;  
}
```

You use the `clone()` method to explicitly make a copy of the data in the `values` array. The `SensorEvent` object (and the array of values it contains) is reused across calls to `onSensorChanged()`. Cloning those values prevents the data you're currently interested in from being changed by more recent data before you're done with it.

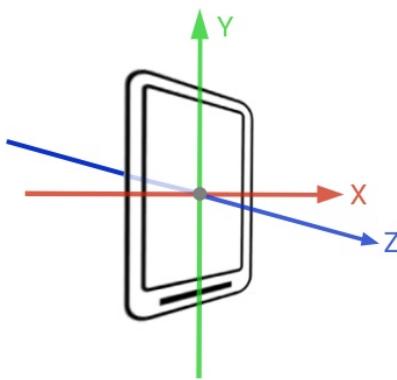
5. After the `switch` statement, use the `SensorManager.getRotationMatrix()` method to generate a rotation matrix (explained below) from the raw accelerometer and magnetometer data. The matrix is used in the next step to get the device orientation, which is what you're really interested in.

```
float[] rotationMatrix = new float[9];  
boolean rotationOK = SensorManager.getRotationMatrix(rotationMatrix,  
    null, mAccelerometerData, mMagnetometerData);
```

A **rotation matrix** is a linear algebra term that translates the sensor data from one coordinate system to another—in this case, from the device's coordinate system to the Earth's coordinate system. That matrix is an array of nine `float` values, because each point (on all three axes) is expressed as a 3D vector.

The device-coordinate system is a standard 3-axis (`x`, `y`, `z`) coordinate system relative to the device's screen when it is held in the default or natural orientation. Most sensors use this coordinate system. In this orientation:

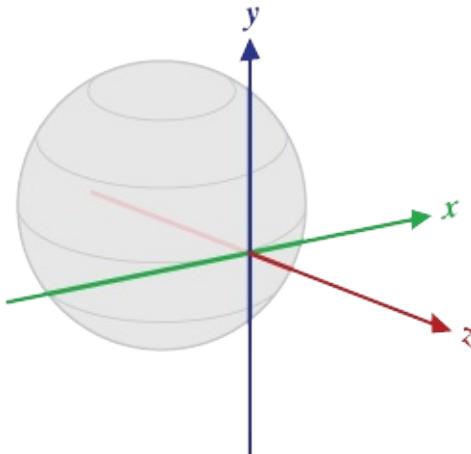
- The `x`-axis is horizontal and points to the right edge of the device.
- The `y`-axis is vertical and points to the top edge of the device.
- The `z`-axis extends up from the surface of the screen. Negative `z` values are



behind the screen.

The Earth's coordinate system is also a 3-axis system, but relative to the surface of the Earth itself. In the Earth's coordinate system:

- The y -axis points to magnetic north along the surface of the Earth.
- The x -axis is 90 degrees from y , pointing approximately east.
- The z -axis extends up into space. Negative z extends down into the ground.



A reference to the array for the rotation matrix is passed into the `getRotationMatrix()` method and modified in place. The second argument to `getRotationMatrix()` is an inclination matrix, which you don't need for this app. You can use null for this argument.

The `getRotationMatrix()` method returns a boolean (the `rotationOK` variable), which is `true` if the rotation was successful. The boolean might be `false` if the device is free-falling (meaning that the force of gravity is close to 0), or if the device is pointed very close to magnetic north. The incoming sensor data is unreliable in these cases, and the matrix can't be calculated. Although the boolean value is almost always `true`, it's good practice to check that value anyhow.

6. Call the `SensorManager.getOrientation()` method to get the orientation angles from the rotation matrix. As with `getRotationMatrix()`, the array of `float` values containing those angles is supplied to the `getOrientation()` method and modified in place.

```

float orientationValues[] = new float[3];
if (rotationOK) {
    SensorManager.getOrientation(rotationMatrix, orientationValues);
}

```

The angles returned by the `getOrientation()` method describe how far the device is oriented or tilted with respect to the Earth's coordinate system. There are three components to orientation:

- *Azimuth* : The direction (north/south/east/west) the device is pointing. 0 is magnetic north.
- *Pitch* : The top-to-bottom tilt of the device. 0 is flat.
- *Roll* : The left-to-right tilt of the device. 0 is flat.

All three angles are measured in radians, and range from  $-\pi$  (-3.141) to  $\pi$ .

7. Create variables for azimuth, pitch, and roll, to contain each component of the `orientationValues` array. You adjust this data later in the practical, which is why it is helpful to have these separate variables.

```

float azimuth = orientationValues[0];
float pitch = orientationValues[1];
float roll = orientationValues[2];

```

8. Get the placeholder strings, from the resources, fill the placeholder strings with the orientation angles and update all the text views.

```

mTextSensorAzimuth.setText(getResources().getString(
    R.string.value_format, azimuth));
mTextSensorPitch.setText(getResources().getString(
    R.string.value_format, pitch));
mTextSensorRoll.setText(getResources().getString(
    R.string.value_format, roll));

```

The string (`value_format` in `strings.xml`) contains placeholder code (`"%1$.2f"`) that formats the incoming floating-point value to two decimal places.

```
<string name="value_format">%1$.2f</string>
```

## 1.3 Build and run the app

1. Run the app. Place your device flat on the table. The output of the app looks something



like this:

Even a motionless device shows fluctuating values for the azimuth, pitch, and roll. Note also that even though the device is flat, the values for pitch and roll may not be 0. This is because the device sensors are extremely sensitive and pick up even tiny changes to the environment, both changes in motion and changes in ambient magnetic fields.

2. Turn the device on the table from left to right, leaving it flat on the table.



Note how the value of the azimuth changes. An azimuth value of 0 indicates that the device is pointing (roughly) north.

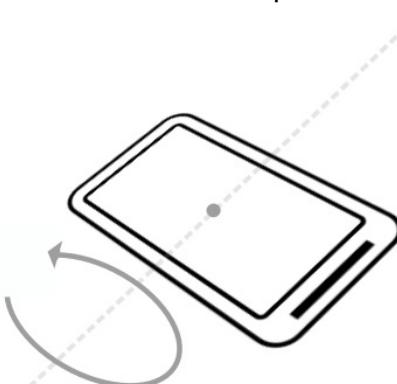
Note that even if the value of the azimuth is 0, the device may not be pointing exactly north. The device magnetometer measures the strength of any magnetic fields, not just that of the Earth. If you are in the presence of other magnetic fields (most electronics emit magnetic fields, including the device itself), the accuracy of the magnetometer may not be exact.

**Note:** If the azimuth on your device seems very far off from actual north, you can calibrate the magnetometer by waving the device a few times in the air [in a figure-eight](#)



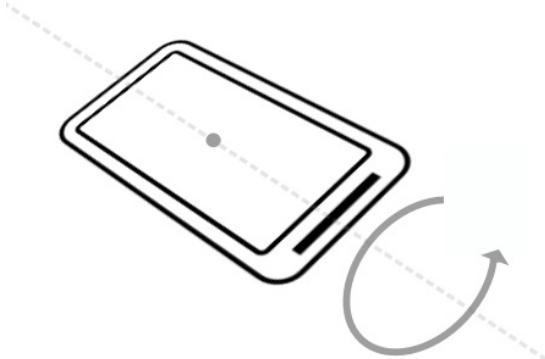
[motion](#).

3. Lift the bottom edge of the device so the screen is tilted away from you. Note the change to the pitch value. Pitch indicates the top-to-bottom angle of tilt around the



device's horizontal axis.

4. Lift the left side of the device so that it is tilted to the right. Note the change to the roll value. Roll indicates the left-to-right tilt along the device's vertical axis.



5. Pick up the device and tilt it in various directions. Note the changes to the pitch and roll values as the device's tilt changes. What is the maximum value you can find for any tilt direction, and in what device position does that maximum occur?

## Task 2. Add the spots

In this task you update the layout to include spots along each edge of the screen, and change the opaqueness of the spots so that they become darker when a given edge of the screen is tilted up.

The color changes in the spots rely on dynamically changing the alpha value of a shape drawable in response to new sensor data. The alpha determines the opacity of that drawable, so that smaller alpha values produce a lighter shape, and larger values produce a darker shape.

### 2.1 Add the spots and modify the layout

1. Add a new file called `spot.xml` to the project, in the `res/drawable` directory. (Create the directory if needed.)
2. Replace the selector tag in `spot.xml` with an oval shape drawable whose color is solid black (`"@android:color/black"`):

```
<shape  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:shape="oval">  
    <solid android:color="@android:color/black"/>  
</shape>
```

3. Open `res/values/dimens.xml`. Add a dimension for the spot size:

```
<dimen name="spot_size">84dp</dimen>
```

4. In `activity_layout.xml`, add an `ImageView` after the inner `ConstraintLayout`, and before the outer one. Use these attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/spot_top"</code>
<code>android:layout_width</code>	<code>"@dimen/spot_size"</code>
<code>android:layout_height</code>	<code>"@dimen/spot_size"</code>
<code>android:layout_margin</code>	<code>"@dimen/base_margin"</code>
<code>app:layout_constraintLeft_toLeftOf</code>	<code>"parent"</code>
<code>app:layout_constraintRight_toRightOf</code>	<code>"parent"</code>
<code>app:layout_constraintTop_toTopOf</code>	<code>"parent"</code>
<code>app:srcCompat</code>	<code>"@drawable/spot"</code>
<code>tools:ignore</code>	<code>"ContentDescription"</code>

This view places a spot drawable the size of the `spot_size` dimension at the top edge of the screen. Use the `app:srcCompat` attribute for a vector drawable in an `ImageView` (versus `android:src` for an actual image.) The `app:srcCompat` attribute is available in the [Android Support Library](#) and provides the greatest compatibility for vector drawables.

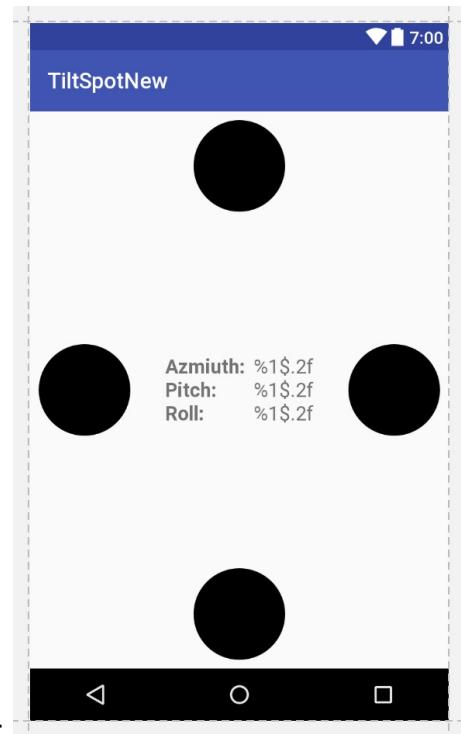
The `tools:ignore` attribute is used to suppress warnings in Android Studio about a missing content description. Generally `ImageView` views need alternate text for sight-impaired users, but this app does not use or require them, so you can suppress the warning here.

5. Add the following code below that first `ImageView`. This code adds the other three spots along the remaining edges of the screen.

```
<ImageView
    android:id="@+id/spot_bottom"
    android:layout_width="@dimen/spot_size"
    android:layout_height="@dimen/spot_size"
    android:layout_marginBottom="@dimen/base_margin"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:srcCompat="@drawable/spot"
    tools:ignore="ContentDescription" />

<ImageView
    android:id="@+id/spot_right"
    android:layout_width="@dimen/spot_size"
    android:layout_height="@dimen/spot_size"
    android:layout_marginEnd="@dimen/base_margin"
    android:layout_marginRight="@dimen/base_margin"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:srcCompat="@drawable/spot"
    tools:ignore="ContentDescription"/>

<ImageView
    android:id="@+id/spot_left"
    android:layout_width="@dimen/spot_size"
    android:layout_height="@dimen/spot_size"
    android:layout_marginLeft="@dimen/base_margin"
    android:layout_marginStart="@dimen/base_margin"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:srcCompat="@drawable/spot"
    tools:ignore="ContentDescription" />
```



The layout preview should now look like this:

6. Add the  `android:alpha` attribute to all four  `ImageView` elements, and set the value to `"0.05"`. The alpha is the opacity of the shape. Smaller values are less opaque (less visible). Setting the value to 0.05 makes the shape very nearly invisible, but you can still see them in the layout view.

## 2.2 Update the spot color with new sensor data

Next you modify the  `onSensorChanged( )` method to set the alpha value of the spots in response to the pitch and roll values from the sensor data. A higher sensor value indicates a larger degree of tilt. The higher the sensor value, the more opaque (the darker) you make the spot.

1. In  `MainActivity` , add member variables at the top of the class for each of the spot  `ImageView` objects:

```
private ImageView mSpotTop;
private ImageView mSpotBottom;
private ImageView mSpotLeft;
private ImageView mSpotRight;
```

2. In  `onCreate()` , just after initializing the text views for the sensor data, initialize the spot views:

```
mSpotTop = (ImageView) findViewById(R.id.spot_top);
mSpotBottom = (ImageView) findViewById(R.id.spot_bottom);
mSpotLeft = (ImageView) findViewById(R.id.spot_left);
mSpotRight = (ImageView) findViewById(R.id.spot_right);
```

3. In `onSensorChanged()`, right after the lines that initialize the azimuth, pitch, and roll variables, reset the pitch or roll values that are close to 0 (less than the value of the `VALUE_DRIFT` constant) to be 0:

```
if (Math.abs(pitch) < VALUE_DRIFT) {  
    pitch = 0;  
}  
if (Math.abs(roll) < VALUE_DRIFT) {  
    roll = 0;  
}
```

When you initially ran the TiltSpot app, the sensors reported very small non-zero values for the pitch and roll even when the device was flat and stationary. Those small values can cause the app to flash very light-colored spots on all the edges of the screen. In this code if the values are close to 0 (in either the positive or negative direction), you reset them to 0.

4. Scroll down to the end of `onSensorChanged()`, and add these lines to set the alpha of all the spots to 0. This resets all the spots to be invisible each time `onSensorChanged()` is called. This is necessary because sometimes if you tilt the device too quickly, the old values for the spots stick around and retain their darker color. Resetting them each time prevents these artifacts.

```
mSpotTop.setAlpha(0f);  
mSpotBottom.setAlpha(0f);  
mSpotLeft.setAlpha(0f);  
mSpotRight.setAlpha(0f);
```

5. Update the alpha value for the appropriate spot with the values for pitch and roll.

```
if (pitch > 0) {  
    mSpotBottom.setAlpha(pitch);  
} else {  
    mSpotTop.setAlpha(Math.abs(pitch));  
}  
if (roll > 0) {  
    mSpotLeft.setAlpha(roll);  
} else {  
    mSpotRight.setAlpha(Math.abs(roll));  
}
```

Note that the pitch and roll values you calculated in the previous task are in radians, and their values range from  $-\pi$  to  $+\pi$ . Alpha values, on the other hand, range only from 0.0 to 1.0. You could do the math to convert radian units to alpha values, but you may have

noted earlier that the higher pitch and roll values only occur when the device is tilted vertical or even upside down. For the TiltSpot app you're only interested in displaying dots in response to *some* device tilt, not the full range. This means that you can conveniently use the radian units directly as input to the alpha.

## 6. Build and run the app.

You should now be able to tilt the device and have the edge facing "up" display a dot which becomes darker the further up you tilt the device.

## Task 3. Handle activity rotation

The Android system itself uses the accelerometer to determine when the user has turned the device sideways (from portrait to landscape mode, for a phone). The system responds to this rotation by ending the current activity and recreating it in the new orientation, redrawing your activity layout with the "top," "bottom," "left," and "right" edges of the screen now reflecting the new device position.

You may assume that with TiltSpot, if you rotate the device from landscape to portrait, the sensors will report the correct data for the new device orientation, and the spots will continue to appear on the correct edges. That's not the case. When the activity rotates, the activity drawing coordinate system rotates with it, but the sensor coordinate system remains the same. The sensor coordinate system *never* changes position, regardless of the orientation of the device.

The second tricky point for handling activity rotation is that the default or natural orientation for your device may not be portrait. The default orientation for many tablet devices is landscape. The sensor's coordinate system is always based on the natural orientation of a device.

The TiltSpot starter app included a line in `onCreate()` to lock the orientation to portrait mode:

```
setRequestedOrientation (ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
```

Locking the screen to portrait mode in this way solves one problem—it prevents the coordinate systems from getting out of sync on portrait-default devices. But on landscape-default devices, the technique forces an activity rotation, which causes the device and sensor-coordinate systems to get out of sync.

Here's the right way to handle device and activity rotation in sensor-based drawing: First, use the `Display.getRotation()` method to query the current device orientation. Then use the `SensorManager.remapCoordinateSystem()` method to remap the rotation matrix from the sensor data onto the correct axes. This is the technique you use in the TiltSpot app in this task.

The `getRotation()` method returns one of four integer constants, defined by the `Surface` class:

- `ROTATION_0` : The default orientation of the device (portrait for phones).
- `ROTATION_90` : The "sideways" orientation of the device (landscape for phones). Different devices may report 90 degrees either clockwise or counterclockwise from 0.
- `ROTATION_180` : Upside-down orientation, if the device allows it.
- `ROTATION_270` : Sideways orientation, in the opposite direction from `ROTATION_90`.

Note that many devices do not have `ROTATION_180` at all or return `ROTATION_90` or `ROTATION_270` regardless of which direction the device was rotated (clockwise or counterclockwise). It is best to handle all possible rotations rather than to make assumptions for any particular device.

### 3.1 Get the device rotation and remap the coordinate system

1. In `MainActivity`, edit `onCreate()` to remove or comment out the call to

```
setRequestedOrientation().  
//setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
```

2. In `MainActivity`, add a member variable for the `Display` object.

```
private Display mDisplay;
```

3. At the end of `onCreate()`, get a reference to the window manager, and then get the default display. You use the display to get the rotation in `onSensorChanged()`.

```
WindowManager wm = (WindowManager) getSystemService(WINDOW_SERVICE);  
mDisplay = wm.getDefaultDisplay();
```

4. In `onSensorChanged()`, just after the call to `getRotationMatrix()`, add a new array of `float` values to hold the new adjusted rotation matrix.

```
float[] rotationMatrixAdjusted = new float[9];
```

5. Get the current device rotation from the display and add a `switch` statement for that

value. Use the rotation constants from the `Surface` class for each case in the switch. For `ROTATION_0`, the default orientation, you don't need to remap the coordinates. You can just clone the data in the existing rotation matrix:

```
switch (mDisplay.getRotation()) {
    case Surface.ROTATION_0:
        rotationMatrixAdjusted = rotationMatrix.clone();
        break;
}
```

6. Add additional cases for the other rotations, and call the `SensorManager.remapCoordinateSystem()` method for each of these cases.

This method takes as arguments the original rotation matrix, the two new axes on which you want to remap the existing `x`-axis and `y`-axis, and an array to populate with the new data. Use the axis constants from the `SensorManager` class to represent the coordinate system axes.

```
case Surface.ROTATION_90:
SensorManager.remapCoordinateSystem(rotationMatrix,
    SensorManager.AXIS_Y, SensorManager.AXIS_MINUS_X,
    rotationMatrixAdjusted);
break;
case Surface.ROTATION_180:
SensorManager.remapCoordinateSystem(rotationMatrix,
    SensorManager.AXIS_MINUS_X, SensorManager.AXIS_MINUS_Y,
    rotationMatrixAdjusted);
break;
case Surface.ROTATION_270:
SensorManager.remapCoordinateSystem(rotationMatrix,
    SensorManager.AXIS_MINUS_Y, SensorManager.AXIS_X,
    rotationMatrixAdjusted);
break;
```

7. Modify the call to `getOrientation()` to use the new adjusted rotation matrix instead of the original matrix.

```
SensorManager.getOrientation(rotationMatrixAdjusted,
    orientationValues);
```

8. Build and run the app again. The colors of the spots should now change on the correct edges of the device, regardless of how the device is rotated.

## Solution code

Android Studio project: [TiltSpot](#)

## Coding challenge

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

**Challenge:** A general rule is to avoid doing a lot of work in the `onSensorChanged()` method, because the method runs on the main thread and may be called many times per second. In particular, the changes to the colors of the spot can look jerky if you're trying to do too much work in `onSensorChanged()`. Rewrite `onSensorChanged()` to use an `AsyncTask` object for all the calculations and updates to views.

## Summary

- Motion sensors such as the accelerometer measure device movement such as tilt, shake, rotation, or swing.
- Position sensors such as the geomagnetic field sensor (magnetometer) can determine the device's position relative to the Earth.
- The accelerometer measures device acceleration, that is, how much the device is accelerating and in which direction. Acceleration forces on the device include the force of gravity.
- The magnetometer measures the strength of magnetic fields around the device. This includes Earth's magnetic field, although other fields nearby may affect sensor readings.
- You can use combined data from motion and position sensors to determine the device's orientation (its position in space) more accurately than with individual sensors.
- The 3-axis device-coordinate system that most sensors use is relative to the device itself in its default orientation. The *y*-axis is vertical and points toward the top edge of the device, the *x*-axis is horizontal and points to the right edge of the device, and the *z*-axis extends up from the surface of the screen.
- The Earth's coordinate system is relative to the surface of the Earth, with the *y*-axis pointing to magnetic north, the *x*-axis 90 degrees from *y* and pointing east, and the *z*-axis extending up into space.
- Orientation angles describe how far the device is oriented or tilted with respect to the Earth's coordinate system. There are three components to orientation:
  - *Azimuth* : The direction (north/south/east/west) the device is pointing. 0 is magnetic north.
  - *Pitch* : The top-to-bottom tilt of the device. 0 is flat.
  - *Roll* : The left-to-right tilt of the device. 0 is flat.
- To determine the orientation of the device:
- Use the `SensorManager.getRotationMatrix()` method. The method combines data from

the accelerometer and magnetometer and translates the data into the Earth's coordinate system.

- Use the `SensorManager.getOrientation()` method with a rotation matrix to get the orientation angles of the device.
- The alpha value determines the opacity of a drawable or view. Lower alpha values indicate more transparency. Use the `setAlpha()` method to programmatically change the alpha value for a view.
- When Android automatically rotates the activity in response to device orientation, the activity coordinate system also rotates. However, the device-coordinate system that the sensors use remains fixed.
- The default device-coordinate system sensors use is also based on the natural orientation of the device, which may not be "portrait" or "landscape."
- Query the current device orientation with the `Display.getRotation()` method.
- Use the current device orientation to remap the coordinate system in the right orientation with the `SensorManager.remapCoordinateSystem()` method.

## Related concept

The related concept documentation is in [Motion and position sensors](#).

## Learn more

Android developer documentation:

- [Sensors Overview](#)
- [Motion Sensors](#)
- [Position Sensors](#)

Android API reference documentation:

- `Sensor`
- `SensorEvent`
- `SensorManager`
- `SensorEventListener`
- `Surface`
- `Display`

Other:

- [Accelerometer Basics](#)
- [Sensor fusion and motion prediction](#) (written for VR, but many of the basic concepts

apply to basic apps as well)

- [Android phone orientation overview](#)
- [One Screen Turn Deserves Another](#)

# 4.1A: Using the Profile GPU Rendering tool

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Run the Profile GPU Rendering tool](#)
- [Task 2. Run the Profile GPU Rendering tool on an app with performance issues](#)
- [Solution code](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

Every app is different and could have different performance issues. While there are best practices, you need to analyze the unique configuration of your app to find and fix performance issues.

To discover what causes your app's specific performance problems, use tools to collect data about your app's execution behavior. Understand and analyze what you see, then improve your code. Android Studio and your device provide profiling tools to record and visualize the rendering, compute, memory, and battery performance of your app.

In this practical, you use the [Profile GPU Rendering tool](#) on your device to visualize how long it takes an app to draw frames to the screen.

**Important:** To run the Profile GPU Rendering tool and complete this practical exercise, you need a physical or virtual device running at least Android 4.1 (API level 16) with [Developer Options](#) turned on.

Your app must consistently draw the screen fast enough for your users to see smooth, fluid motions, transitions, and responses. To this end, typically, modern devices strive to display 60 frames per second (a frame rate of 60 FPS), making 16 milliseconds available to each frame. Therefore, your app has to do its work in **16 milliseconds** or less for every screen refresh.

**Important:** For real-world performance tuning, run your app and the tools on a physical device, not an emulator, as the data you get from running on an emulator may not be accurate. Always test on the lowest-end device that your app is targeting. If you do not have a physical device, you can use an emulator to get an idea of how your app is performing, even though some performance data from the emulator may not be accurate.

# What you should already KNOW

- You should be able to download an app from GitHub, open it with Android Studio, and run it.
- It helps to be familiar with the following concepts and terminology:
- **CPU**: A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control, and input/output (I/O) operations specified by the instructions.
- **GPU**: A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and display images.
- **Frame rate**: Frame rate (expressed in frames per second, or FPS) is the frequency (rate) at which consecutive images, called *frames*, are displayed in an animated display. Mobile devices typically display 60 frames per second, which appears as smooth motion to the human eye.
- **Display list**: A display list (or display file) is a series of graphics commands that define an output image as a series of primitives (such as lines and basic shapes).
- **Rendering pipeline**: A conceptual model that describes the steps that a graphics system performs when it renders graphical information on the screen. At the highest level, the pipeline creates a display list from program instructions in the CPU, passes the display list to the GPU, and the GPU executes the display list to draw the screen.

As a refresher, watch the [Invalidation, Layouts, and Performance](#) video for an overview of the Android rendering pipeline. Read the [Rendering and Layout](#) concept chapter as well as [How Android Draws Views](#).

# What you will LEARN

You will learn how to:

- Use the Profile GPU Rendering tool to visualize Android drawing the screen.

# What you will DO

- Run the Profile GPU Rendering tool on the RecyclerView app and examine the results.
- Create a simple app that has performance problems and run the Profile GPU Rendering tool on it.

# App overview

- For the first part of this practical, use the [RecyclerView app](#).
- For the second part of this practical, you build an app that loads images, [LargeImages](#), to see how image size affects performance. In a later practical, you learn more about using images correctly.

The following screenshot shows one way of implementing the LargeImages app.



2:15 PM

## Largelimages



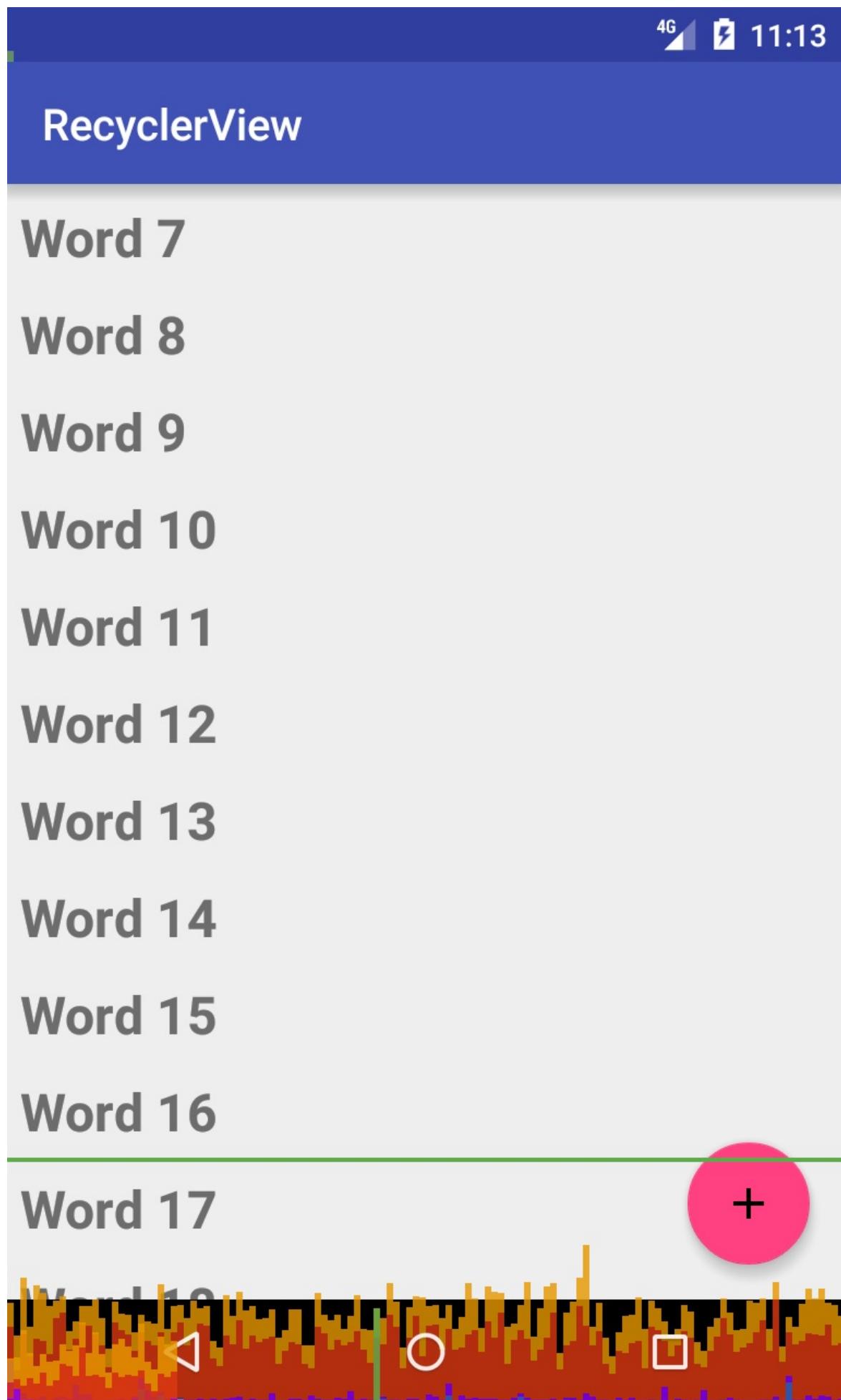
## Task 1. Run the Profile GPU Rendering tool

Putting pixels on the screen involves four primary pieces of hardware:

- The CPU computes [display lists](#). Each display list is a series of graphics commands that define an output image.
- The GPU (graphics processing unit) renders images to the display.
- The memory stores your app's images and data, among other things.
- The battery provides electrical power for all the hardware on the device.

Each of these pieces of hardware has constraints. Stressing or exceeding those constraints can cause your app to be slow, have bad display performance, or exhaust the battery.

The [Profile GPU Rendering](#) tool gives you a quick visual representation of how much time it takes to render the frames of a UI window relative to the 16-ms-per-frame benchmark. The tool shows a colored bar for each frame. Each bar has colored segments indicating the relative time that each stage of the rendering pipeline takes, as shown in the screenshot below.



What it's good for:

- Quickly seeing how screens perform against the 16-ms-per-frame target.
- Identifying whether the time taken by any part of the rendering pipeline stands out.
- Looking for spikes in frame rendering time associated with user or program actions.

## 1.1 Disable Instant Run

**Note:** While profiling an app, [disable Instant Run inAndroidStudio](#). There is a small performance impact when [using Instant Run](#). This performance impact could interfere with information provided by performance profiling tools. Additionally, the stub methods generated while using Instant Run can complicate stack traces. See [About Instant Run](#) for details.

In Android Studio, turn off Instant Run for the physical and virtual devices you want to profile.

1. Open the **Settings or Preferences** dialog: On Windows or Linux, select **File > Settings** from the menu bar. On Mac OSX, select **Android Studio > Preferences** from the menu bar.
2. Navigate to **Build, Execution, Deployment > Instant Run**.
3. Deselect the **Restart activity on code changes** checkbox.

## 1.2 Install and run the RecyclerView app

For this tool walkthrough, use the RecyclerView app from the [Android Developer Fundamentals](#) course.

1. Download the [RecyclerView app](#).
2. Open the app in Android Studio and run it. You may need to uninstall previous versions of the app.
3. Interact with the app. Does the app scroll smoothly?
4. Press the + button and add words to the list. Does this change how you perceive the performance of the app?

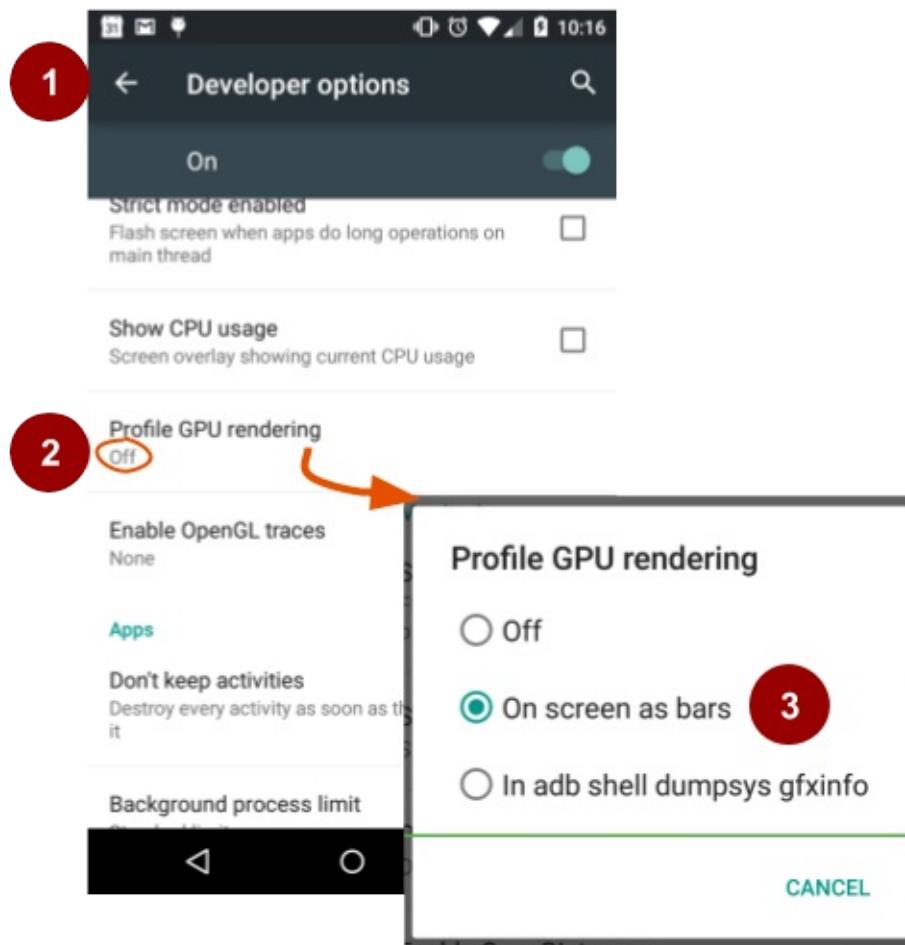
RecyclerView is an efficient way of displaying information, and on most devices and recent versions of Android, the app performs smoothly. What users perceive and how they experience the app are your ultimate benchmarks for performance.

## 1.3 Turn on the Profile GPU Rendering tool

Ensure that [developer options](#) is turned on, and allow USB Debugging if prompted.

Go to **Settings > Developer options** and follow the steps as illustrated in the screenshot below.

1. Scroll down to the **Monitoring** section.
2. Select **Profile GPU rendering**.
3. Choose **On screen as bars** in the dialog box. Immediately, you see colored bars on your screen. The bars vary depending on your device and version of Android, so they may not look exactly as illustrated.



4. Switch to the RecyclerView app and interact with it.

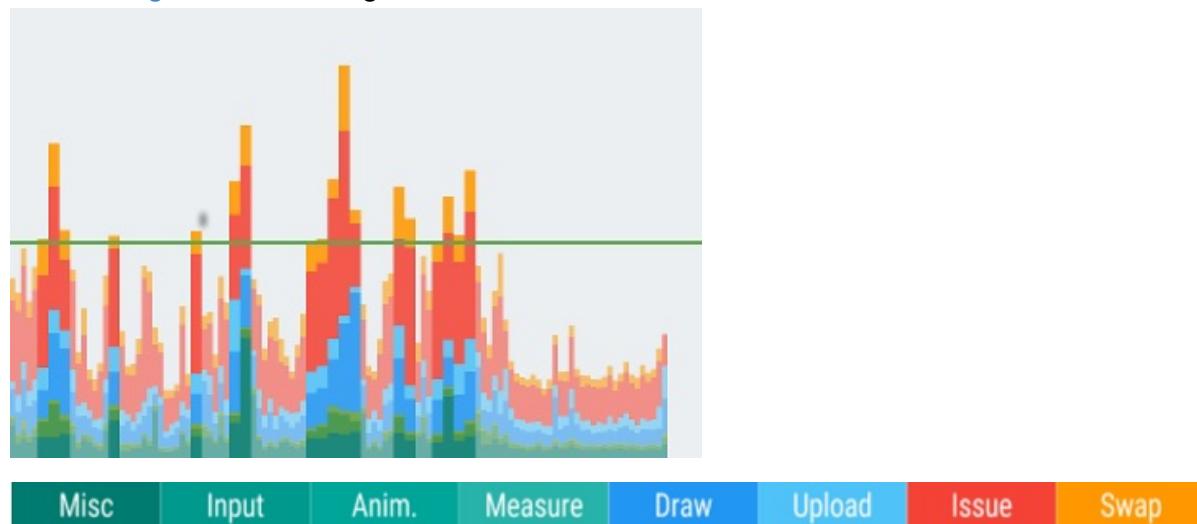
Note the following:

- Each bar represents one frame rendered.
- If a bar goes above the green line, the frame took more than 16ms to render. Ideally, most of the bars stay below the green line most of the time.

This is not an absolute requirement. For example, when you first load an image, the bar may go above the green line, but users may not notice a problem, because they may expect to wait a moment for an image to load. Having some tall bars does not mean your app has a performance problem. However, if your app does have a performance problem, tall bars can give you a hint as to where to start looking.

- The colors in each bar represent the different stages in rendering the frame. The colored sections visualize the stages and their relative times. The table below gives an explanation.

The image below shows the bars and color legend for a device that is running Android 6.0 or higher. The bars for older versions use different coloring. See the [Profile GPU Rendering Walkthrough](#) for the bar legend for older versions.



The following table shows the component bars in Android 6.0 and higher.

Color	Rendering stage	Description
Orange	Swap buffers	Represents the time the CPU is waiting for the GPU to finish its work. If this part of the bar is tall, the app is doing too much work on the GPU.
Red	Command issue	Represents the time spent by Android's 2-D renderer as it issues commands to OpenGL to draw and redraw display lists. The height of this part of the bar is directly proportional to the sum of the time it takes for all the display lists to execute—more display lists equals a taller red segment of the bar.
Cyan	Sync and upload	Represents the time it takes to upload bitmap information to the GPU. If this part of the bar is tall, the app is taking considerable time loading large amounts of graphics.
Blue	Draw	Represents the time used to create and update the view's display lists. If this part of the bar is tall, there may be a lot of custom view drawing, or a lot of work in <code>onDraw</code> methods.
Teal	Measure and layout	Represents the amount of time spent on <code>onLayout</code> and <code>onMeasure</code> callbacks in the view hierarchy. A large segment indicates that the view hierarchy is taking a long time to process.
Light Green	Animation	Represents the time spent evaluating animators. If this part of the bar is tall, your app could be using a custom animator that's not performing well, or unintended work is happening as a result of properties being updated.
Dark Green	Input handling	Represents the time that the app spent executing code inside of an input event callback. If this part of the bar is tall, the app is spending too much time processing user input. Consider offloading such processing to a different thread.
Dark Teal	Misc. time / Vsync delay	Represents the time that the app spends executing operations between consecutive frames. It might indicate too much processing happening in the UI thread that could be offloaded to a different thread. (Vsync is a display option found in some 3-D apps.)

For example, if the green **Input handling** portion of the bar is tall, your app spends a lot of time handling input events, executing code called as a result of input event callbacks. To improve this, consider when and how your app requests user input, and whether you can handle it more efficiently.

Note that `RecyclerView` scrolling can show up in the **Input handling** portion of the bar.

`RecyclerView` scrolls immediately when it processes the touch event. For this reason, processing the touch event needs to be as fast as possible.

If you spend time using the Profile GPU Rendering tool on your app, it will help you identify which parts of the UI interaction are slower than expected, and then you can take action to improve the speed of your app's UI.

**Important:** As the Android system and tools improve, recommendations may change. Your best and most up-to-date source of information on performance is the Android developer documentation.

## Task 2. Run the Profile GPU Rendering tool on an app with performance issues

**Important:** We do not want to teach you how to write bad apps. We don't even want to show you the code. However, we can simulate a slow app by using images that are too large; in a different chapter, you learn how to improve performance by using the right image sizes and types for your app.

### 2.1 Create an app with image backgrounds

Build a LargeImages app that has performance problems. (You re-use this app in a later practical.)



2:15 PM

## Largelimages



One way to implement the LargeImages app is to show simple instructions in a `TextView` and the large background image that is provided with the solution code.

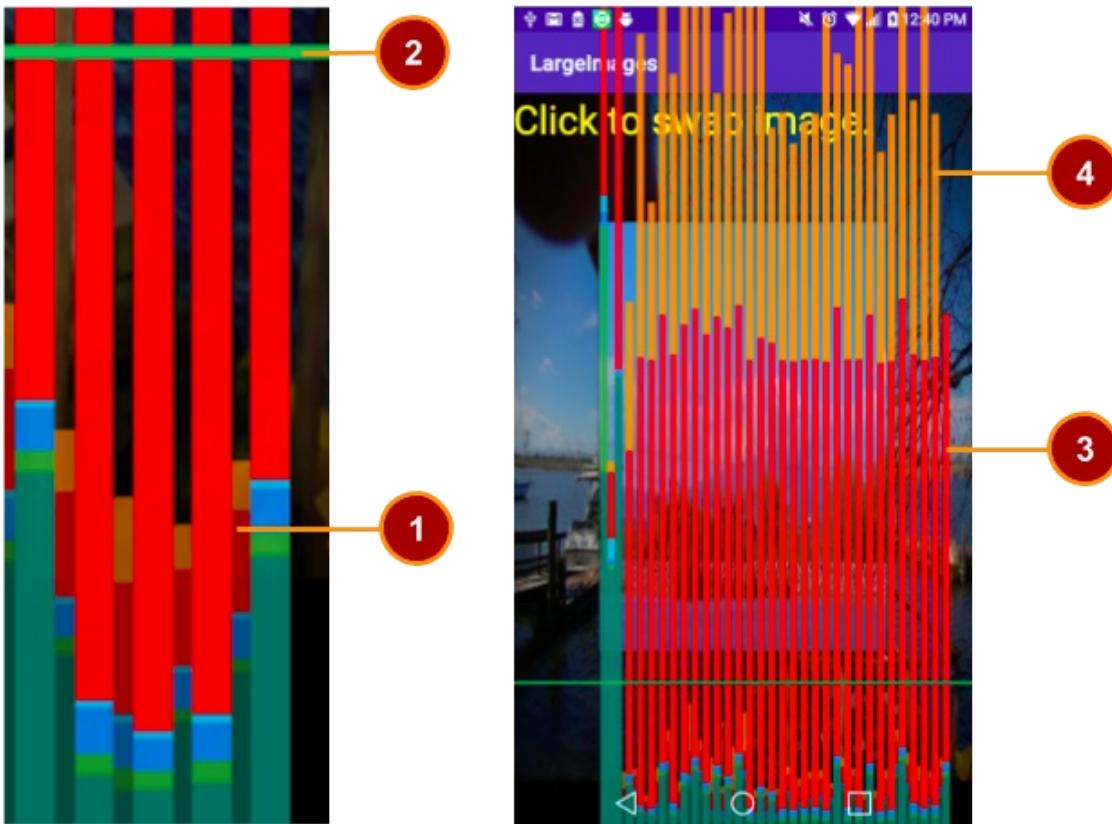
Do the following:

1. Create an app named LargeImages using the **Empty Template**.
2. Choose 5.1, Lollipop (API 22) as the minimum SDK. (Lossless and transparent WebP images that you use later in this app are supported in Android 4.3, API level 18 and higher, and API 22 is recommended.)
3. Put at least one small and one large image into the drawables resource folder.
  - You can copy images from the solution code in [GitHub](#).
  - Small images should be around 12KB.
  - The largest image should be a few hundred KB. The `dinosaur_medium.jpg` supplied with the solution code is 495KB and a good starting point. The max size of the image you can load depends on the amount of device memory available to your app. You can look at the specifications for a phone model to find out how much RAM it has. Not all of this is available to a single app, so you may have to experiment a bit. You may need to try different size images to find the largest image that won't crash your app.
4. Add an `Activity`.
5. Create a layout with a `LinearLayout` with a `TextView`. Set the height and width of the `TextView` to `match_parent`.
6. Set the background (`android:background`) on the `TextView` to the small image.
7. Add this text to this to the `Textview`: "Click to swap image."
8. Add a click handler to the `TextView` that changes the background when the screen is tapped. Use the following code snippet as a template. The code uses a numeric toggle so that you can experiment with any number of images.

```
private int toggle = 0;  
...  
public void changeImage(View view){  
    if (toggle == 0) {  
        view.setBackgroundResource(R.drawable.dinosaur_medium);  
        toggle = 1;  
    } else {  
        view.setBackgroundResource(R.drawable.ankylo);  
        toggle = 0;  
    }  
}
```

9. Run the app with Profile GPU Rendering turned on and tap to swap pictures. When you load the small image, you should see relatively short bars. When you load the large

image, there should be relatively tall bars, similar to the ones shown below.



Profile GPU Rendering bars for the large image app are shown in the previous screenshots. The detail on the left shows the faint short bars for the small image (1) staying below the green line (2). The screenshot on the right show the bars as they appear on your device emphasizing the bars that cross the green line (3,4).

1. The narrow, faint bar for the small image is below the green line, so there is no performance issue.
2. The green horizontal line indicates the 16 millisecond rendering time. Ideally, most bars should be close to or below this line.
3. After the small image is replaced by the large image, the **red** bar segment is huge. This **Command issue** segment represents the time spent by Android's 2D renderer issuing commands to draw and redraw display lists.
4. The large image also has a huge **orange** segment. This **Swap buffers** segment represents the time the CPU is waiting for the GPU to finish its work.

The GPU works in parallel with the CPU. The Android system issues draw commands to the GPU, and then moves on to its next task. The GPU reads those draw commands from a queue. When the CPU issues commands faster than the GPU consumes them, the queue between the processors becomes full, and the CPU blocks. The CPU waits until there is space in the queue to place the next command.

To mitigate this problem, reduce the complexity of work occurring on the GPU, similar to what you would do for the "Command issue" phase.

See [Analyzing with Profile GPU Rendering](#) for details on each stage.

To fix the app, you would use a different or smaller image. Finding problems can be as straightforward as this, or it can take additional analysis to pinpoint what is happening in your app's code.

## Troubleshooting

- If your app crashes with the images provided for the sample app, you may need to use smaller images for your device.
- If your app crashes on the emulator, edit the emulator configuration in the AVD Manager. Use your computer's graphics card instead of simulating the device's graphic software by selecting "Graphics: Hardware - GLES" on the configuration screen.
- If you get a `failed to find style 'textviewstyle' in current theme` or other rendering error in the **Design** view in Android Studio, click the refresh link provided with the error message.

## 2.2 Simulate long input processing

To demonstrate how doing too much work on the UI thread slows down your app, slow down drawing by putting the app to sleep:

1. Add code to the click handler of the LargeImages app to let your app sleep for two screen refreshes before switching the background to the smaller image. This means that instead of refreshing the screen every 16 ms, your app now refreshes every 48 ms with new content. This will be reflected in the bars displayed by the Profile GPU Rendering tool.

```
public void changeImage(View view){  
    if (toggle == 0) {  
        view.setBackgroundResource(R.drawable.dinosaur_large);  
        toggle = 1;  
    } else {  
        try {  
            Thread.sleep(32); // two refreshes  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        view.setBackgroundResource(R.drawable.ankylo);  
        toggle = 0;  
    }  
}
```

- Run your app. The previously short bars for the small image should now be much taller. Wait a while before clicking the `TextView` to get the next image; otherwise, the Android system may display an ANR.

If your app is only swapping backgrounds, the impact from this change may not be significant for the user. However, if you are running animated content, skipping two out of three frames will result in a stuttering and jagged animation.

While Profile GPU Rendering cannot tell you exactly what to fix in your code, it can hint at where you should start looking. And let's admit it, the on-screen bars are pretty cool!

## Solution code

Android Studio project: [LargeImages](#).

## Summary

- You can use the [Profile GPU Rendering](#) tool on your device to visualize the relative times it takes for your app to render frames on screen.
- Because it can be turned on quickly and its results are immediate and graphical, this tool is a good first step when analyzing potential app performance issues related to rendering.
- The sections of the colored bars can indicate where in your app you might look for performance problems.
- Common reasons for slow drawing are an app taking too long to process user input on the UI thread, and backgrounds and images that are unnecessary or too large.

## Related concepts

The related concept documentation is in [Rendering and layout](#).

## Learn more

- [Profile GPU Rendering Walkthrough](#)
- [Analyzing with Profile GPU Rendering](#)
- [How Android Draws Views](#)
- Video: [Invalidation, Layouts, and Performance](#)



# 4.1B: Using the Debug GPU Overdraw and Layout Inspector tools

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Visualize overdraw with Debug GPU Overdraw](#)
- [Task 2. Inspect the view hierarchy](#)
- [Solution code](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

You can make your apps interesting and visually compelling by following [Material Design](#) guidelines, creating nested hierarchies of layouts, and using drawables as background elements for your views. However, with increasing complexity come performance challenges, and layouts draw faster and use less power and battery if you spend time designing them in the most efficient way. For example:

- Overlapping views result in *overdraw*, where the app wastes time drawing the same pixel multiple times, and only the final rendition is visible to the user. Size and organize your views so that every pixel is only drawn once or twice.
- Deeply nested layouts force the Android system to perform more passes to lay out all the views than if your view hierarchy is flat.
- To simplify your view hierarchy, combine, flatten, or eliminate views.

## What you should already KNOW

You should be able to:

- Create apps with Android Studio and run them on a mobile device.
- Work with the Developer options settings on a mobile device.

## What you will LEARN

You will learn how to:

- Run the Debug GPU Overdraw tool to visualize Android drawing to the screen.
- Use the Layout Inspector tool to inspect the view hierarchy of an app.

## What you will DO

- Create a testing app, StackedViews, to illustrate overdraw.
- Run the Debug GPU Overdraw tool on the StackedViews app.
- Fix the StackedViews app to reduce overdraw.
- Use the Layout Inspector tool to identify how you could simplify the view hierarchy.

## App overview

The StackedViews app arranges three overlapping `TextView` objects inside a `ConstraintLayout` to demonstrate overdraw. While this is a contrived and simplistic example, overlapping views can easily happen during development as you add more features and complexity to your app. The screenshot below shows an arrangement of three overlapping views. **View 1** fills the screen. **View 2** fills the bottom two thirds of the screen. **View 3** fills the bottom third of the screen to create progressively more overlap.

## StackedViews

View 1

View 2

View 3

# Task 1. Visualize overdraw with Debug GPU Overdraw

One possible cause for slow rendering performance is overdraw, which is when your app draws over the same area multiple times, but the user sees only what has been drawn last. This can result in performance problems, especially for older or less powerful devices.

Learn more about overdraw in the [Rendering and layout](#) concept chapter.

## 1.1 Create an app to illustrate overdraw

Create an app called StackedViews that stacks views on top of each other.

1. Create an app using the **Empty template**, which uses `ConstraintLayout`.
2. Set a light gray background on the `ConstraintLayout`.
3. Add a `TextView` element where the width and height match the parent. Set a light gray background, such as `#fafafa`.
4. Add a second, overlapping, `TextView` element. Make the width match the parent. Make the height about three quarters of the height from the bottom. You can do this by making the height match the parent, then use `layout_marginTop` to add space at the top. Use a darker gray background, such as `#e0e0e0`.
5. In the same way, add a third, overlapping `TextView` element that fills the width of the parent and about half of the height from the bottom. Use an even darker gray background, such as `#bdbdbd`.
6. Optionally, use a large font such as `@style/TextAppearance.AppCompat.Display1` for the `TextViews`.
7. Run the app on your device. It should look similar to the screenshot below, with three progressively more overlapping views.

## StackedViews

View 1

View 2

View 3

Your final code should look similar to this:

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.android.stackedsviews.MainActivity"
    android:background="@android:color/background_light">

    <TextView
        android:id="@+id/view1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/very_light_gray"
        android:text="@string/view1"
        android:textAppearance="@style/TextAppearance.AppCompat.Display1"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/view2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="156dp"
        android:background="@color/light_gray"
        android:text="@string/view2"
        android:textAppearance="@style/TextAppearance.AppCompat.Display1"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/view3"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="300dp"
        android:background="@color/medium_gray"
        android:text="@string/view3"
        android:textAppearance="@style/TextAppearance.AppCompat.Display1"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

## 1.2 Turn on Debug GPU Overdraw on your device

Make sure [developer options](#) is turned on. Turn on USB Debugging, and allow it when prompted.

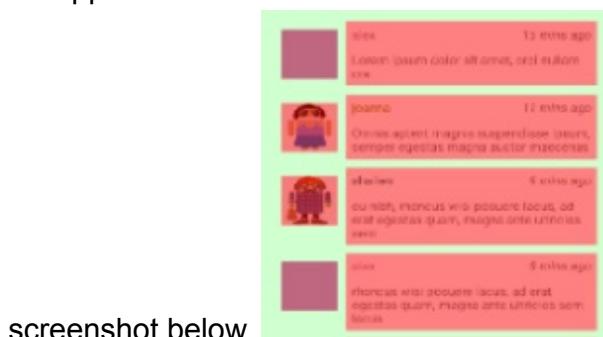
Turn on Debug GPU Overdraw on your mobile device:

1. Go to **Settings > Developer options**.
2. In the **Hardware accelerated rendering** section, select **Debug GPU Overdraw**.
3. In the **Debug GPU overdraw** dialog, select **Show overdraw areas**.
4. Watch your device turn into a rainbow of colors. The colors hint at the amount of overdraw on your screen for each pixel.

The following key shows which colors indicate how much overdraw.

True color has no overdraw.	
1X Overdraw	Purple/blue is overdrawn once.
2X Overdraw	Green is overdrawn twice.
3X Overdraw	Pink is overdrawn three times.
4X Overdraw	Red is overdrawn four or more times.

An app with too much overdraw will have a lot of pink and red coloring, as shown in the



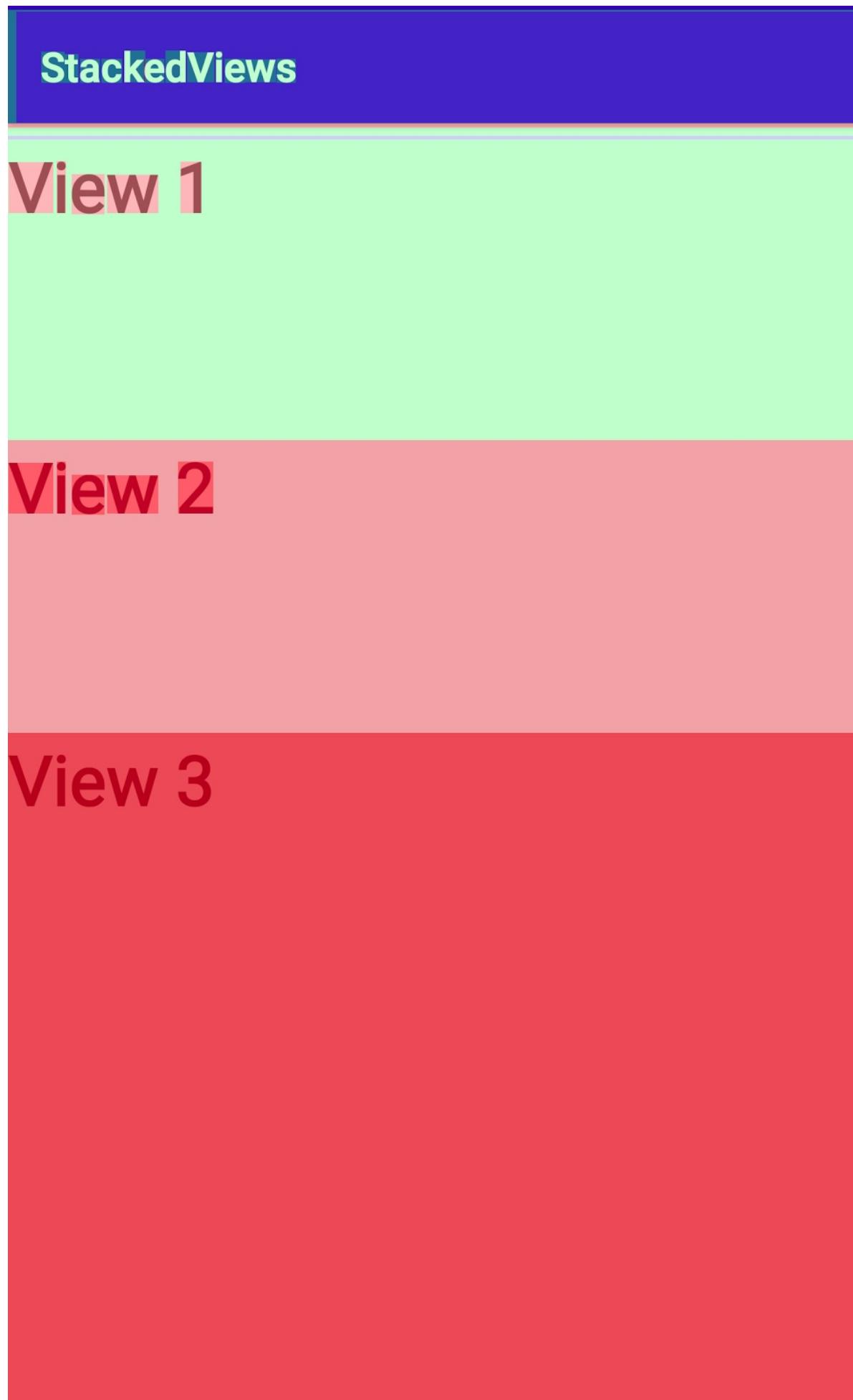
screenshot below.

A properly laid out app has little overdraw and primarily show true and purple coloring, as shown in the screenshot below. Some overdraw is unavoidable, for example, when drawing



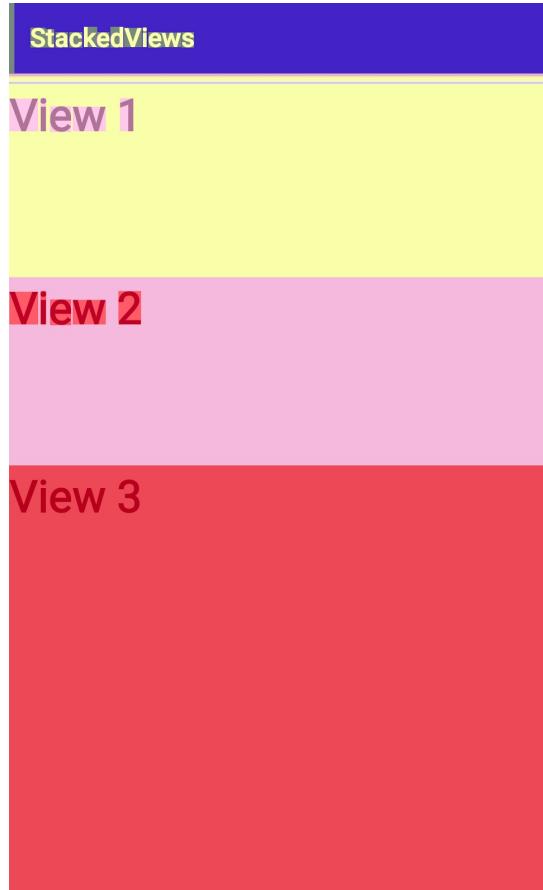
text onto a background.

1. Run the StackedViews app. The screen should look similar to the one below, with green, pink, and red colors that indicate significant areas of overdraw where the `TextView` objects overlap.



**Important:** If it's hard for you to see the difference between the colors that the tool shows, try adjusting the tool for deuteranomaly:

1. Go to **Settings > Developer options**.
2. In the **Hardware accelerated rendering** section, select **Debug GPU Overdraw**.
3. In the **Debug GPU overdraw** dialog, select **Show areas for Deuteranomaly** to show



Compare the colors of the views in this app with the color key shown above, and you see that **View 3** and **View 2** are colored for 3x and 4x overdraw. This is because as laid out in the XML file, you stacked the views on top of each other. A `ConstraintLayout` with a background is the bottom layer, and View 1 is drawn on top of that, so **View 1** is overdrawn twice, and so on.

In a real-world app, you would not use a layout like this, but remember that this is a simple example to demonstrate the tool. Common, realistic examples that produce overdraw include:

- Lists with items that contain thumbnails, text, and controls.
- Custom views, such as custom menus or drawers.
- Views that overlap with other views for any reason.

## 1.3 Fix overdraw

Fixing overdraw can be as basic as removing invisible views, and as complex as re-architecting your view hierarchy. To fix overdraw, you usually need to take one or more of the following actions.

- Eliminate unnecessary backgrounds, for example, when a `View` displays an image that fills it, the background is completely covered, so it can be removed.
- Remove invisible views. That is, remove views that are completely covered by other views.
- In custom views, clip generously. (See [Rendering and Layout](#). You learn about custom views in another chapter.)
- Reduce the use of transparency. For example, instead of using transparency to create a color, find a color. Instead of using transparency to create a shine-through effect for two overlapping images, preprocess the two images into one.
- Flatten and reorganize the view hierarchy to reduce the number of views. See the [Rendering and Layout](#) concept chapter for examples and best practices.
- Resize or rearrange views so they do not overlap.

To reduce overdraw for the `StackedViews` app, rearrange the views in the app so that they do not overlap. In the XML code:

1. Remove the background of the `ConstraintLayout`, because it's invisible.
2. Set the `layout_width` and `layout_height` of each view to `0dp`.
3. Remove `layout_marginTop` for `view2` and `view3`, or set it to `0dp`.
4. Use constraints to lay out the views relative to each other.

The following code shows the constraints for each view:

```
<TextView
    android:id="@+id/view1"
    ...
    app:layout_constraintBottom_toTopOf="@+id/view2"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/view2"
    ...
    app:layout_constraintBottom_toTopOf="@+id/view3"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/view1" />

<TextView
    android:id="@+id/view3"
    ...
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/view2" />
```

Your final layout should produce output from Debug GPU Overdraw similar to one of the images below. (The second image shows overdraw for deuteranomaly.) You draw once for the background, and on top of that, the text.

## StackedViews

View 1

View 2

View 3



## StackedViews

View 1

View 2

View 3

In a realistic setting without custom views, overdraw may not be a huge performance problem for your app since the Android framework optimizes drawing the screen. However, since overdraw is usually straightforward to find and fix, you should always minimize it so it does not pollute your other performance data.

## Task 2. Inspect the view hierarchy

The process of rendering views to the screen includes a measure-and-layout stage, during which the system appropriately positions the relevant items in your view hierarchy. The "measure" part of this stage determines the sizes and boundaries of `View` objects. The "layout" part determines where on the screen to position the `View` objects.

The following recommendations cover the most common cases:

### **Remove views that do not contribute to the final image.**

Eliminate from your code the views that are completely covered, never displayed, or outside the screen. This seems obvious, but during development, views that later become unnecessary can accumulate.

### **Flatten the view hierarchy to reduce nesting**

Android [Layouts](#) allow you to nest UI objects in the view hierarchy. This nesting can impose a cost. When your app processes an object for layout, the app performs the same process on all children of the layout as well.

Keep your view hierarchy flat and efficient by using `ConstraintLayout` wherever possible.

### **Reduce the number of views**

If your UI has many simple views, you may be able to combine some of them without diminishing the user experience.

- Combining views may affect how you present information to the user and will include design trade-offs. Opt for simplicity wherever you can.
- Reduce the number of views by combining them into fewer views. For example, you may be able to combine `TextView` objects if you reduce the number of fonts and styles.

### **Simplify nested layouts that trigger multiple layout passes**

Some layout containers, such as `RelativeLayout`, require two layout passes in order to finalize the positions of their child views. As a result, their children also require two layout passes. When you nest these types of layout containers, the number of layout passes

increases exponentially with each level of the hierarchy. See the [Optimizing View Hierarchies](#) documentation and the [Double Layout Taxation](#) video.

Be conscious of layout passes when using:

- `RelativeLayout`
- `LinearLayout` that also use `measureWithLargestChild`
- `GridView` that also use `gravity`
- Custom view groups that are subclasses of the above
- Weights in `LinearLayout`, which can sometimes trigger multiple layout passes

Using any of the above view groups as the root of a complex view hierarchy, the parent of a deep subtree, or using many of them in your layout, can hurt performance. Consider whether you can achieve the same layout using a view group configuration that does not result in these exponential numbers of layout passes, such as a `ConstraintLayout`.

## 2.1 Inspect your app's layout with Layout Inspector

The [Layout Inspector](#) is a tool in Android Studio that allows you see your view hierarchy at runtime.

Design view shows you the static layout of an activity as defined in your XML files, but it does not include any views that you might create, destroy, or move at runtime.

Inspecting the view hierarchy can show you where you might simplify your view hierarchy. In addition, the Layout Inspector can also help you identify overlapping views. If Debug GPU Overdraw shows that your app has overdraw, you may need to run Layout Inspector to identify the exact views involved, or to help you determine which views you might be able to rearrange.

With Layout Inspector you take a snapshot of the view hierarchy of a running app and display it for inspection.

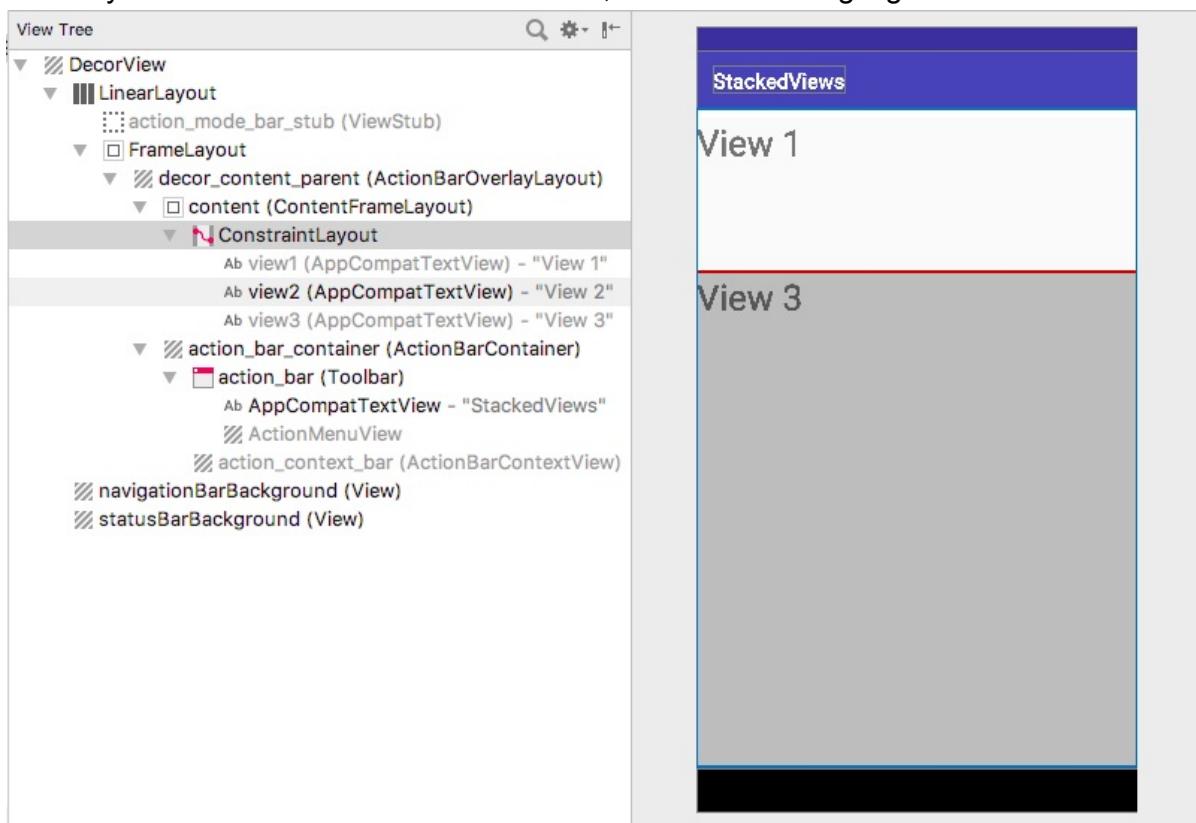
## 2.2 Find overlapping views

If your layout includes completely overlapping views, then by default, only the front-most view is clickable in the screenshot. To make the view behind clickable, right-click the front-most view in the **View Tree** panel and uncheck **Show in preview**, as follows:

1. In your code, change the size of **View 2** to completely cover **View 3**. In the layout for `@+id/view3`, set `android:layout_marginTop="156dp"`
2. Run the app.
3. Open the Layout Inspector. **Tools > Android > Layout Inspector**.

In the preview, you can only see outlines around **View 1** and **View 3**, which are on top.

4. In the **Tree View**, right-click **View 3** and deselect **Show in preview**.
5. In the **Tree View**, right-click **View 1** and deselect **Show in preview**.
6. Click `ConstraintLayout` in the **Tree View**. Now if you mouse over the preview, the only outline you see is around the hidden **View 2**, and **View 2** is highlighted in **Tree View**.



Layout Inspector does not connect to your running app. It creates a static capture based on your code.

You can find the captures Android Studio made in the captures folder of your Android Studio project. The files use the package name and the date, and have a `.li` extension (for example, `com.example.android.largimages_2017.4.12_12.01.li` ). In Project view, expand the captures folder and double-click any capture file to show its contents.

Inspecting the view hierarchy, the view arrangement and boundaries, and the properties of each view can help you determine how to optimize your app's view hierarchy.

## Solution code

Android Studio projects:

- [StackedViews\\_with\\_overdraw](#) – initial version of the StackedViews app with overdraw issues.

- [StackedViews\\_fixed](#) – final version of the StackedViews app after the overdraw issues have been fixed.

## Summary

- Use the Debug GPU Overdraw tool to identify overlapping views.
- Reduce overdraw by removing unused backgrounds and invisible views; use clipping to reduce overlapping of views.
- Use the Layout Inspector to investigate your view hierarchy.
- Simplify your view hierarchy by flattening the hierarchy, combining views, eliminating invisible views, or rearranging your layout.

## Related concept

The related concept documentation is in [Rendering and layout](#).

## Learn more

- [Debug GPU Overdraw Walkthrough](#)
- [Layout Inspector](#)
- [Reducing Overdraw](#)
- [Optimizing Layout Hierarchies](#)
- [Performance and View Hierarchies](#)
- [Optimizing Your UI](#)

# 4.1C: Using the Systrace and dumpsys tools

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Run the Systrace tool to analyze the WordListSQL app](#)
- [Task 2. Run the Systrace tool to analyze the LargeImages app](#)
- [Task 3. Run the dumpsys tool](#)
- [Coding challenge](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

[Systrace](#) (Android System Trace) captures and displays execution times of your app's processes and other Android system processes, which helps you analyze the performance of your app. The tool combines data from the Android kernel such as the CPU scheduler, disk activity, and app threads. The tool uses this data to generate an HTML report that shows an overall picture of an Android-powered device's system processes for a given period of time.

Systrace is particularly useful in diagnosing display problems when an app is slow to draw or stutters while displaying motion or animation.

[dumpsys](#) is an Android tool that runs on the device and dumps interesting information about the status of system services.

Both of these powerful tools let you take a detailed look at what is happening when your app runs. They produce a huge amount of detailed information about the system and apps. This information helps you create, debug, and improve the performance of your apps.

This practical will help you get started, but fully analyzing Systrace and `dumpsys` output takes a lot of experience and is beyond the scope of this practical.

## What you should already KNOW

You should be able to:

- Create apps with Android Studio and run them on a mobile device.
- Work with Developer Options on a mobile device.

## What you will LEARN

You will learn how to:

- Use Systrace to collect data about your app and view its reports.
- Run `dumpsy` to get a snapshot of information about app performance.

## What you will DO

- Use Systrace to capture information about the WordListSQL app on your device.
- View and navigate the trace output for WordListSQL.
- Find the **Frames** section in the Systrace output and get more information about specific frames.
- Examine the alerts, which show potential performance issues with your code.
- Run a trace of the LargeImages app and identify a problem with the app. Correlate what you find with the earlier Profile GPU Rendering tool output.
- Run the `dumpsy` tool to get additional information about frames.

## App overview

You will not build a new app in this practical. Instead you will use apps from other practicals.

- You need the WordListSQL app. Use the version of the WordListSQL app that you built in the Android Fundamentals Course, or download the app from the [WordListSql\\_finished folder](#) from GitHub.
- You also need the LargeImages app from a previous practical, or you can download the course version of the app from [GitHub](#).

## Task 1. Run the Systrace tool to analyze the WordListSQL app

Systrace (Android System Trace) helps you analyze how the execution of your app fits into the many running systems on an Android device. It puts together system and application thread execution on a common timeline.

To analyze your app with Systrace, you first collect a trace log of your app and the system activity. The generated trace allows you to view highly detailed, interactive reports showing everything happening in the system for the traced duration.

You can run the Systrace tool from one of the Android SDK's graphical user interface tools, or from the command line. The following sections describe how to run the tool using either of these methods.

## 1.1 Prerequisites

To run Systrace, you need:

- Android SDK Tools 20.0.0 or higher, which you already have if you've followed this course.
- A mobile device running at least Android 4.1 with USB Debugging enabled in [Developer Options](#). You already have this, if you've followed this course.
  - You can run on an emulator for practice, but your trace data may not be accurate.
  - Some devices do not have Systrace enabled on the kernel they are running, and you will get an error while generating the trace. In this case, you will need to use an emulator. See this [Stack Overflow post](#) if you want to explore deeper.
- [Python](#) language installed and included in your development computer's execution path. Check your Python installation from a command-line window:
  - Mac and Linux: `python -v`
  - Windows: `python`

If you do not have Python installed, follow the instructions for your platform at [python.org](#).

You can also find instructions in this [Check for Python](#) documentation.

## 1.2a Run Systrace from Android Device Monitor

The following instructions are for Android 4.2 and higher. If you are running on an older version of Android, follow the instructions on the [Analyze UI Performance with Systrace](#) page.

1. If you don't already have the WordListSQL app, download the [WordListSQL](#) app from the `wordListSql_finished` folder on GitHub. Open the app in Android Studio and run it on your device.
2. From Android Studio, choose **Tools > Android > Android Device Monitor**. If you have instant run enabled, you may be asked to **Disable ADB Integration**. Click **Yes**.

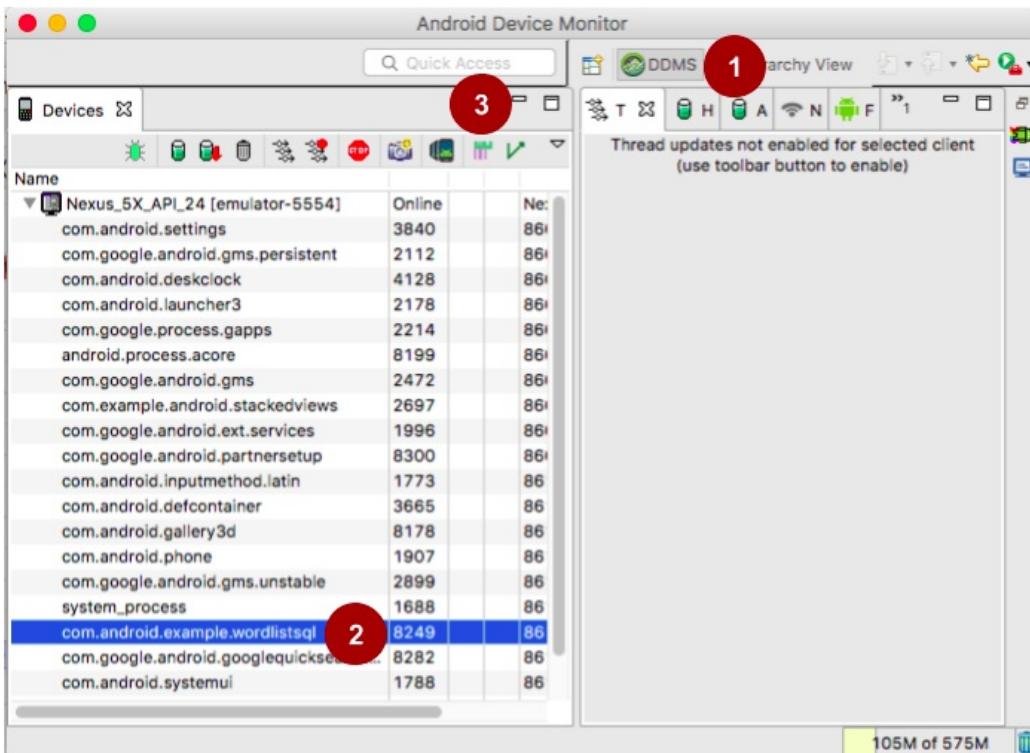
**Android Device Monitor** launches in a separate window.

**Important:** If you run your app on an emulator, you must restart `adb` as `root` in order to get a trace. Type `adb root` at the command line.

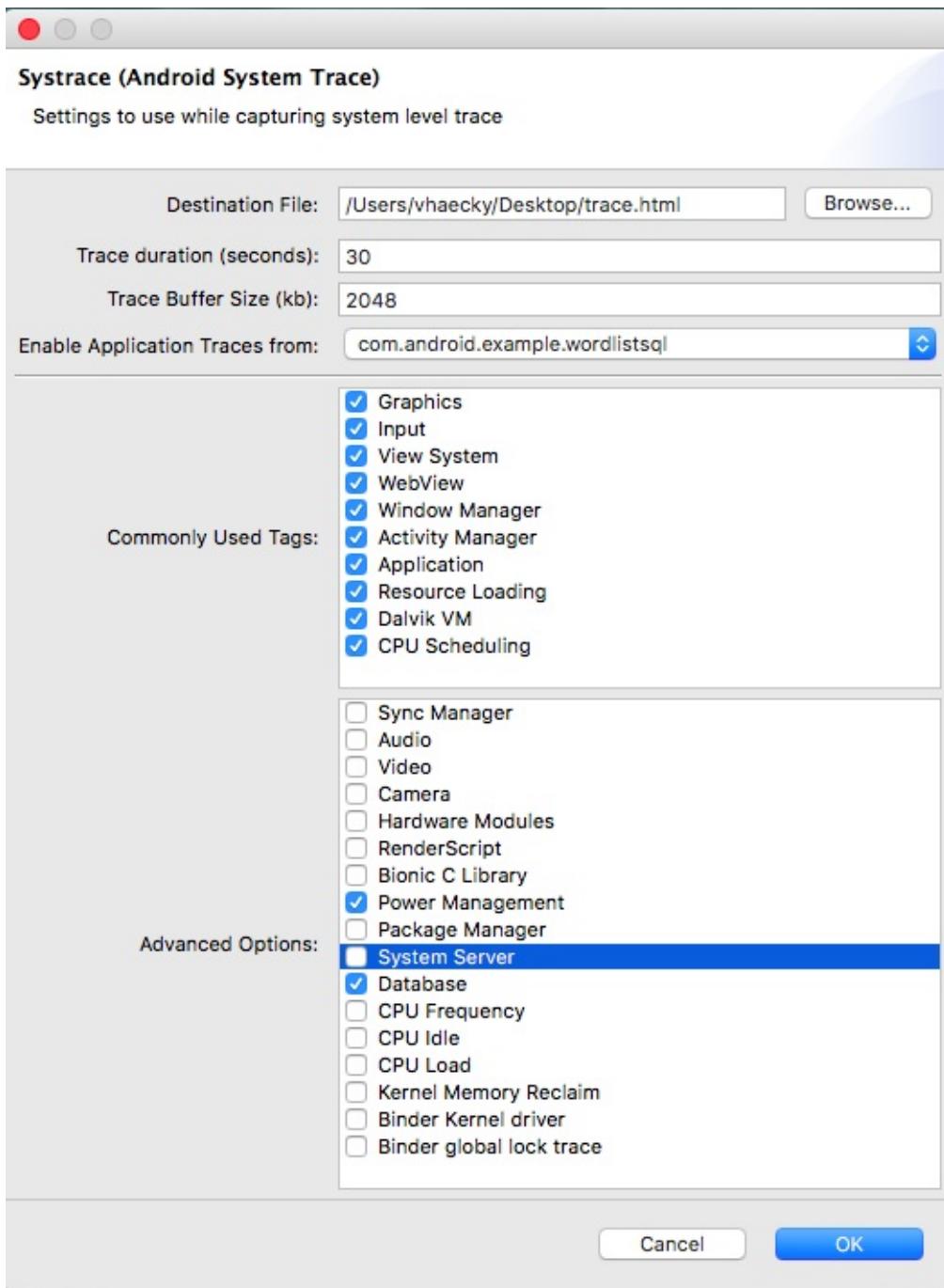
Follow the steps as illustrated in the screenshot below:



1. Click **DDMS** if it is not already selected. (1) in the screenshot below.
2. Select your app by package name in the **Devices** tab. (2) in the screenshot below.
3. Click the **Systrace** button to initiate the trace. (3) in the screenshot below.



4. In the **Systrace (Android System Trace)** pop-up, choose your settings for the trace.
  - **Destination File:** Where the trace is stored as an HTML file. Default is in your home directory with the filename `trace.html`.
  - **Trace duration:** Default is 5 seconds, and 15-30 seconds is a good time to choose.
  - **Trace Buffer Size:** Start with the default.
  - **Enable Application Traces from:** Make sure your app is visible and selected.
  - Now select tags to enable. Choose all the **Commonly Used Tags**. In the **Advanced Options**, choose **Power Management** and **Database**. Use fewer tags to limit the size and complexity of the trace output.



5. Click **OK** to start tracing. A pop-up appears, indicating that tracing is active.
6. Interact with the WordListSQL app on your device. Create, edit, and delete an item.
7. When time is up, the pop-up closes, and your trace is done.

## 1.2b [Optional] Run Systrace from the command line

If you prefer, you can run Systrace from the command line of any terminal window in Android Studio or on your desktop. The following steps use a terminal in Android Studio.

The following instructions are for Android 4.3 and higher. If you are running on an older version of Android, follow the instructions in the [systrace user guide](#).

To run Systrace from the command line:

1. In Android Studio, open the **Terminal** pane, for example, by selecting **View > Tool Windows > Terminal**.
2. In the terminal, change directory to where Systrace is located in the Android SDK: `cd < android-sdk >/platform-tools/systrace`

If you need to find the directory where the Android SDK is installed: In Android Studio, choose **Android Studio > Preferences** and search for "sdk". Alternatively, open **Tools > Android > SDK Manager**. The top of the pane has a text field that shows the Android SDK location.

3. Start Systrace: `python systrace.py --time=10`
  - This runs systrace for 10 seconds, with default category tags, and saves the output HTML file to the `systrace` directory, which is the current directory.
  - Systrace is written in the Python language.
  - The `time` argument is the time to run, in seconds.
  - See the [Systrace](#) documentation for versions and options.
  - If you get an error about `serial` not being installed, install `serial` using this command: `sudo pip install pyserial`
4. Interact with the WordListSQL app on your device until the trace stops, which happens when time is up. Create, edit, and delete an item. If you need more time to perform these actions, run the command with `--time=30`, as you did with the Android Device Monitor example above.

Here is an example command-line command and results:

```
$ cd android-sdk/platform-tools/systrace

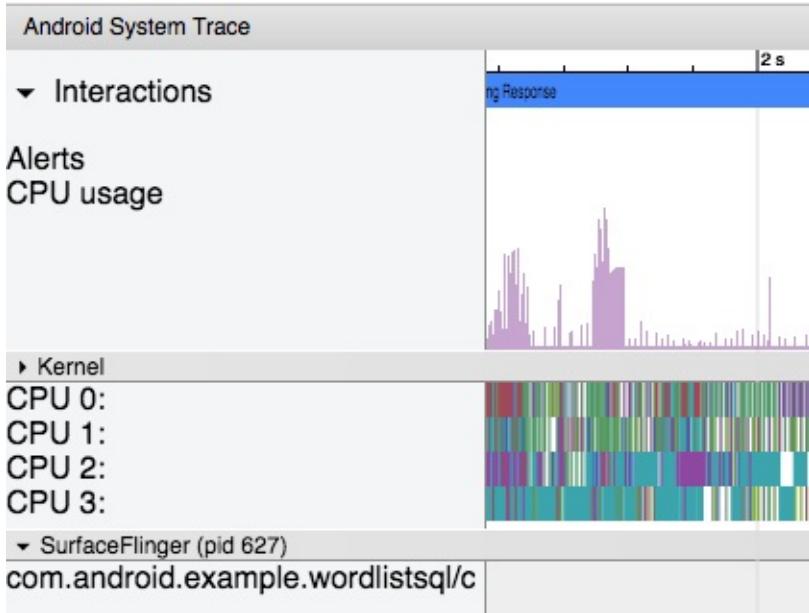
$python systrace.py --time=10
These categories are unavailable: disk
Starting tracing (10 seconds)
Tracing completed. Collecting output...
Outputting Systrace results...
Tracing complete, writing results
Wrote trace HTML file: file:///Users/<you>/sdk/platform-tools/systrace/trace.html
```

## 1.3 View and explore the Systrace output

Analyzing the whole trace file is beyond the scope of this practical. Systrace is a powerful tool, and the best way to learn is by using it a lot, by spending time looking at all the outputs. In this task, you learn to navigate the trace output and look at the **Frames** section.

The file generated for WordListSQL is large, and for a more complex app will be even larger. Practice navigating around the file to find information about your app.

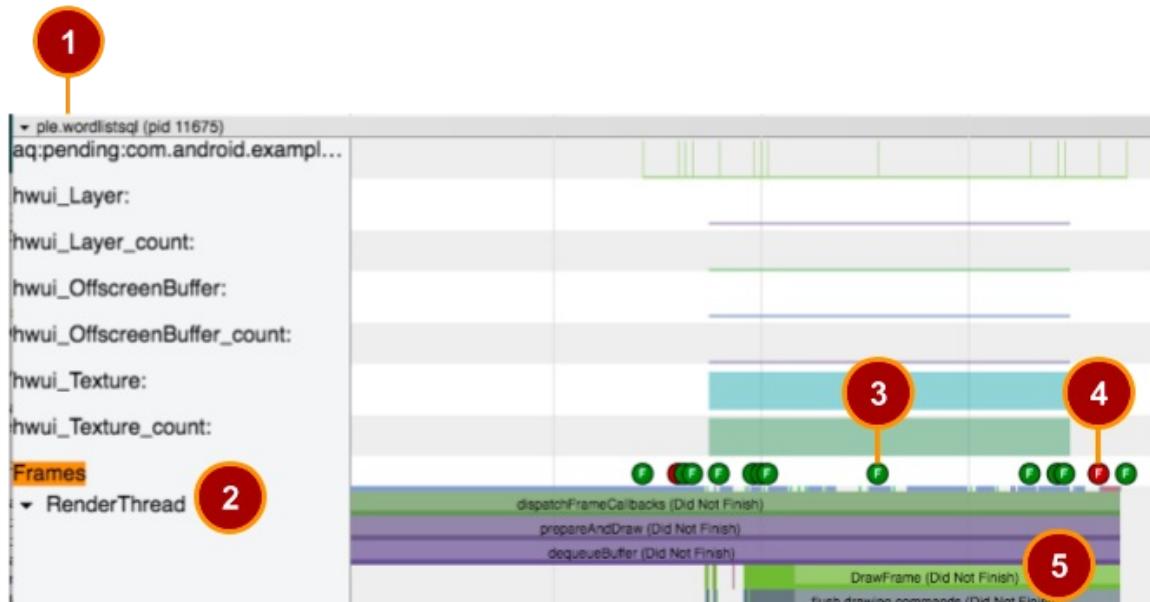
1. Open one of your saved `trace.html` files in your Chrome browser. At this time, you must use Chrome. Using a different browser may show a blank page. The file should look similar to the one shown below.



The groupings are in the order **Kernel**, **SurfaceFlinger** (the Android compositor process), followed by apps, each labeled by package name. Each app process contains all of the tracing signals from each thread it contains, including a hierarchy of high level tracing events based on the enabled tracing categories.

You can navigate the file using these keyboard controls:

- **w**—Zoom into the trace timeline.
- **s**—Zoom out of the trace timeline.
- **a**—Pan left on the trace timeline.
- **d**—Pan right on the trace timeline.
- **e**—Center the trace timeline on the current mouse location.
- **g**—Show grid at the start of the currently selected task.
- **Shift+g**—Show grid at the end of the currently selected task.
- **Right Arrow**—Select the next event on the currently selected timeline.
- **Left Arrow**—Select the previous event on the currently selected timeline.
- **m**—Mark this location with a vertical line so you can examine concurrent events in the trace.



The image shows part of the Systrace output for WordListSQL. The numbers have the following meanings:

1. Section for each process
2. Frames section
3. Frame indicator for frame that rendered successfully
4. Frame indicator for frame that did not complete rendering
5. Method calls for rendering selected frame

To explore the Systrace output:

1. The page has one section for each process that was running when the trace was recorded. Scroll through the very long file and find the section for the WordListSQL app, which is called `ple.wordlistsq1 (pid ...)`. The results will look different depending on how you interacted with your app, what device you used, and what configuration you used. You may need to scroll down quite a bit. If you get lost, close the tab and reopen the file to start afresh. **Tip:** Instead of scrolling, you can use the search function of your browser to search for **Frames** until you find the correct one.
2. Inside the `ple.wordlistsq1` section, find **Frames**.
3. In the **Frames** line, in the graph to the right, you should see circles that are either green or red and are labeled with the letter **F** for Frame. You may need to zoom in (**w** on the keyboard) and pan right (**d** on the keyboard) to see individual, labeled circles.

Select a green circle. This frame is green because it completed rendering in the allotted 16 milliseconds per frame.

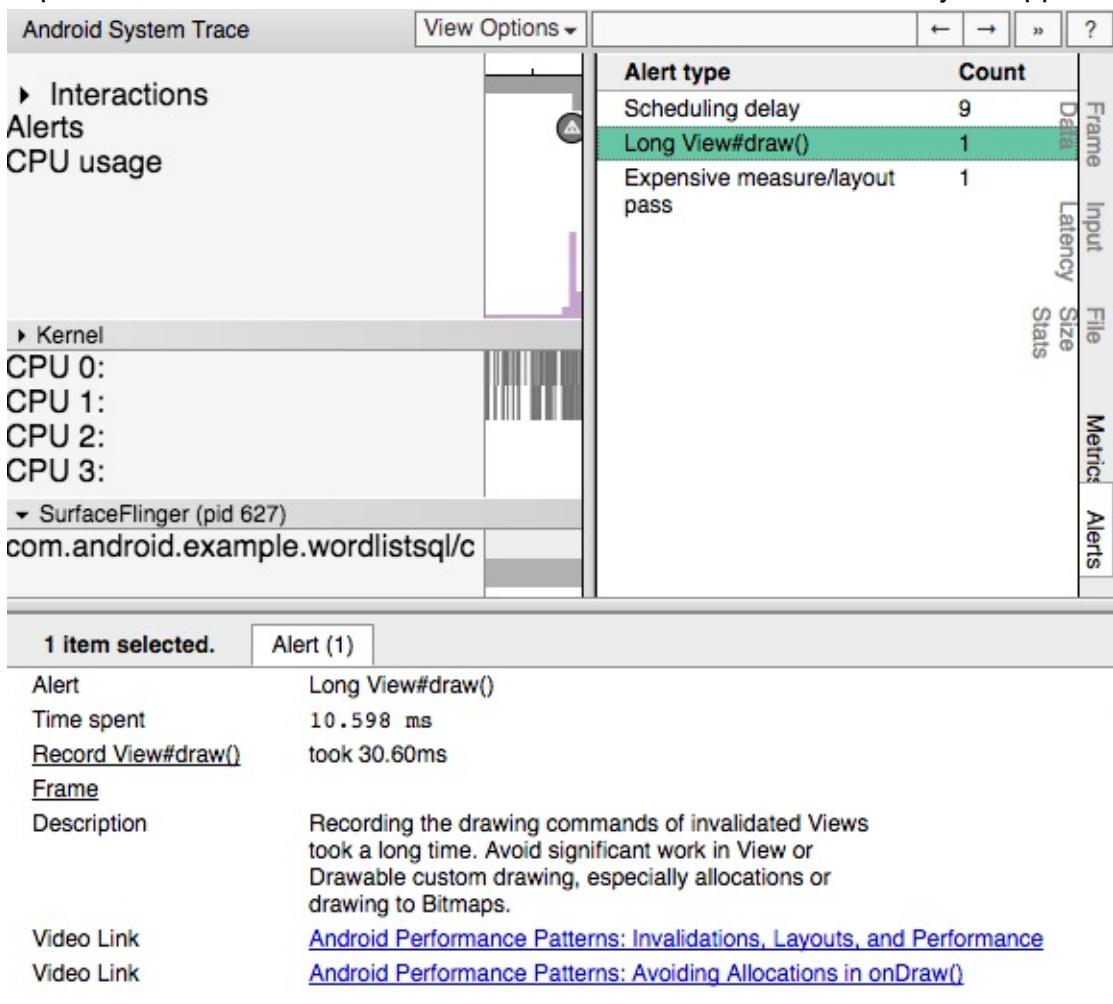
4. Select a red circle. This frame is red because it did not complete rendering within 16 milliseconds.

5. In the left-hand pane, scroll down if necessary until you find **Render Thread**. On devices running Android 5.0 (API level 21) or higher, this work is split between the **UI Thread** and **RenderThread**. On prior versions, all work in creating a frame is done on the UI thread. If you do not see **Render Thread**, select **UI Thread** instead.
6. Click on the black triangle to the right of **Render Thread** to expand the graph. This expands the bars that show you how much time was spent in each system action involved in rendering this frame.
7. Click on a system action to show more timing information in the pane below the graph. Systrace analyzes the events in the trace and highlights many performance problems as alerts, suggesting what to do next. Below is an example of a frame that was not scheduled, perhaps because the frame was waiting for other work to be completed.

Alert	Scheduling delay
Running	50.763 ms
Not scheduled, but runnable	519.709 ms
Sleeping	201.414 ms
Uninterruptible Sleep   WakeKill	1.268 ms
Blocking I/O delay	0.440 ms
<u>Frame</u>	
Description	Work to produce this frame was descheduled for several milliseconds, contributing to jank. Ensure that code on the UI thread doesn't block on work being done on other threads, and that background threads (doing e.g. network or bitmap loading) are running at android.os.Process#THREAD_PRIORITY_BACKGROUND or lower so they are less likely to interrupt the UI thread. These background threads should show up with a priority number of 130 or higher in the scheduling section under the Kernel process.

8. Open the **Alerts** tab on the right edge of the trace window. Click on an **Alert** type to see a list of all alerts in the bottom pane. Click an alert to see details, a description of what may be the problem, and links to further resources.

Think of the alerts panel as a list of bugs to be fixed. Often a tiny change or improvement in one area can eliminate an entire class of alerts from your app!



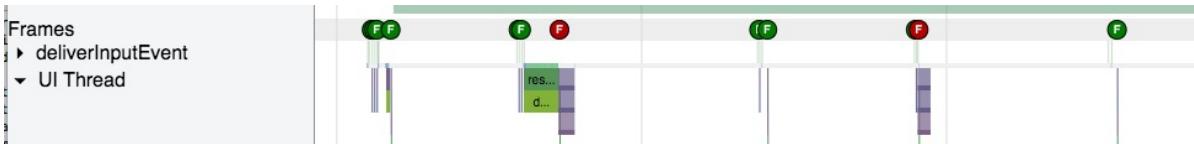
## Task 2. Run the Systrace tool to analyze the LargelImages app

### 2.1 Run Systrace on the LargelImages app

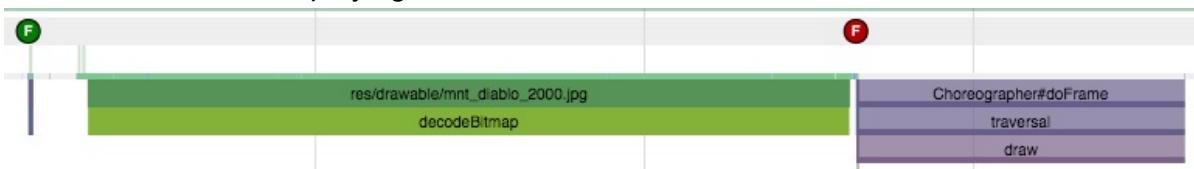
LargelImages is a smaller demo app and the Systrace output will be easier to analyze. In the previous practical, Profile GPU Rendering identified a problem with rendering some of the frames for this app. Use Systrace to get more information about possible causes of the problem. In this case, you may already have guessed what the problem may be. When doing performance analysis, guessing is a valid method of narrowing down your first approach. Be open to having guessed wrong!

1. Open the [LargelImages](#) app in Android Studio and run it on your connected device.
2. Start a Systrace trace that is 10 seconds long either from the command line or Android Device Monitor.
3. Flip a few times between the images of the running LargelImages app, going through at

- least two cycles.
4. When it has finished, open the trace in your Chrome browser. This trace should be easier to navigate than the trace from the WordListSQL, because this trace is a lot smaller.
  5. Find the section for the **largeimages** process.
  6. Find the **Frames** section for LargeImages and expand the **UI Thread**.
  7. If necessary, zoom (**w** and **s** keys on the keyboard) and pan (**a** and **d** keys on the keyboard) to see individual frames.



8. Notice the pattern of green and red frames. This pattern corresponds with the pattern of tall and short bars shown by Profile GPU Rendering in the Profile GPU Rendering practical. Turn on Profile GPU Rendering to refresh your memory, if necessary.
9. Click the red circle frame to get more information.
10. Looking at the alert information does not tell you specifics about possible causes.
11. In the **Frames** section, zoom in so that you can see the system actions on the UI thread that are involved in displaying this frame.



12. Notice `decodeBitmap` and above it, the name of the resource that is being decoded. Decoding the image is taking up all the time AND this decoding is being done on the UI Thread.

Now you know the cause of the problem. In a real app, you could address this problem by, for example, using a smaller or lower resolution image.

**Important:** Your trace may look different, and you may need to explore to find the information you need. Use the above steps as a guideline.

**Important:** Systrace is a powerful tool that gathers enormous amounts of systems data that you can use to learn more about the Android system and your app's performance. See the [systrace](#) documentation. Explore on your own!

## Task 3. Run the `dumpsystool`

`dumpsystool` is an Android tool that runs on the device and dumps information about the status of system services since the app started. You can use `dumpsystool` to generate diagnostic output for all system services running on a connected device. Passing the `gfxinfo`

command to `dumpsy` provides output in Android Studio's `logcat` pane. The output includes performance information that relates to frames of animation.

The purpose of this practical is to get you started with this powerful tool. The `dumpsy` tool has many other options that you can explore in the [dumpsy developer documentation](#).

## 3.1 Run `dumpsy` from the command line

1. Run the following command to get information about frames.

```
adb shell dumpsy gfxinfo <PACKAGE_NAME>
```

For example:

```
adb shell dumpsy gfxinfo com.example.android.largeimages
```

2. Starting with Android 6.0, you can get even more detailed information using the `framestats` command with `dumpsy`. For example, run the following command against a device with Android 6.0 or later:

```
adb shell dumpsy gfxinfo com.example.android.largeimages framestats
```

Learn more about how to use the data generated by this command in the [Testing UI Performance](#) and `dumpsy` documentation.

**Important:** This practical has only given you the first steps in using Systrace and `dumpsy`. Systrace and `dumpsy` are powerful debugging tools that provide you with a lot of information. Making efficient use of this information takes practice.

## Coding challenge

- The tracing signals defined by the system do not have visibility into everything your app is doing, so you may want to add your own signals. In Android 4.3 (API level 18) and higher, you can use the methods of the `Trace` class to add signals to your code. This technique can help you see what work your app's threads are doing at any given time. Learn more at [Instrument your app code](#).

## Summary

- Systrace is a powerful tool for collection runtime information about your app, for a slice of time. Use alerts generated by the tool to find potential performance issues. Use the

following command. The output is stored in `trace.html` in the `systrace` directory.

```
$cd android-sdk/platform-tools/systrace  
$python systrace.py --time=10
```

- The `dumpsy` tool collects comprehensive statistics about your app since its start time, including frame rendering statistics. Use these commands:

```
adb shell dumpsy gfxinfo <PACKAGE_NAME> adb shell dumpsy gfxinfo <PACKAGE_NAME>  
framestats
```

## Related concept

The related concept documentation is [Rendering and layout](#).

## Learn more

Android developer documentation:

- [systrace](#), including a walkthrough and a description of command-line options that are not covered in this practical
- [Android Profiler](#)
- [Testing UI Performance](#)
- [dumpsy](#)
- [Instrument your app code](#)
- [Trace class and trace.h File Reference](#)
- [Reading Bug Reports](#)
- [FrameStats](#)

## 4.2: Using the Memory Profiler tool

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Run the Memory Profiler tool](#)
- [Task 2. Dump and inspect the app heap](#)
- [Task 3. Record memory allocations](#)
- [Solution code](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

All processes, services, and apps require memory to store their instructions and data. As your app runs, it allocates memory for objects and processes in its assigned memory heap. This heap has a limited but somewhat flexible size. The Android system manages this limited resource for you by increasing or decreasing allocatable memory size. The system also frees memory for reuse by removing objects that are no longer used. If your app uses more memory than the system can make available, the system can terminate the app, or the app may crash.

- *Memory allocation* is the process of reserving memory for your app's objects and processes.
- *Garbage collection* is an automatic process where the system frees space in a computer's memory by removing data that is no longer required, or no longer in use.

Android provides a [managed memory environment](#). When the system determines that your app is no longer using some objects, the garbage collector releases the unused memory back to the heap. How Android finds unused memory is constantly being improved, but on all Android versions, the system must at some point briefly pause your code. Most of the time, the pauses are imperceptible. However, if your app allocates memory faster than the system can collect unused memory, your app might be delayed while the collector frees memory. The delay could cause your app to skip frames and look slow.

Even if your app doesn't seem slow, if your app leaks memory, it can retain that memory even while in the background. This behavior can slow the rest of the system's memory performance by forcing unnecessary garbage-collection events. Eventually, the system is

forced to stop your app process to reclaim the memory. When the user returns to your app, the app must restart completely.

To help prevent these problems, use the Memory Profiler tool to do the following:

- Look for undesirable memory-allocation patterns in the timeline. These patterns might be causing performance problems.
- Dump the Java heap to see which objects are using memory at any given time. Dumping the heap several times over an extended period can help you identify memory leaks.
- Record memory allocations during normal and extreme user interactions. Use this information to identify where your code is allocating too many or large objects in a short time, or where your code is not freeing the allocating objects and causing a memory leak.

For information about programming practices that can reduce your app's memory use, read [Manage Your App's Memory](#).

## What you should already KNOW

You should be able to:

- Create apps with Android Studio and run them on a mobile device.

## What you will LEARN

You will learn how to:

- Use Memory Profiler to collect data about your app.
- View Memory Profiler reports.
- Dump the Java heap and inspect it.
- Record memory allocation data for your app.

## What you will DO

- Run Memory Profiler and generate, save, and inspect data.

## App overview

- You will run the [LargeImages](#) and [RecyclerView](#) apps from previous practicals, using

the Memory Profiler.

- You will run the [MemoryOverload](#) app, which creates thousands of views, eventually using up all available memory.

<https://github.com/google-developer-training/android-advanced/tree/master/LargeImages>

# MemoryOverload

01234567891011121314151617181920212223242526272829303  
1

01234567891011121314151617181920212223242526272829303  
1

01234567891011121314151617181920212223242526272829303  
1



# Task 1. Run the Memory Profiler tool

[Android Profiler](#) is a set of tools that provide real-time information about your app, such as memory allocation and network usage. You can capture and view data as your app runs, and store data in a file that you can analyze in various viewers.

In this practical, you learn the basics of using [Memory Profiler](#) to track down performance problems and crashes related to your app's memory usage.

If you did any of the previous performance tools practicals, your environment is already set up for debugging with the Android Profiler. Otherwise, see the [Prerequisites](#).

## 1.1 Start the Memory Profiler tool with the LargelImages app

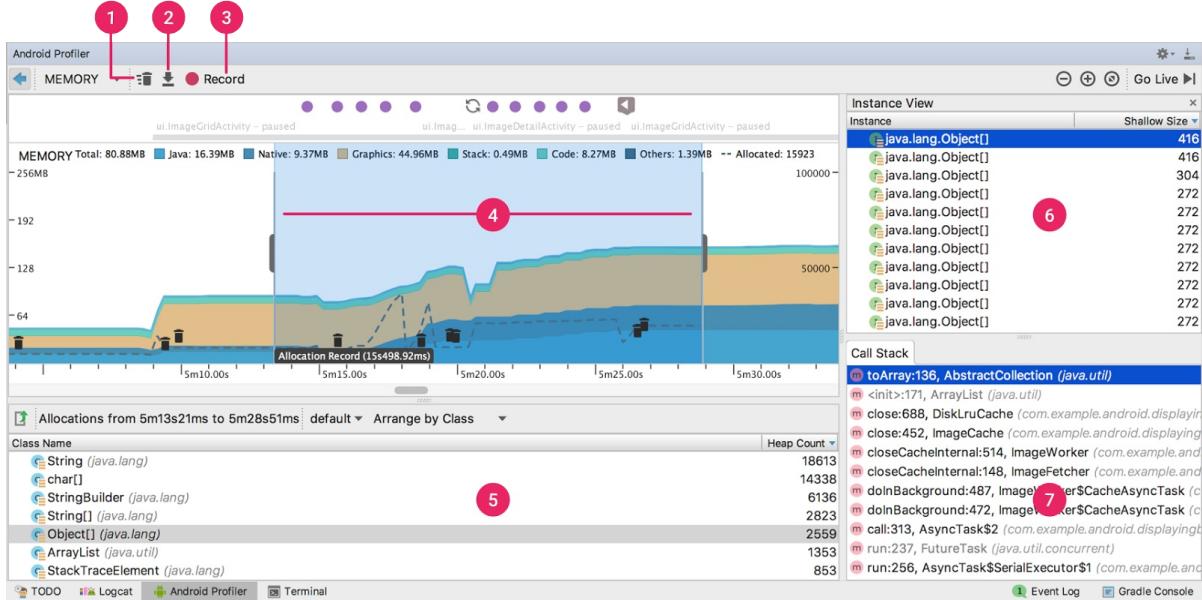
1. Open the [LargelImages](#) app in Android Studio.
2. Run LargelImages on a device with **Developer options** and **USB Debugging** enabled.  
If you connect a device over USB but don't see the device listed, ensure that you have [enabled USB debugging on the device](#).

For the next steps, use the screenshot below as a reference.

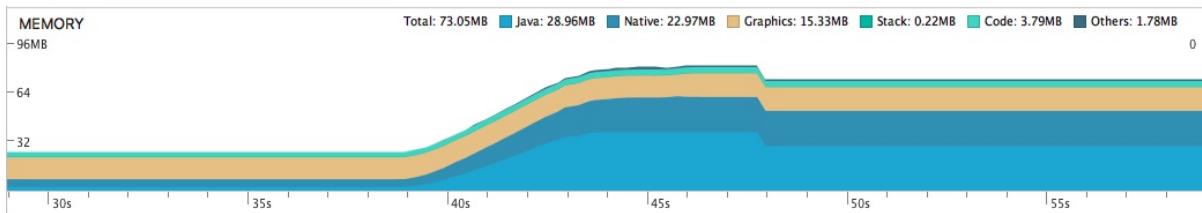
1. To open the Android Profiler, at the bottom of Android Studio click the **Android Profiler** tab (shown as 1 in the screenshot).
2. Select your device and app, if they are not automatically selected (2 in the screenshot).

The Memory graph starts to display. The graph shows real-time memory use (3). The x - axis shows time elapsed, and the y -axis shows the amount of memory used by your app (4).

3. In the app, swap the images to see the Memory graph change.

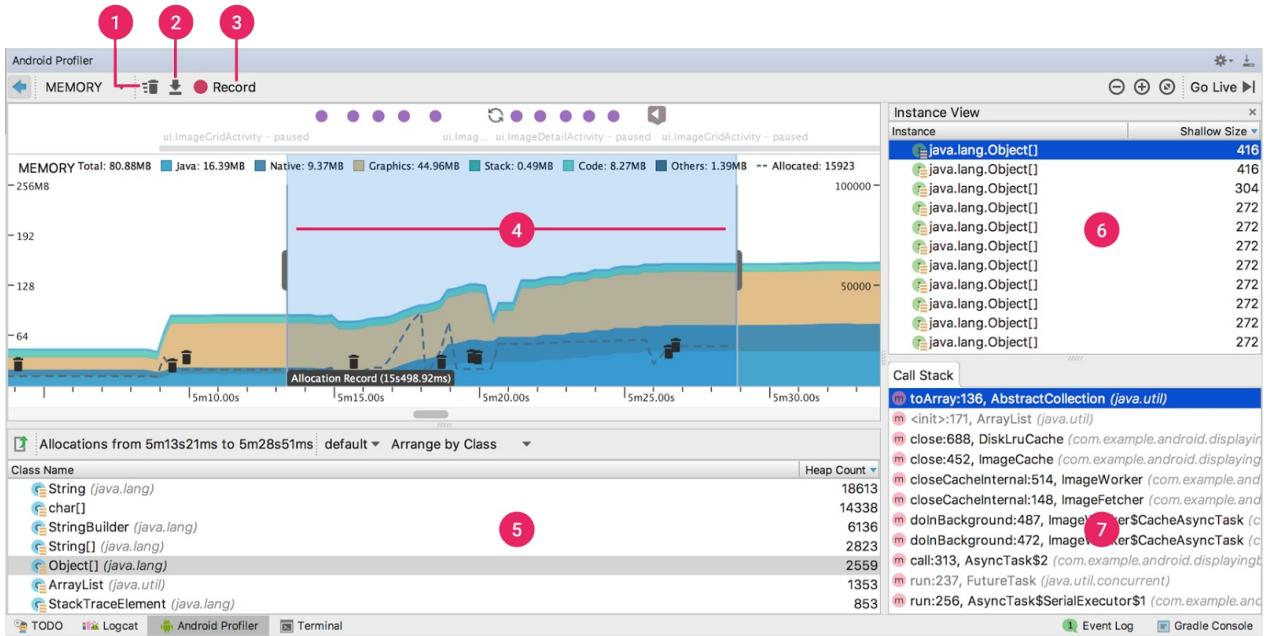


4. Click in the Memory graph, and the graph expands and separates into memory types. Each memory type (such as Java, Native, and Graphics) is indicated with a different color in a stacked graph. Check the key along the top of the graph to match colors and memory types.



## 1.2 Read about the Memory Profiler tool

Familiarize yourself with the [Memory Profiler](#) user interface, panes, and features with the help of the annotated screenshot below. Not all of these tools are open when you start Memory Profiler.



The Memory Profiler panes and features that you use in this practical are shown in the screenshot, as follows:

- (1) Force garbage collection. Small Trash icons in the graph indicate garbage-collection events that you or the system triggered.
- (2) Capture a heap dump and display its contents.
- (3) Record memory allocations and display the recorded data.
- (4) The highlighted portion of the graph shows allocations that have been recorded. The purple dots above the graph indicate user actions.
- (5) Allocation-recording and heap-dump results appear in a pane below the timeline. This example shows the memory allocation results during the time indicated in the timeline.
- (5 and 6) When you view either a heap dump or memory allocations, you can select a class name from this list (5) to view the list of instances on the right (6).
- (7) Click an instance to open an additional pane. When you are viewing the allocation record, the additional pane shows the stack trace for where that memory was allocated. When you are viewing the heap dump, the additional pane shows the remaining references to that object.

See the [Memory Profiler documentation](#) for a full list of controls and features.

## 1.3 Run Memory Profiler for the MemoryOverload app

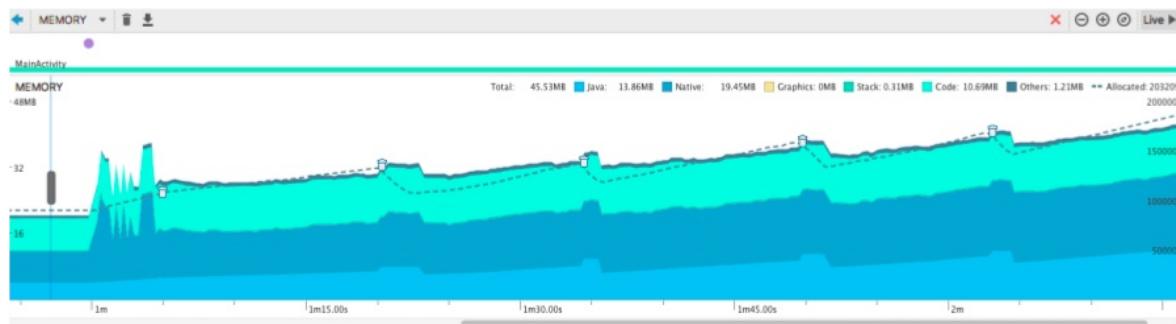
A *memory leak* is when an app allocates memory that is never freed, even after the memory is no longer needed. A memory leak can happen when an app allocates many objects and does not free unused or dereferenced objects. Memory leaks can slow down an app or in

the worst case, eventually make the app crash. Finding and fixing memory leaks is a lot easier if you have a tool that shows you what's happening with the memory that your app is using.

To demonstrate a memory leak, the `MemoryOverload` app creates and loads hundreds of `TextView` objects at the tap of a button. When you run the app and monitor it with Memory Profiler, you see a graph that shows more and more memory being allocated. Eventually, the app runs out of memory and crashes.

1. Download [MemoryOverload](#).
2. Run the app.
3. In Android Studio, open the Android Profiler. Click in the Memory graph to see the detail view for Memory Profiler.
4. In the app, tap the floating action button (+). Wait until the app is done adding the views to the screen. This may take a while. In Memory Profiler, observe how the app is using more memory as views are added.
5. Tap the + button a few more times. Your app will generate a graph similar to the one shown below. The graph shows more and more memory used and few, small, or no garbage-collection events. This allocation-graph pattern can indicate a memory leak. (The graph can look very different for different devices.)

Profile GPU Rendering from your device's **Developer options**.



6. Keep adding views until the app crashes and shows an Application Not Responding (ANR) dialog. Logcat displays a message like this one:

```
03-24 13:05:05.226 10057-10057/com.example.android.memoryoverload A/libc: Fatal signal 6 (SIGABRT), code -6 in tid 10057 (.memoryoverload)
```

`SIGABRT` is the signal to initiate `abort()` and is usually called by library functions that detect an internal error or some seriously broken constraint.

7. After overloading and crashing your device, it is a good idea to remove the app from your device and restart your device.

The MemoryOverload app is a made-up example to show a pattern, and it does not follow best practices! However, allocating and not releasing views is a common cause of memory problems. See the [Memory and Threading video](#) for more on this topic.

One fix for the MemoryOverload app would be to not create views that are not visible on the screen. A second solution would be to combine views. Instead of creating a view for each rectangle in a row, you could create a patterned background of rectangles and show multiple rectangles in one view.

## 1.4 Run Memory Profiler for RecyclerView

1. Run the [RecyclerView](#) app.
2. In the app, scroll through the items while Memory Profiler is open.
3. Click the floating action button (+) to add a lot more list items You may see changes in memory allocations and some spikes in the graph.
4. Scroll through all of the items again. Notice that the memory bar is flat when you are scrolling. [RecyclerView](#) is an efficient way of displaying lists, because views that become invisible are reused to display new content as the list is scrolled.

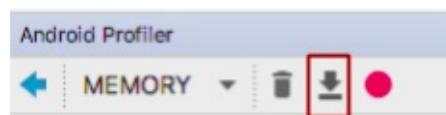
#WIDTH: 798.66[IMMAGEINFO]:

dg\_memory\_monitor\_recyclerview.png, The Memory graph for the RecyclerView app is steady

## Task 2. Dump and inspect the app heap

### 2.1 Dump the Java (app) heap

1. Run the [MemoryOverload](#) app.
2. In Android Studio, open the Android Profiler.
3. Click the Memory graph to fill the **Android Profiler** pane with the detailed view.
4. In the MemoryOverload app, tap the floating action button (+) once to add a set of views. Wait for the row of views to appear.
5. In Android Studio, click the **Dump Java Heap** button  to capture the app heap into a file and open the list of classes. This can take a long time. The **Heap Dump** pane will

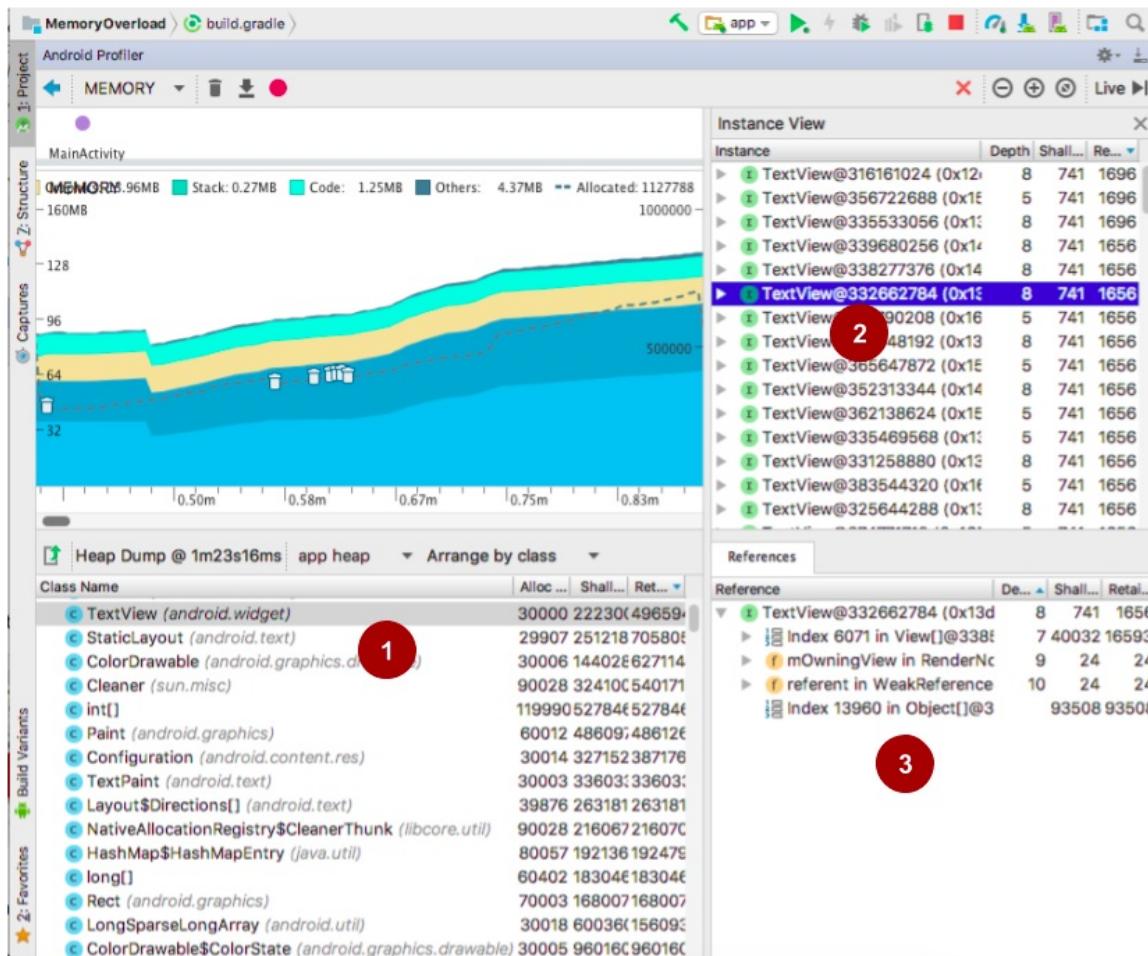


open after the heap dump is complete.

### 2.2 Inspect the dumped heap

Do the following to open all the information panes, then refer to the screenshot annotations for an explanation of each pane.

1. In the **Heap Dump** pane (1), find and click the `TextView` class. This opens an **Instance View** pane.
2. In the **Instance View** pane (2), click one of the `TextView` instances. This opens a **References** pane (3). Your screen should now look similar to the screenshot below.



3. The **Heap Dump** pane (1) shows all the classes related to your app, as they are represented on the heap. The columns give you size information for all the objects of this class. Click a column header to sort by that metric.
4. **Allocation Count:** Number of instances of this class that are allocated.
5. **Shallow Size:** Total size of all instances of this class.
6. **Retained Size:** Size of memory that all instances of this class are dominating.

After you click the floating action button (+) in the MemoryOverload app, you see a large number of `TextView` instances on the screen. Corresponding allocations are recorded on the graph and in the allocation count for the `TextView` class.

7. The **Instance View** pane (2) lists all the instances of the selected `TextView` class that are on the heap. The columns are as follows.

8. **Depth:** Shortest number of hops from any garbage-collection root to the selected instance.
9. **Shallow Size:** Size of this instance, in bytes.
10. **Retained Size:** Total size of memory being retained due to all instances of this class, in bytes.
11. The **References** pane (3) shows all the references to the selected instance. For example, in the MemoryOverload app, all the views are created and added to the view hierarchy. When you are debugging an app, look for classes and instances that should not be there, and then check their references.

For example, if you are offloading work to another thread, it is possible that references to views or activities remain after an `Activity` has been restarted, leaking memory on every configuration change. Look for long-lived references to `Activity`, `Context`, `View`, `Drawable`, and other objects that might hold a reference to the `Activity` or `Context` container.

12. Right-click a class or instance and select **Jump to Source**. This opens the source code, and you can inspect it for potential issues.
13. Click the **Export** button at the top of the **Heap Dump** pane to export your snapshot of the Java heap to an Android-specific Heap/CPU Profiling file in `HPROF` format. `HPROF` is a binary heap-dump format originally supported by J2SE. See [HPROF Viewer and Analyzer](#) if you want to dig deeper.

See [Memory Profiler](#), [Processes and Threads](#), and [Manage Your App's Memory](#).

## Task 3. Record memory allocations

Dumping the heap gives you a snapshot of the allocated memory at a specific point in time. Recording allocations shows you how memory is being allocated over a period of time.

### 3.1 Record allocations

1. Run the MemoryOverload app.
2. In Android Studio, open the Android Profiler.
3. Click the Memory graph to fill the Android Profiler pane with the detailed view.

4. Click the **Record Memory Allocations** button .



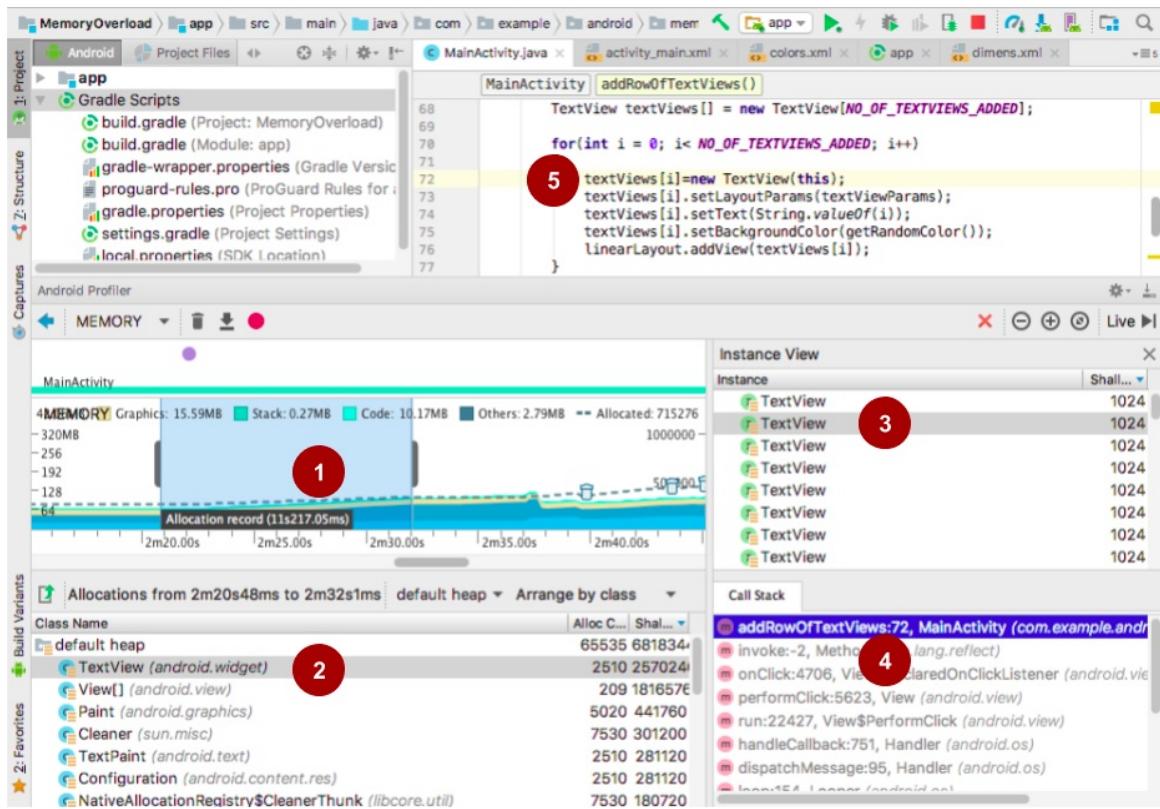
5. On your device with the MemoryOverload app running, tap the floating action button (+) to add a set of views.
6. Wait only a few seconds, then click the **Record Memory Allocations** button again. The button is now a black square  indicating that clicking it will pause the recording. Don't wait too long to pause the recording, because the recorded files can get large.

## 3.2 Inspect recorded allocations

Refer to the screenshot below for the next steps.

1. The Memory graph indicates which portion was recorded (1). Select the recorded portion, if necessary. You can record more than one section of the graph and switch between them.
2. As in the previous task, the **Heap Dump** pane (2) shows the recorded data.
3. Click a class name to see instances allocated during this period of time (3). As in the previous task, there should be many instances of the `TextView` class.
4. Click an instance to see its **Call Stack** (4).
5. At the top of the **Call Stack** (which is actually the bottom...) is the method call in your code that initiated creation of this instance. In this case, the method is  
`addRowOfTextViews()`.

Click `addRowOfTextViews` to locate this call in your code (5).



To export the recordings to an `hprof` file (for heaps) or an `alloc` file (for allocations), click the **Export**

 button in the top-left corner of the **Heap Dump** or **Allocations** pane. Load the file into Android Studio later for exploration.

## Solution code

Android Studio project: [MemoryOverload](#)

## Summary

- Use Memory Profiler to observe how your app uses memory over time. Look for patterns that indicate memory leaks.
- Use Java heap dumps to identify which classes allocate large amounts of memory.
- Record allocations over time to observe how apps allocate memory and where in your code the allocation is happening.

## Related concepts

The related concept documentation is in [Memory](#).

## Learn more

Android developer documentation:

- [Android Profiler overview](#)
- [Memory Profiler](#)
- [Overview of Android Memory Management](#)
- [Manage Your App's Memory](#)
- [Investigating your RAM Usage](#)

The following videos show older tools but the principles all apply.

- [Android Performance Patterns video series](#)
- [Memory Monitor](#)
- [Do Not Leak Views](#)
- [Memory and Threading](#)

## 4.3: Optimizing network, battery, and image use

### Table of Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Run the Network Profiler tool
- Task 2. Run battery statistics and visualization tools
- Task 3. Convert images to WebP format
- Summary
- Related concepts
- Learn more

Networking is one of the biggest users of battery. Optimizing your networking by following best practices will reduce battery drain, as well as reducing the size and frequency of data transfers.

In this practical, you learn how to use tools to measure and analyze network and battery performance. You also learn how to compress images to reduce the size of your data stream.

Optimizing network and battery performance is a huge topic, and as devices change, so do some of the details and recommendations. The Android team is constantly improving the framework and APIs to make it easier for you to write apps that perform well.

See the [Best Practices: Network, Battery, Compression](#) concept for an essential overview. Make use of the extensive linked resources to dive deeper and get the most up-to-date recommendations.

### What you should already KNOW

You should be able to:

- Create apps with Android Studio and run them on a mobile device.
- Work with **Developer options** on a mobile device.
- Start the [Android Profiler](#).

# What you will LEARN

You will learn how to:

- Monitor networking using the Networking Profiler tool.
- Dump and view battery usage using `batterystats` and Battery Historian 2.0.
- Convert images to WebP format.

# What you will DO

- Run the Networking Profiler tool.
- Dump battery statistics and display them as a chart using Battery Historian 2.0.
- Convert an image to WebP format using Android Studio.

## App overview

- You'll use the [WhoWroteIt](#) app, or any app of your choice that makes network calls.
- You'll create a simple demo app with a large image and an image converted to WebP format.

## Task 1. Run the Network Profiler tool

[Android Profiler](#) includes the [Network Profiler](#) tool, which makes it possible to track when your app is making network requests in real time. Using the Network Profiler, you can monitor how and when your app transfers data. With this information, you can optimize your code to reduce the frequency of data transfers, and optimize the amount of data transferred during each connection.

In this task you run [Network Profiler](#) with the [WhoWroteIt](#) app.

### 1.1 Modify and run the WhoWroteIt app

1. Download the [WhoWroteIt](#) app and open it in Android Studio.
2. To help demonstrate the tools, modify the `MainActivity.searchBooks()` method of the WhoWroteIt app:
  - Make multiple calls for each search. This will save you some tedious on-screen typing.
  - Call the `sleep()` method between requests to ensure that each network call is

made separately—that is, to ensure that the system does not batch the calls for you.

Obviously, you should not do either of these things in a production app.

In the `MainActivity.searchBooks()` method, replace this code:

```
// If the network is active and the search field is not empty, start a FetchBook A syncTask.  
if (networkInfo != null && networkInfo.isConnected() && queryString.length()!=0) {  
    new FetchBook(mTitleText, mAuthorText, mBookInput).execute(queryString);  
}
```

With this code:

```
// If the network is active and the search field is not empty, start a FetchBook A syncTask.  
if (networkInfo != null && networkInfo.isConnected() && queryString.length()!=0) {  
    new FetchBook(mTitleText, mAuthorText, mBookInput).execute(queryString);  
    SystemClock.sleep(3000);  
    new FetchBook(mTitleText, mAuthorText, mBookInput).execute(queryString);  
}
```

3. Run your app and perform one search. The app's user interface has not changed, and you should see one search result for your query.

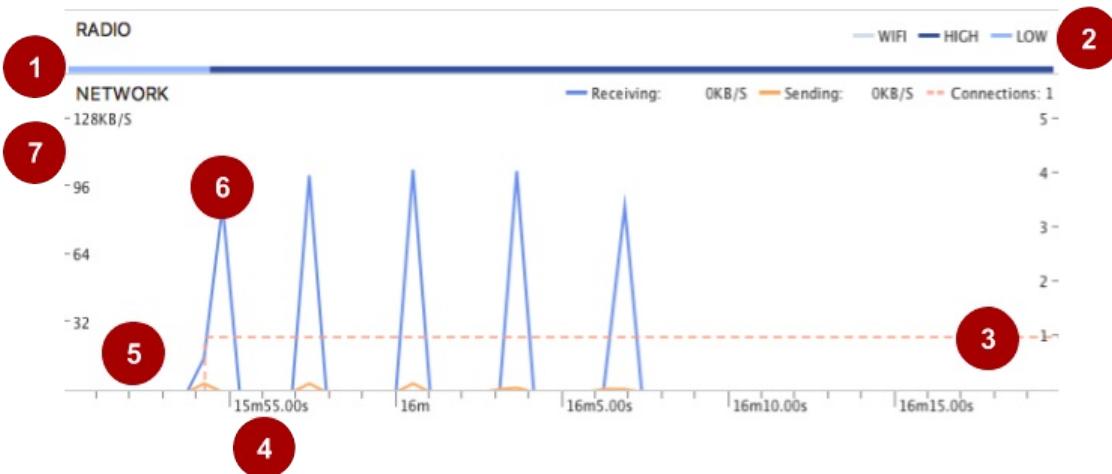
## 1.2 Run the Network Profiler tool

1. On your device, turn off Wi-Fi to make sure that your device uses the mobile radio for networking requests. Make sure that mobile data is enabled on the device.
2. Run your app on the device. If possible, do this on a physical device to get accurate measurements.
3. In Android Studio, open **Android Profiler**.
4. Enter a book title in the app and click the **Search Books** button. The Network Profiler should display a summary graph similar to the one below. The pattern may look different for different devices and network connections.
  - Orange spikes indicate data sent, and blue spikes data received. The WhoWroteIt app does not send a lot of data, so the orange spikes are short, while the blue spikes are much taller.

- The horizontal axis moves in time, and the vertical axis shows data transfer rate in kilobytes per second.
- Every time your app makes a network request, a vertical spike on the Network Profiler indicates the activity.
- The width of the base of the spike is how long the request took. The height of the spikes is the amount of data sent or received.



5. Click the **Network Profiler** section of the Android Profiler to expand the pane and see more details, as shown in the annotated screenshot below. To keep the graph from moving, click the **Live** button in the top-right corner of the profiler, or just scroll back. Doing this does not stop the recording.



6. The horizontal colored bar at the top indicates whether the request was made on Wi-Fi (gray) or the mobile radio (dark or light blue). The power state of the mobile radio is indicated by dark blue for high (using more battery power) and light blue for low (using less battery power).
7. Legend for type of radio used and power state of mobile radio.
8. The orange dotted line indicates the number of active connections over time, as shown on the y-axis on the right.
9. The x-axis shows the time that has passed.
10. Orange spikes mark data sent. The width indicates how long it took, and the height how much data was sent. The WhoWroteIt app requests are small, so it does not take long to send this small about of data.
11. Blue spikes mark data received. The width indicates how long data was received and the height how much data was received. Depending on how many books match the

- request, the size of the received data, and how long it takes to receive it, can vary.
12. The y-axis on the left shows numeric values for the amounts of data in KB per second.
  13. Experiment with different queries to see whether it slightly changes the network request patterns.

## 1.3 Network Profiler details

To show you advanced profiling data, Android Studio must inject monitoring logic into your compiled app. Features provided by advanced profiling include:

- The event timeline on all profiler windows
- The number of allocated objects in the Memory Profiler tool
- Garbage collection events in the Memory Profiler tool
- Details about all transmitted files in the Network Profiler tool

To enable the advanced profiling in Android Studio, follow these steps:

1. Select **Run > Edit Configurations**.
2. Select your app module in the left pane.
3. Click the **Profiling** tab, and then select **Enable advanced profiling**.
4. Now build and run your app again to access the complete set of profiling features.  
Advanced profiling slows your build speed, so enable it only when you want to start profiling your app.
5. Open the **Android Profiler** tab.
6. Click on the Network Profiler graph to open the detail view, then click on it again to open the advanced profiling panes.
7. On your connected mobile device, search for a book to get data moving over the network and generate a graph. The example below uses a query for the word "jungle".

Refer to the annotated screenshot below for the next steps.

1. Scroll back the graph to stop real-time viewing.

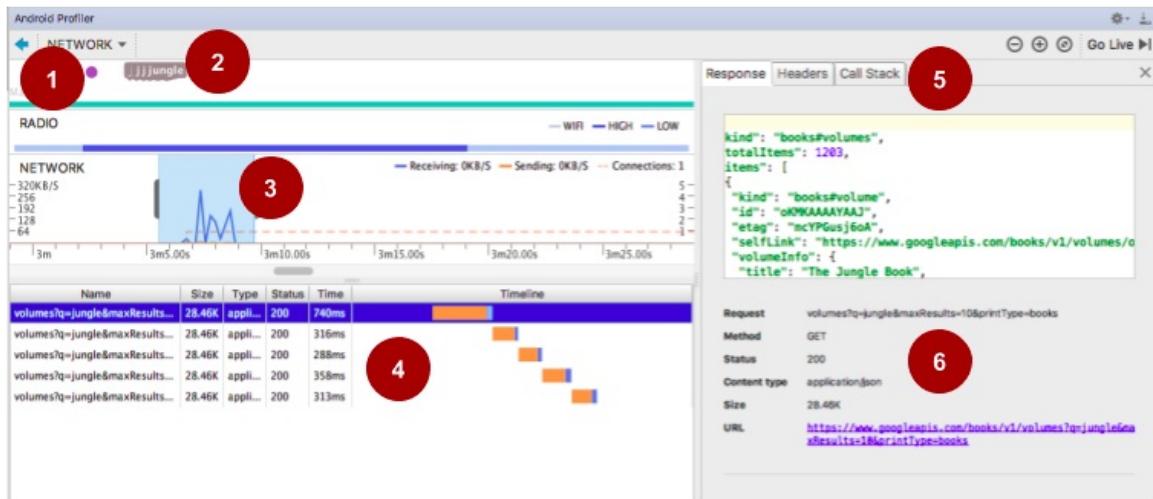
Note the purple dot at the top of the pane (1), which indicates a user action. In this case, the action was pressing the **Search Books** button.

Next to the user action, you see the payload for your request, which in this case consists of the word "jungle" (2).

2. Select a portion of the graph to get more details about it (3).

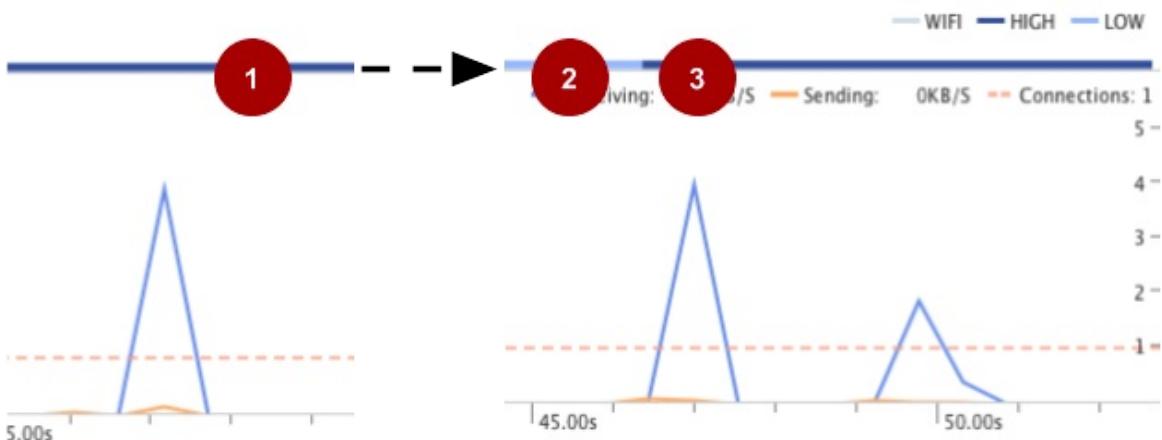
A pane opens below the graph listing all the requests and responses with additional information in the selected portion of the graph (4).

3. Click on a request, and a new pane with three tabs opens (5). By default, you see the contents of the first tab, displaying the contents of the response (6).

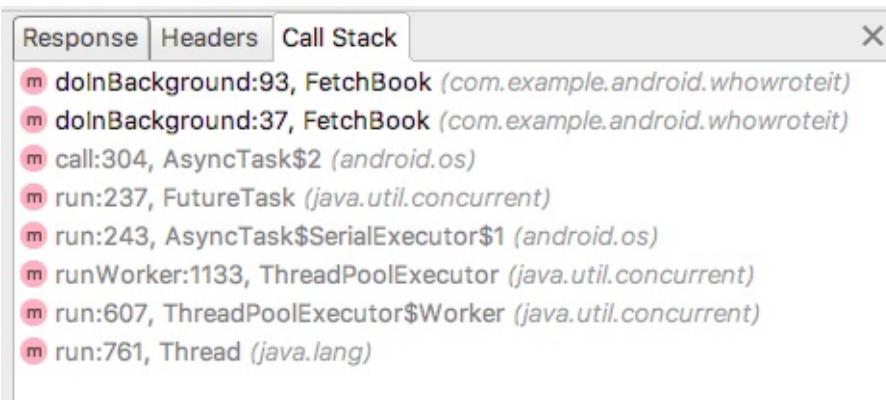


4. In the screenshot above, notice that the blue bar above the selected portion of the graph remains dark, indicating that the radio is in a high-power state throughout the requests. This is because the requests follow each other closely enough to keep the radio on.
5. In the WhoWroteIt app, change the sleep time for one of the requests until the radio powers off between requests. How long the mobile radio stays in the high state can vary greatly between devices.

In the screenshot below, after a request, the radio is in the high-power state (1). The radio stays in the high-power state for almost 30 seconds before powering down (2). Then the radio powers up again for the next request (3).



6. Click the **Call Stack** tab to see where the request originated in your code. When your app makes many different types of network requests, the **Call Stack** tab helps you pinpoint where in your code you may want to start optimizing.



When sending requests and receiving responses, try to optimize for power consumption of the mobile radio by minimizing the time the mobile radio is in a high-power state. This can be challenging, and a full explanation is beyond the scope of this practical. However, here are some tips to get you started.

- Limit the size of your requests and responses. Formulate your queries to only request the data you absolutely need.
- Batch your requests so that you can send several of them together without delay to take advantage of the high-power state. Space the sending of requests to maximize for the mobile radio staying in lower power states.

For more best practices, see [Optimizing Battery Life](#) and the [Best Practices: Network, Battery, Compression](#) concept.

## Task 2. Run battery statistics and visualization tools

`dumpsyst` is an Android tool that runs on the mobile device and dumps interesting information about the status of system services as you use the device. The `dumpsys` command retrieves this information from the device so that you can use it to debug and optimize your app.

You used `dumpsys framestats` in the [Systrace](#) and `dumpsys` [practical](#).

`BatteryStats` collects battery data from your device, and the `Battery Historian` tool converts that data into an HTML visualization that you can view in your browser. `BatteryStats` is part of the Android framework, and the `Battery Historian` tool is open-sourced and available on [GitHub](#). `BatteryStats` shows you where and how processes are drawing current from the battery, and it helps you identify tasks in your app that could be deferred or even removed to improve battery life.

You can use the [Battery Historian](#) tool on the output of the `dumpsyst` command to generate a visualization of power-related events from the system logs, as shown in the screenshot below. This information makes it easier for you to understand and diagnose battery-related issues. Battery Historian is not part of the Android framework, and you will need to download and install it from the internet.

In this practical, you create and work with a Battery Historian chart.

**WARNING:** Installing the Docker and Battery Historian tools requires downloading about 500 MB of data to your local computer, which may take some time. Make sure that you have sufficient time, bandwidth, and disk space before you begin.

**IMPORTANT:** Use the Chrome browser to work with Battery Historian files.



## 2.1 Install the Docker software management platform

The Battery Historian tool is open sourced and available on [GitHub](#).

There are different ways to install the tool. Using [Docker](#) is by far the easiest. Docker is an open platform for developers and system administrators to build, ship, and run distributed apps. See the `README.md` file with the [Battery Historian source code on GitHub](#) for more installation options and documentation.

To install and set up Docker:

1. On your computer, install the free community edition of Docker from <https://www.docker.com/community-edition>.
2. Start Docker. This may take some time, because more components will download.

3. Once Docker is ready, open a terminal window in Android Studio or on your desktop.
4. Enter the following commands in the terminal to verify that Docker is installed, running, and working properly:

```
docker --version  
docker ps  
docker run hello-world
```

5. Read Docker's output, as it summarizes how Docker works.

```
docker run hello-world  
Unable to find image 'hello-world:latest' locally  
latest: Pulling from library/hello-world  
78445dd45222: Pull complete  
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7  
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

## 2.2 Get the Battery Historian tool

1. Run the following Docker command.

```
docker run -p 1234:9999 gcr.io/android-battery-historian/stable:3.0 --port 9999
```

1234 is the port number for Battery Historian on your localhost. You can replace 1234 with the port number of your choice. (This might pull the Battery Historian, which might take a while depending on your internet speed.)

**Important:** Check the [README file for Battery Historian](#) in GitHub for the latest version number, because the tool gets updated and improved.

You will see a series of messages in the terminal, similar to the following:

```
Unable to find image 'gcr.io/android-battery-historian/stable:3.0' locally
3.0: Pulling from android-battery-historian/stable
c62795f78da9: Downloading 12.84 MB/45.56 MB
d4fceeb758e: Download complete
[...]
e06a9fa76cf2: Pull complete
Digest: sha256:265a37707f8cf25f2f85afe3dff31c760d44bb922f64bbc455a4589889d3fe91
Status: Downloaded newer image for gcr.io/android-battery-historian/stable:3.0
2017/08/04 18:33:34 Listening on port: 9999
```

Do not close the terminal or terminate the terminal window, because this will stop Battery Historian.

Linux and Mac OS X:

- That's it. Open Battery Historian, which will be available in your browser at <http://localhost:1234/>

Windows:

- You may have to [enable virtualization](#). If you are able to run the emulator with Android Studio, then virtualization is already enabled.
- Once you start Docker, it should tell you the IP address of the machine it is using. If, for example, the IP address is `123.456.78.90`, Battery Historian will be available at <http://123.456.78.90:1234>. (If you used a different port, substitute it in the URL.)

## Battery Historian

### Upload Bugreport

Both .txt and .zip bug reports are accepted.

The screenshot shows a web-based interface for uploading bug reports. At the top, there is a large dark header with the text "Battery Historian". Below it, a section titled "Upload Bugreport" is displayed. A file input field contains the file "bugreport.zip". To the left of the input field is a "Browse" button. To the right is a "Submit" button. Below the input field are two additional buttons: one for "Kernel Wakesource Trace" and another for "Powermonitor File". There is also a link labeled "Switch to Bugreport Comparisor".

## 2.3 Run commands to get battery statistics

Your mobile device collects statistics about itself as it runs. This information is useful for debugging as well as performance analysis. The `dumpsyst batterystats` command gets information related to battery usage from your device.

1. Connect your mobile device to your computer.
2. Turn off device Wi-Fi so that the device uses the mobile radio.
3. If you have not already done so, download the [WhoWroteIt](#) app and open it in Android Studio.

4. Run the WhoWroteIt app on your mobile device.
5. On your computer, open a terminal window. (Use the command prompt on a Windows system, or open the **Terminal** pane in Android Studio.)
6. From the command line, shut down your running `adb` server.

```
adb kill-server
```

7. Restart `adb` and check for connected devices. The command lists any connected devices.

```
$ adb devices
List of devices attached
emulator-5554      device
LGH918ce000b4b    device
```

8. Reset battery data gathering. Resetting erases old battery collection data; otherwise, the output from the dump command will be huge.

```
adb shell dumpsys batterystats --reset
```

9. Disconnect your mobile device from your computer so that you are only drawing power from the device's battery. On the emulator, you will not get accurate data for this exercise.
10. Play with the WhoWroteIt app for a short time. Search for different books by using different search text.
11. Reconnect your mobile device.
12. On your mobile device, turn Wi-Fi on.
13. Make sure that `adb` recognizes your mobile device.

```
adb devices
```

14. Dump all battery data and redirect the output to save it into a text file. This can take a while.

```
adb shell dumpsys batterystats > batterystats.txt
```

15. To create a report from raw data on a device running Android 7.0 and higher:

```
adb bugreport bugreport.zip
```

For devices 6.0 and lower:

```
adb bugreport > bugreport.txt
```

`bugreport` could take several minutes to complete. Do not cancel, close the terminal window, or disconnect your device until it is done.

16. On the Battery Historian starting page in your Chrome browser (see previous screenshot), browse for the `bugreport.txt` or `bugreport.zip` file. Click **Submit** to upload and display the chart. If you used **Terminal** pane in Android Studio, the `bugreport` file will be located in the WhoWroteIt app's root directory.

## 2.4 What you see in the Battery Historian chart

The Battery Historian chart graphs power-relevant events over time.

Each row shows a colored bar segment when a system component is active and drawing current from the battery. The chart does not show how much battery was used by the component, only that the app was active.

1. Hover over the bars to get details about each graph. You will work more with this in a minute.
2. Hover over the "i" information icon on the left to get a color legend, as shown in the

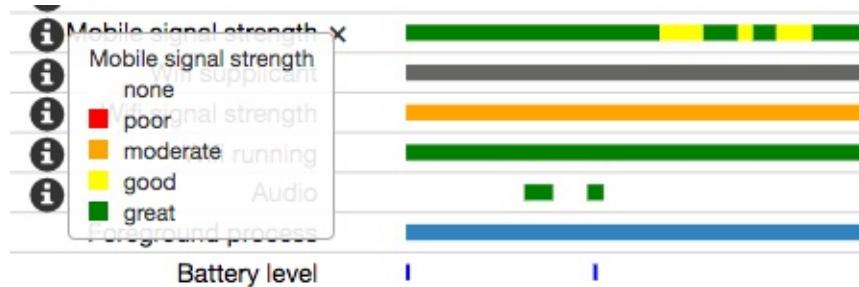


figure below.

3. Using the chart screenshot below and explanations below, explore your chart. Your chart looks different than the screenshot because you are on a different device, and you likely performed different actions.

## Chart guidelines

The following chart shows three minutes of information for an LG V20 mobile device running the modified WhoWroteIt app.



The numbers in the following list reference the numbered callouts in the image below, illustrating some of the powerful information you can gather and analyze.

- (1) The **Mobile radio active** line shows the activity of the mobile radio. The mobile radio is one of the biggest battery users. When working on battery efficiency, you always want to look at and optimize the mobile radio's activity.
- (2) *Movable timeline that displays information about battery charge state.* The upper box shows information about the current battery state. The bottom bars show that the device was not plugged in at this moment, and it was not charging. This information is useful when you want to verify that your code only performs certain battery intensive activities, such as syncs, when the device is plugged in and on Wi-Fi.
- (3) *Mobile radio activity.* The selected bar in the **Mobile radio active** line, and those nearby, are likely the WhoWroteIt app's network calls to fetch books. The information box for the bar underneath the timeline marker shows additional information such as the duration and the number of occurrences.

See the following documentation for more:

- [Battery Historian Charts](#)
- [Analyzing Power Use With Battery Historian](#)

## Task 3. Convert images to WebP format

Most download traffic consists of images fetched from a server. The smaller you can make your downloadable images, the better the network experience your app can provide for users. Downloading smaller images means faster downloads, less time on the radio, and potential savings of battery power. Using WebP images with your app sources reduces APK size, which in turn means faster app downloads for your users.

[WebP](#) is an image file format from Google. WebP provides lossy compression (as JPG does) and transparency (as PNG does), but WebP can provide better compression than either JPG or PNG. A WebP file is often smaller in size than its PNG and JPG counterparts, with at least the same image quality. Even using lossy settings, WebP can produce a nearly identical image to the original. Android has included lossy [WebP support](#) since Android 4.0 (API 14) and support for lossless, transparent WebP since Android 4.3 (API 18).

Serve [WebP](#) files over the network to reduce image load times and save network bandwidth.

In general, use the following algorithm to decide which image format to use:

```
Do you support WebP?  
  Yes: Use WebP  
  No: Does it need transparency?  
    Yes: Use PNG  
    No: Is the image "simple" (colors, structure, subject?)  
      Yes: Use PNG  
      No: Use JPG
```

### 3.1 Create an app and compare image sizes and quality for different formats

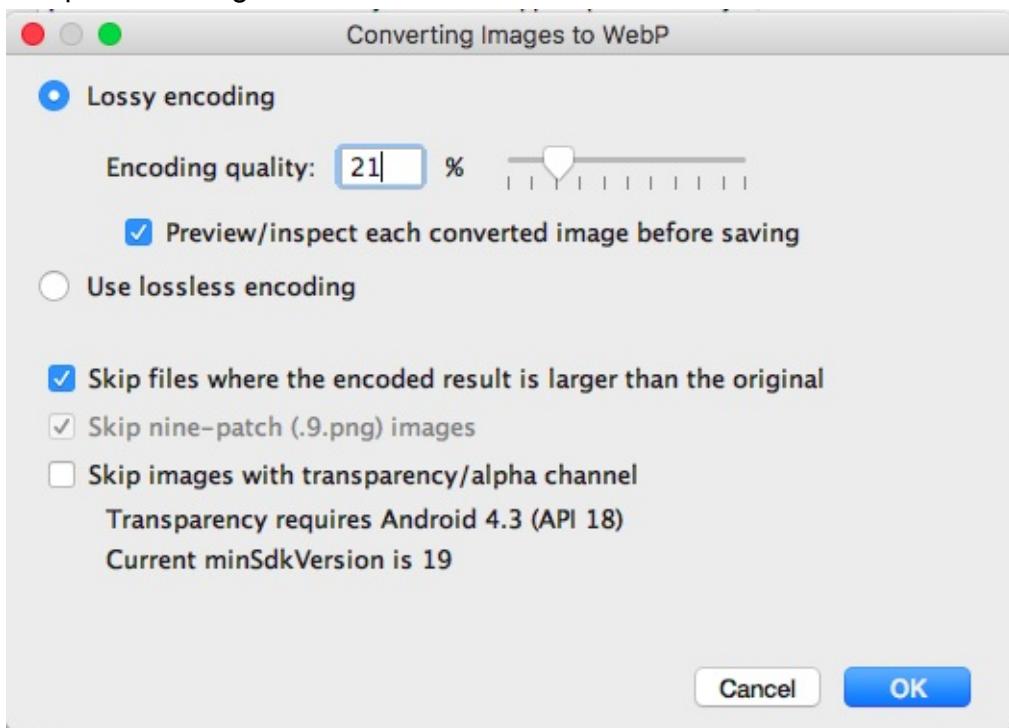
Android has included lossy [WebP support](#) since Android 4.0 (API 14) and support for lossless, transparent WebP since Android 4.2 (API 18).

Support for lossless and transparent WebP images is only available in Android 4.2 and higher. That means your project must declare a `minSdkVersion` of 18 or higher to create lossless or transparent WebP images using Android Studio.

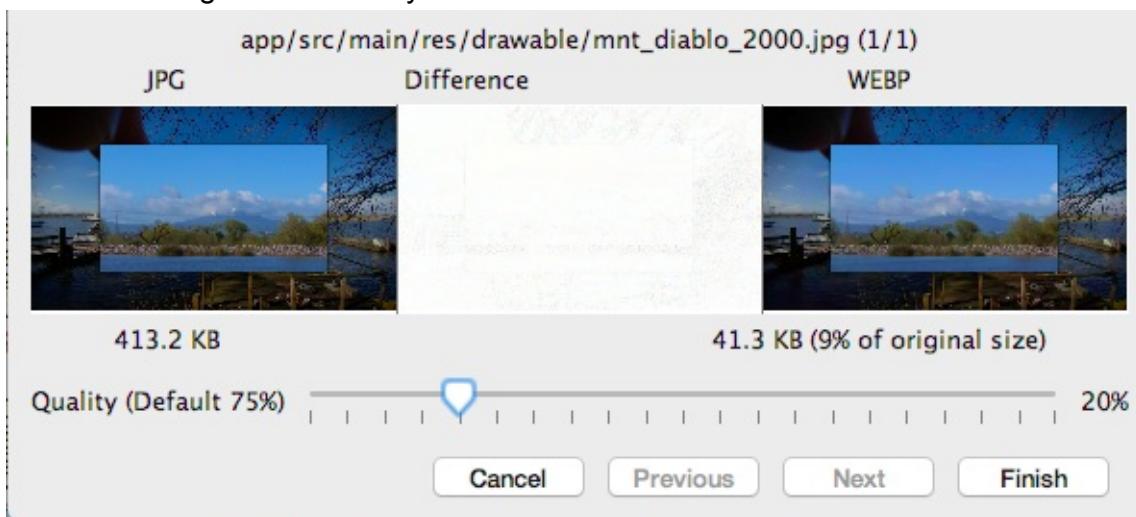
**Note:** You can create a new app for this task, or you can make a copy of the [Largelimages](#) app and modify it in accordance with the steps below.

1. Create an app with a `minSdkVersion` of 18 using the Basic Template.
2. Add the same large image that you used in the [Largelimages](#) app as the background for the text view.
3. Make a backup copy of the image in `res/drawable`. You need the backup because conversion occurs in place, replacing your original image.
4. In your `res/drawable` folder, right-click the image and choose **Convert to WebP** at the bottom of the menu. The **Convert Images to WebP** dialog opens.

This conversion happens "in-place"; that is, your original image is changed into the compressed image.



5. Choose **Lossy encoding**, move the slider to 21%, and click **OK** to preview the files. For this particular image, these settings will give you about 10% of the size for the converted image. This saves you over 350 K!



6. Click **Finish** to save the converted small image and rename it to `mnt_diablo_webp.webp`.
7. Add an `onClick()` handler to the text view that swaps the background between the backed up original and the WebP image.
8. Run your app.
9. Click to swap between the two images. Do you notice any difference in quality? In most cases the difference should be unnoticeable to casual user.

# Summary

- Network Monitor gives you a quick live look at how your app is making network requests.
- The `dumpsy batterystats` command gets battery-related information from your mobile device.
- The Battery Historian tool displays the `batterystats` information as an interactive time chart. You can use the chart to correlate battery usage with activities taken by your app.

## Related concepts

The related concept documentation is in [Best Practices: Networking, Battery, Compression](#) with extensive references to more documentation.

## Learn more

Performance is a huge and important topic, and there are many resources for you to dive deeper.

Network:

- [Android Profiler](#)
- [Inspect Network Traffic with Network Profiler](#)
- [Connectivity for billions](#)
- [Managing Network Usage](#)
- [Battery Drain and Networking \(video\)](#)
- [Transferring Data Without Draining the Battery: in-depth docs](#)

Battery:

- [BatteryStats and Battery Historian Walkthrough](#)
- [Battery Historian Charts](#)
- [Analyzing Power Use with Battery Historian](#)
- [Optimizing Battery Life: in-depth docs](#)
- [Battery Performance 101 \(video\)](#)
- [Battery Drain and Wake Locks \(video\)](#)
- [Battery Drain and Networking \(video\)](#)

WebP:

- [Create WebP Images](#)

- [WebP website](#)
- [Image Compression for Web Developers \(good, comprehensive intro\)](#)
- [Reducing Image Download Sizes](#)
- [Reduce APK Size](#)
- [Reduced data cost for billions](#)

# 5.1: Using resources for languages

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Add another language
- Task 2. Add a right-to-left (RTL) language
- Task 2 solution code
- Task 3. Add a Language option to the options menu
- Task 3 solution code
- Coding challenge
- Challenge solution code
- Summary
- Related concept
- Learn more

Users run Android devices in many different languages. To reach the most users, your app should handle text and layouts in ways appropriate for those languages.

An Android user can change the language for a device in the Settings app. As a developer, you should *localize* your app to support different languages in the locales in which your app is released.

In this chapter, you learn how to provide support for different languages using string resources and the Translations Editor in Android Studio.

## What you should already KNOW

You should be able to:

- Create and run apps in Android Studio.
- Extract string resources and use string resources in the code.
- Use the Layout Editor to align views in a `ConstraintLayout`.
- Edit layouts in XML.
- Change and add an options menu in the Basic Activity template.

## What you will LEARN

You will learn how to:

- Add support for different languages.
- Use the Translations Editor to edit translations.
- Modify layouts to support right-to-left (RTL) languages.

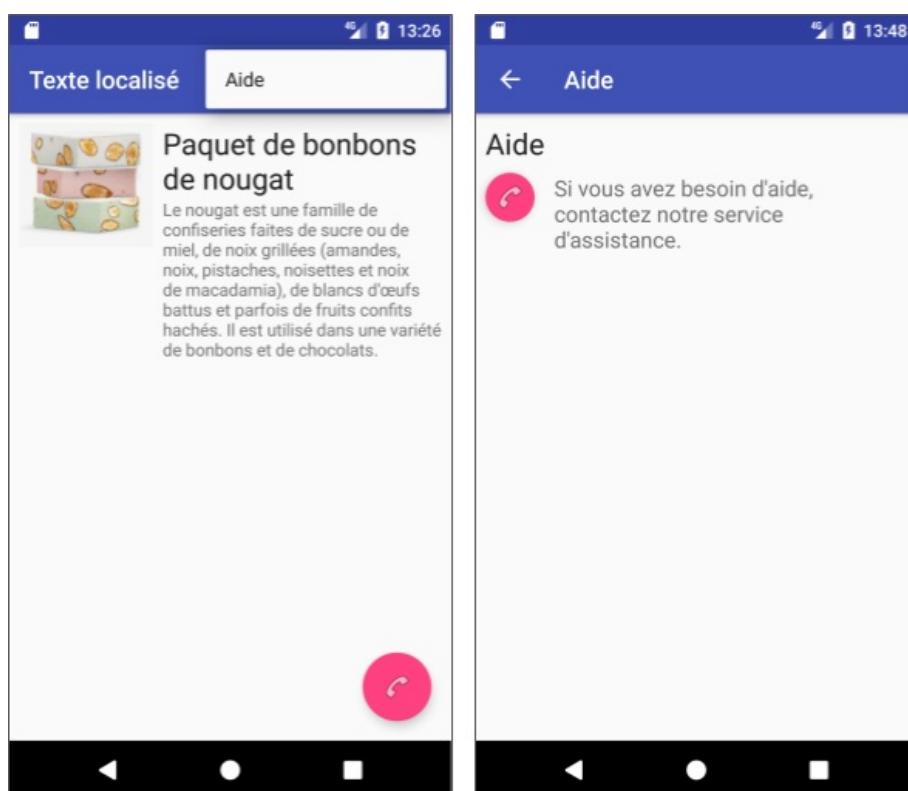
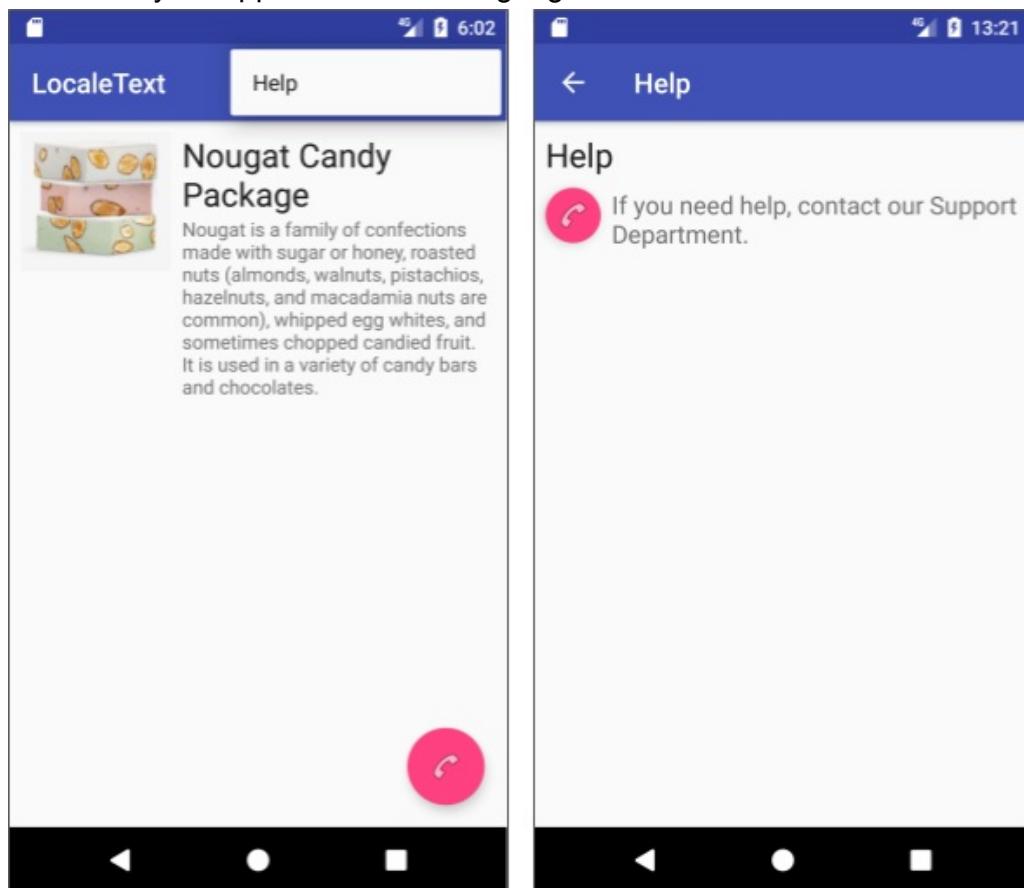
## What you will DO

- Add languages to an app.
- Use the Translations Editor to edit translations.
- Modify the app's layouts to support RTL languages, such as Hebrew.
- Add a **Language** option to the options menu to switch languages.

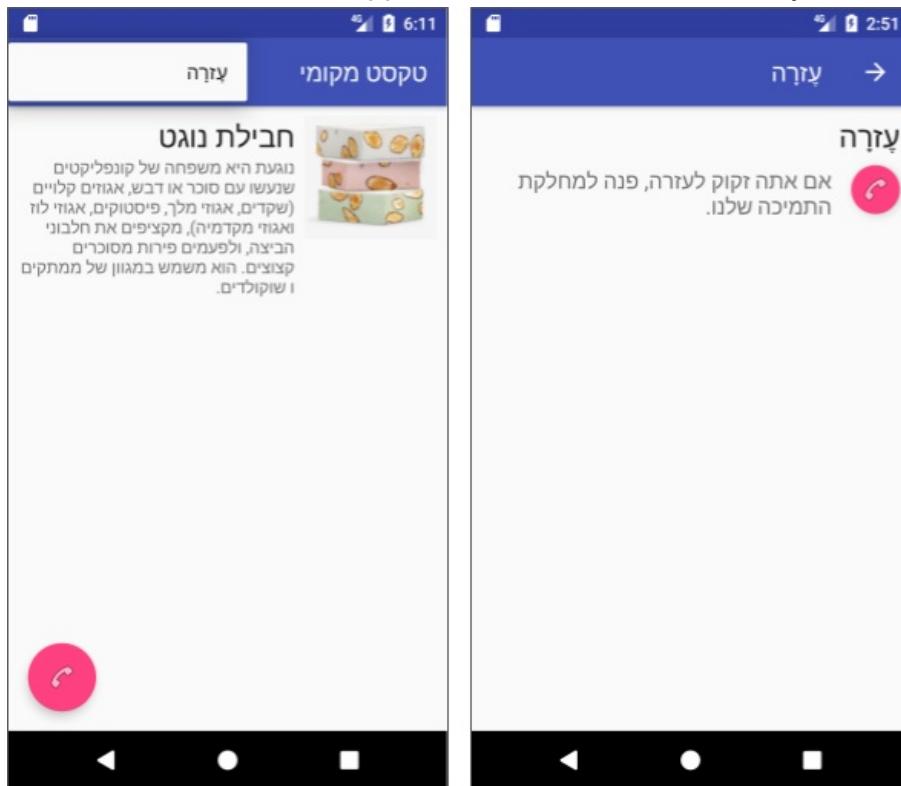
## Apps overview

In this lesson you work from a starter app called `LocaleText_start`, which is a simple app created using the Basic Activity template. Its UI uses the default `ConstraintLayout` provided by the template. It has text, an image, and a floating action button that can make a phone call. The options menu has a single menu item, **Help**, that launches a second activity with help information.

This lesson demonstrates how you can localize an app for a locale, and provide translated text within your app for the other languages.



After adding an RTL language, you will use XML layout attributes to take advantage of the layout mirroring feature added to Android 4.2. Layout mirroring redraws the layout for a right-to-left orientation for RTL languages. As a result, the image, the `TextView` elements, the options menu, and the floating action button are all automatically moved to the opposite side of the screen as shown below. The app demonstrates the attributes you use for layout



mirroring.

## Task 1. Add another language

In this task you will learn the following key practices for adding different languages:

- Save all text strings as resources in the default version of `strings.xml`.
- Add a new language in the Translations Editor.
- Translate the text.
- Test your app in different languages.

### 1.1 Examine the startup app's layout

To save time, the `LocaleText_start` starter app has been prepared with a layout to show the effects of changing languages.

1. Download the `LocaleText_start` app.
2. Rename the `LocaleText_start` project folder to `LocaleText`, open it in Android Studio, and refactor it to use the name `LocaleText`.
3. Run the app to see what it looks like.

4. Examine the app's layouts.

The `MainActivity` layout includes:

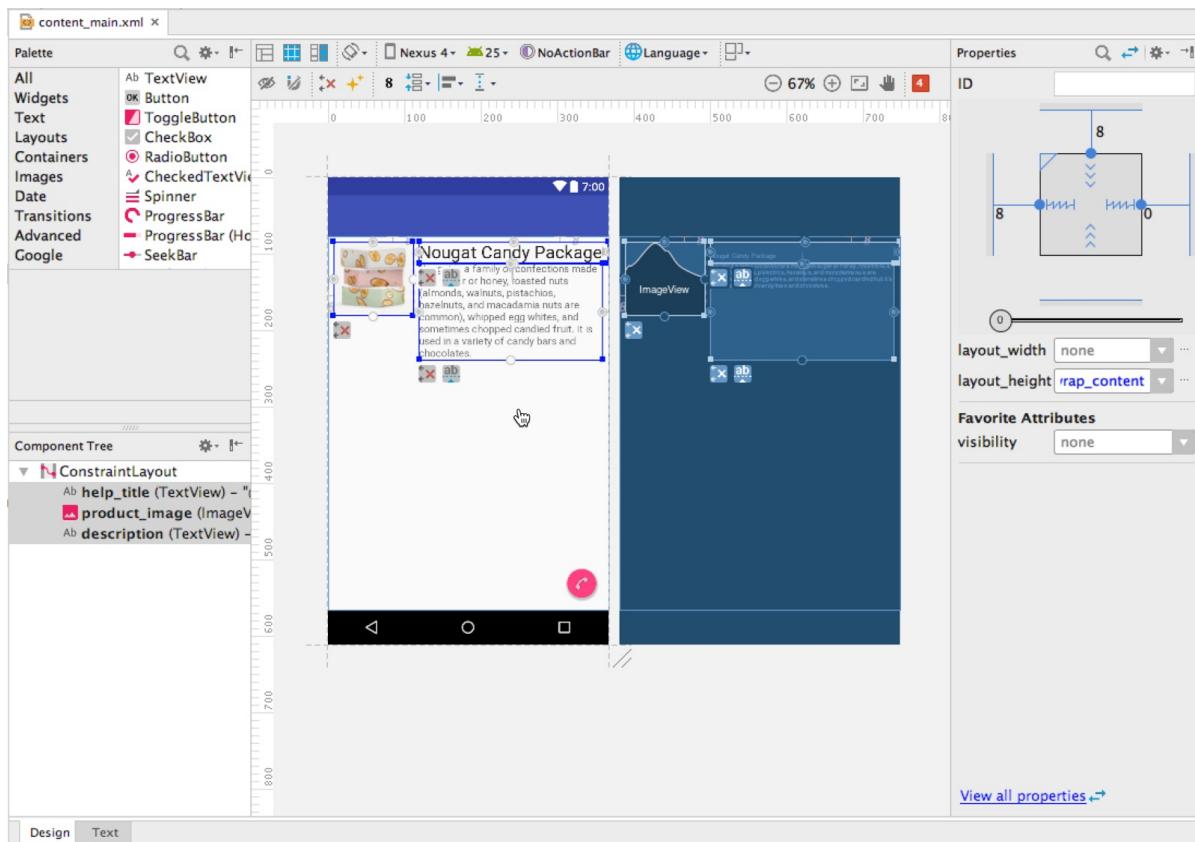
- An `ImageView` constrained to the left margin of the parent (the screen).
- A `TextView` with a heading constrained to the right side of the `ImageView`.
- Another `TextView` with description text, also constrained to the right side of the `ImageView`.
- A floating action button with a phone icon in the lower right side.
- The options menu in the app bar with one choice: **Help**.

The `HelpActivity` layout includes:

- A `TextView` for the title, constrained to the left margin of the parent (the screen).
- A floating action button with a phone icon on the left side.
- Another `TextView` with body text, constrained to the right side of the floating action button and to the title `TextView`.
- The **Up** button and `Activity` name in the app bar.

Open `content_main.xml` to see the `MainActivity` layout in the **Design** tab. You should see an `ImageView` and two `TextView` elements in the layout (shown in the figure below).

- The `product_image` `ImageView` is constrained to the top and left side of the screen.
- The `heading` `TextView` is constrained to the top and right side of the `product_image` `ImageView`, and to the right side of the parent.
- The `description` `TextView` is constrained to the bottom of `heading` `TextView` and to the right side of `product_image` `TextView` and parent/screen.



## 1.2 Examine the startup app's layout and resources

To prepare an app to be translated into another language, save all text strings as resources in `strings.xml`, including text from menu items such as the options menu, text on tabs, and any other navigation elements that use text.

To save time, the `LocaleText_start` starter app has been prepared with all text strings in `strings.xml`. Open `strings.xml`—known as the *default* `strings.xml` file because it is used for the default language—to see the string resources in the app. It includes every piece of text in the app including the `action_help` resource for the options menu item (**Help**). The file include comments that provide the context for the translator to translate your strings—such as the use of grammar structures and product terms.

```
<resources>
    <!-- All comments are notes to translators. -->
    <string name="app_name">LocaleText</string>

    <!-- The Help option in the options menu, no more than 30 chars. -->
    <string name="action_help">Help</string>
    ...
</resources>
```

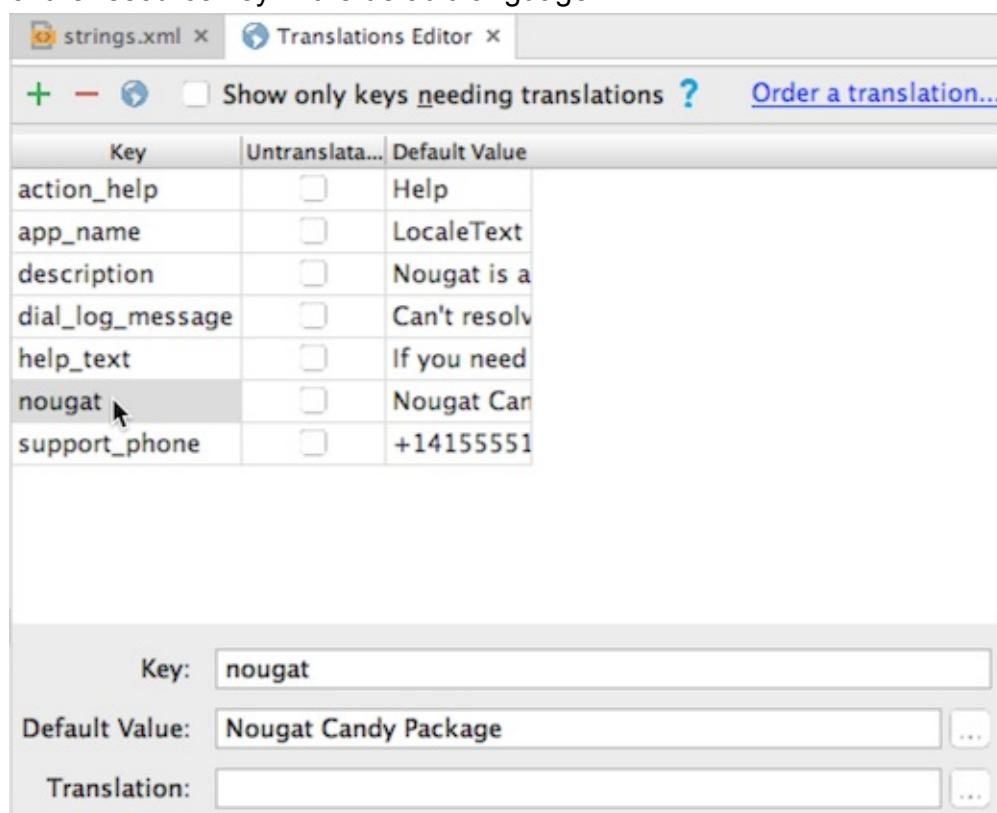
Include comments in the `strings.xml` file to describe the context in which each string is used. If you use a translator to translate your strings, the translator will understand the context from the comments you provide, which will result in a better quality translation.

## 1.3 Add another language resource to the app

Use the Android Studio Translations Editor to create the string resources for each language.

1. Open the `strings.xml` file, and click the **Open editor** link in the top right corner.

The **Translations Editor** appears with a row for each resource key (such as `action_help`, `app_name`, and `nougat`). The **Default Value** column is the default value of the resource key in the default language.



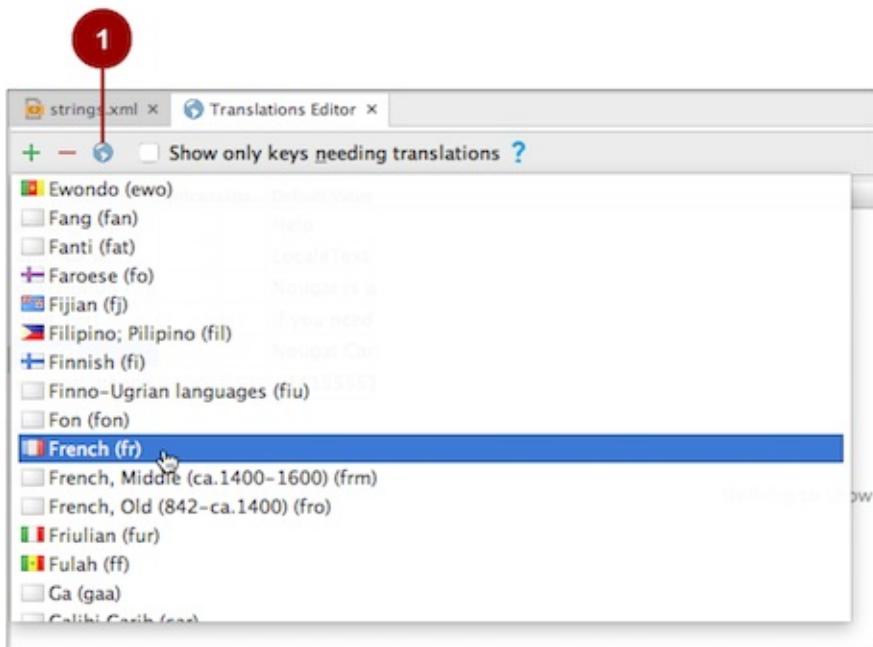
The screenshot shows the Android Studio Translations Editor interface. At the top, there are tabs for "strings.xml" and "Translations Editor". Below the tabs are buttons for adding (+), removing (-), and refreshing (refresh). There is also a checkbox for "Show only keys needing translations" and a link to "Order a translation...".

Key	Untranslated	Default Value
action_help	<input type="checkbox"/>	Help
app_name	<input type="checkbox"/>	LocaleText
description	<input type="checkbox"/>	Nougat is a
dial_log_message	<input type="checkbox"/>	Can't resolv
help_text	<input type="checkbox"/>	If you need
nougat	<input type="checkbox"/>	Nougat Can
support_phone	<input type="checkbox"/>	+14155551

At the bottom of the editor, there are three input fields:

- Key: nougat
- Default Value: Nougat Candy Package
- Translation: (empty)

2. Click the globe button in the top left corner of the **Translations Editor** pane ("1" in the following figure), and select **French** in the dropdown menu:



After you choose a language, a new column with blank entries appears in the **Translations Editor** for that language, and the keys that have not yet been translated appear in red.

To show only the keys that require translation, check the **Show only keys needing translation** checkbox option at the top of the **Translations Editor** pane, next to the globe.

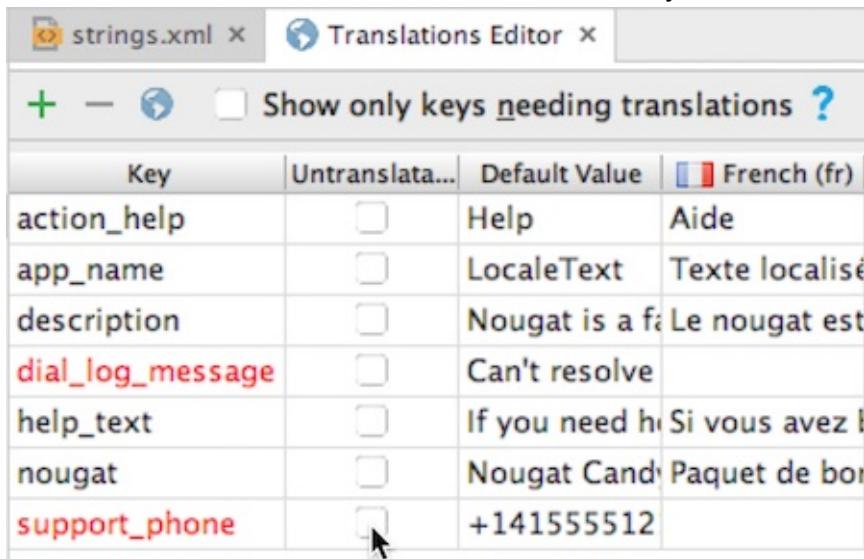
3. Select a key (such as `description`). The **Key**, **Default Value**, and **Translation** fields appear at the bottom of the **Translations Editor** pane.
4. Add the French translation for the key by selecting the key's cell in the column for the language (French), and entering the translation in the **Translation** field at the bottom of the pane as shown in the following figure.

Key:	<code>description</code>
Default Value:	<code>, and macadamia nuts are common, whipped egg whites, and sometimes chopped candied fruit. It is used in a variety of candy bars and chocolates.</code>
Translation:	<input type="text"/>

**Note:** For an app to be released to the public, you would use a translation service if you don't know the language. You can click the **Order a translation** link at the top of the **Translations Editor** pane to start the process with the service. The service should provide a translated `strings.xml` file, which you can copy into your project folder as described in the concept chapter, and the translations appear in the **Translations Editor**. For this example, if you don't know French use [Google Translate](#).

## 1.4 Enter the translations for the strings

- Enter the translations in the **French** column for each key, as shown in the following figure:



The screenshot shows the Translations Editor interface with a table of string keys and their translations. The columns are Key, Untranslatable (checkbox), Default Value, and French (fr). The rows include action\_help (Help/Aide), app\_name (LocaleText/Texte localisé), description (Nougat is a f/Le nougat est), dial\_log\_message (Can't resolve), help\_text (If you need h/Si vous avez b), nougat (Nougat Cand/Paquet de bo), and support\_phone (+141555512). The checkbox for support\_phone is checked.

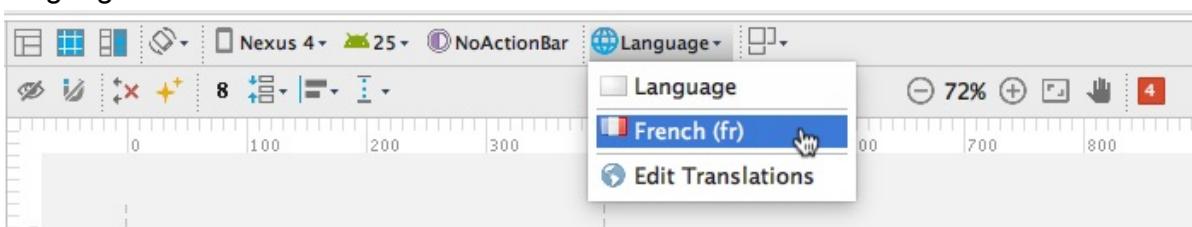
Key	Untranslatable	Default Value	French (fr)
action_help	<input type="checkbox"/>	Help	Aide
app_name	<input type="checkbox"/>	LocaleText	Texte localisé
description	<input type="checkbox"/>	Nougat is a f/Le nougat est	
dial_log_message	<input type="checkbox"/>	Can't resolve	
help_text	<input type="checkbox"/>	If you need h/Si vous avez b	
nougat	<input type="checkbox"/>	Nougat Cand/Paquet de bo	
support_phone	<input checked="" type="checkbox"/>	+141555512	

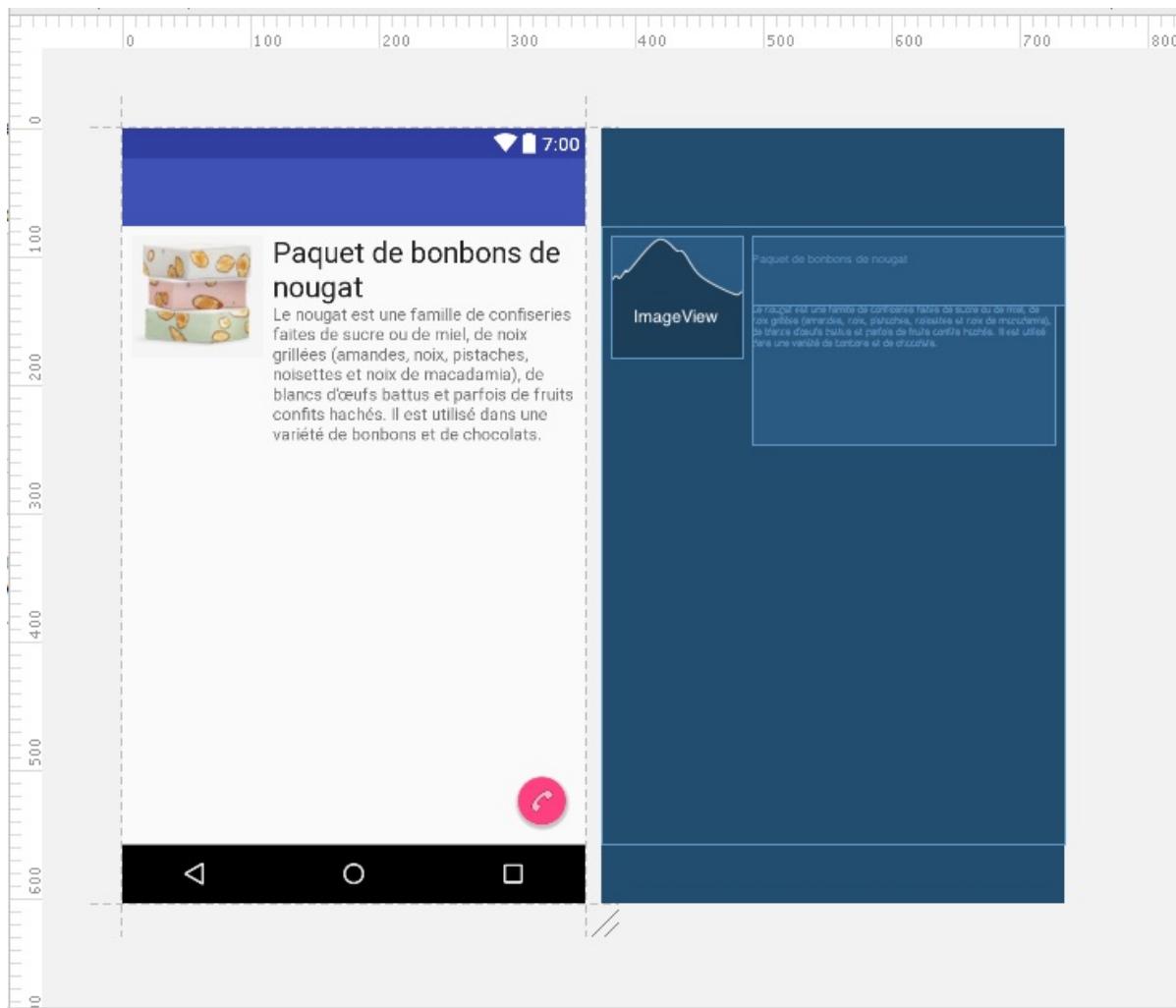
- Click the **Untranslatable** checkboxes for the `support_phone` and `dial_log_message` keys; they change from red to black. Keys shown in black don't need to be translated, either because they've already been translated, or because you've indicated that they are untranslatable.

Since there is no translation for the `support_phone` and `dial_log_message` keys, the default values will be used in the layout no matter what language and locale the user chooses in the Settings app.

If you don't want to translate the app name, you can click the **Untranslatable** checkbox for it; however, you can also modify the app name for each language, as shown in the above figure. The choice of translating the app name is up to you. Many apps add translated names so that the users in other languages understand the app's purpose and can find the app in Google Play in their own languages.

- To see a preview of the layout in the new language, open `content_main.xml` in the **Design** tab, and choose the language you added from the **Language** menu at the top of the Layout Editor. The layout reappears with a preview of what it looks like in that language.

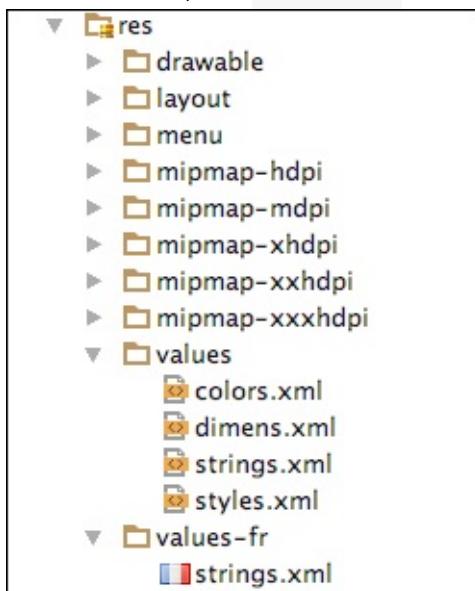




Changing the language can affect the layout. In this case, the headline in the previous figure, "Pacquet de bonbons de nougat," takes up two lines. To prepare for different languages, you need to create layouts that have enough room in their text fields to expand at least 30 percent, because a translated version of the text may take more room.

4. Switch to **Project: Project Files** view, and expand the **res** directory. You see that the Translations Editor created a new directory for the French language: `values-fr`. The `values` directory is the default directory for colors, dimensions, strings, and styles,

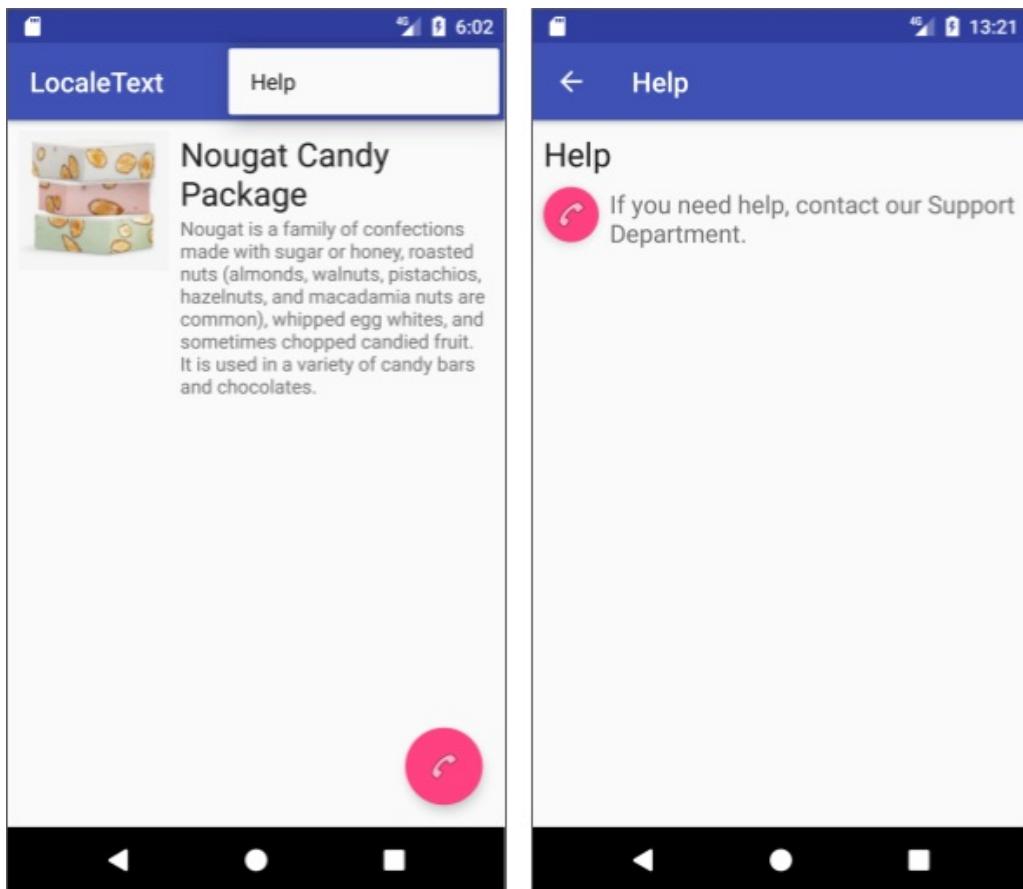
while for now, the `values-fr` directory has only string resources.



An app can include multiple `values` directories, each customized for a specific language and locale combination. When a user runs the app, Android automatically selects and loads the `values` directories that best match the user's chosen language and locale. For example, if the user chooses French as the language for the device in the Settings app, the French `strings.xml` file in the `values-fr` directory is used rather than the `strings.xml` file in the default `values` directory.

## 1.5 Run the app and switch languages

1. Run the app on the emulator or on a device. The strings appear in the language used in the default `strings.xml` file (in this case, English). Click the options menu, which offers one option (**Help**). Click **Help** to see the second `Activity`.



2. To switch the preferred language in your device or emulator, open the Settings app. If



your Android device is in another language, look for the gear icon: [Settings](#)

3. Find the **Languages & input** settings in the Settings app, and choose **Languages**.

Be sure to remember the globe icon for the **Languages & input** choice, so that you can find it again if you switch to a language you do not understand. **Languages** is the first choice on the **Languages & input** screen. 

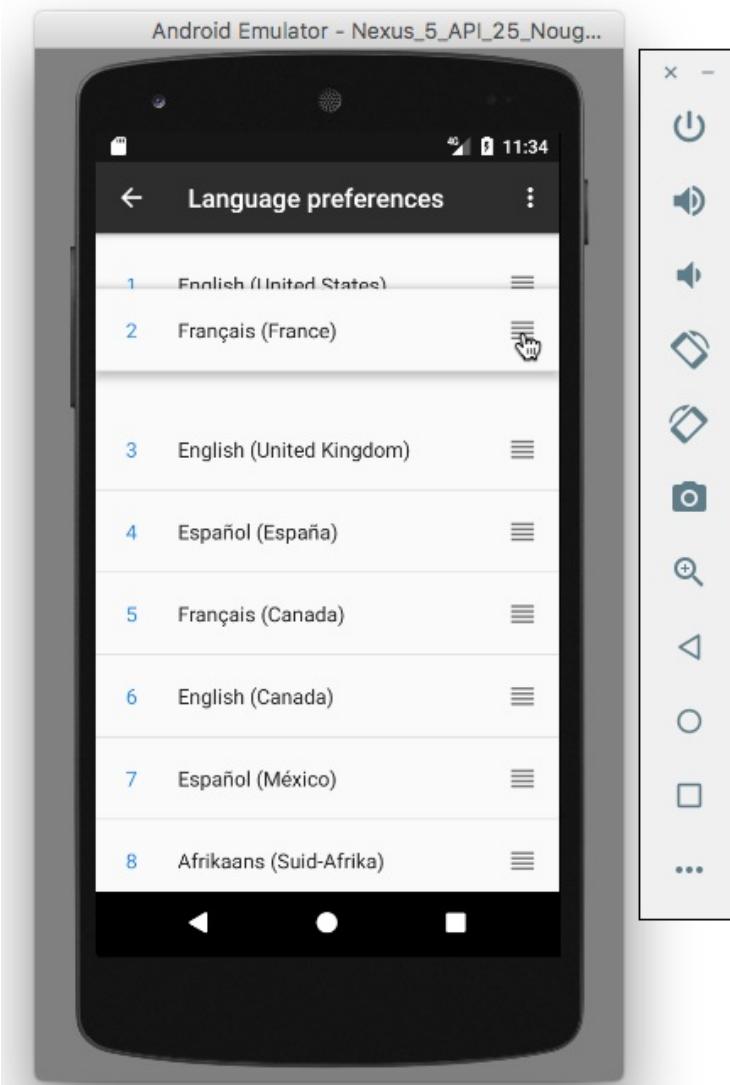
4. For devices and emulators running a version of Android previous to Android 7, choose **Language** on the **Languages & input** screen, select **Français (France)**, and skip the following steps.

(In versions of Android previous to Android 7, users can choose only one language. In Android 7 and newer versions, users can choose multiple languages and arrange them by preference. The primary language is numbered 1, as shown in the following figure, followed by lower-preference languages.)

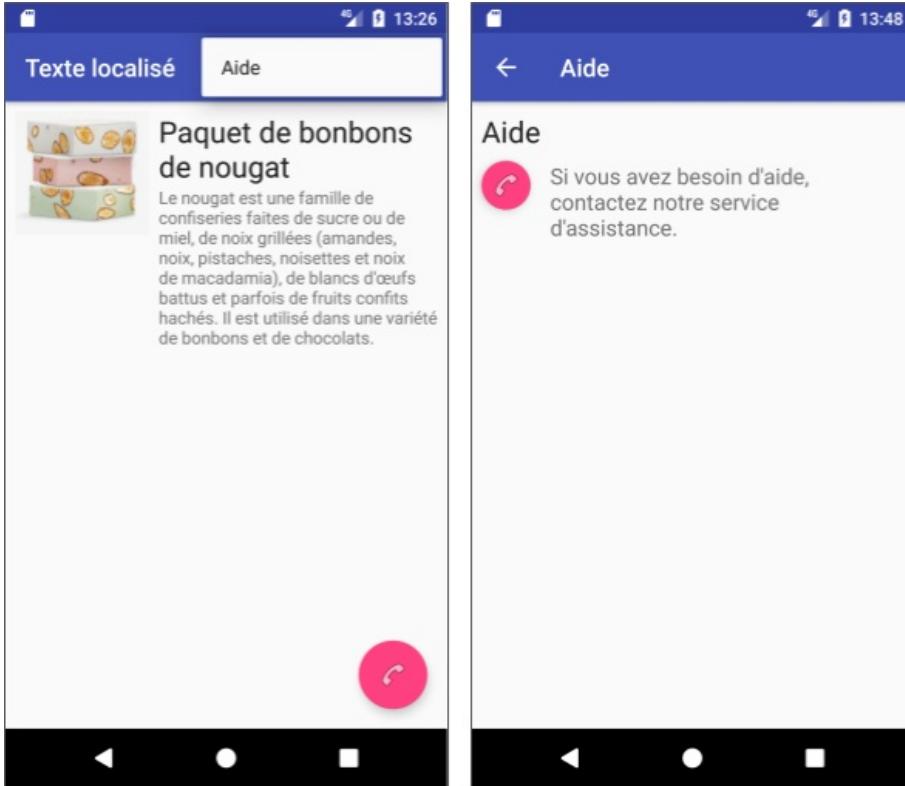
5. For devices and emulators running Android 7 or newer, follow these steps:

6. Choose **Languages** on the **Languages & input** screen.

7. To add a language not on the list, click **Add a language**, select **Français**, and then select **France** for the locale.
8. Use the move icon on the right side of the **Language preferences** screen to drag **Français (France)** to the top of the list.



9. Run the app again. The app now appears in French:



If the user has chosen a language that your app does not support, the app displays the strings that are in the default `strings.xml` file. For example, if the user has chosen Spanish (or any other language which your app does not support), the app displays strings in English, which is the language used in the default `strings.xml` file for this app.

**Note:** Remember to include all the strings you need for the app in the default `strings.xml` file. If the default file is absent, or if it is missing a string that the app needs, the app may stop.

## Task 2. Add a right-to-left (RTL) language

Some languages, such as English, are read from left to right (LTR), while others, such as Arabic or Hebrew, are read from right to left (RTL). While the text fields in an LTR language would be arranged in a layout from left to right, they should be reversed in a layout for an RTL language. Android 4.2 and newer versions provide [full native support for RTL layouts](#) so that you can use the same layout for both types of languages.

So far, the app uses two LTR languages: English (for the default) and French. In this task you add Hebrew, an RTL language.

### 2.1 Check for RTL support

Android provides the *layout mirroring* feature, which redraws the layout for RTL languages so that the image view, the text view, the options menu, and the floating action button are all automatically moved to the opposite side of the layout.

Open `AndroidManifest.xml` to check that the following attribute is part of the `<application>` element. This enables RTL layout mirroring:

```
android:supportsRtl="true"
```

Android Studio adds the above attribute to new projects. If you change `true` to `false`, RTL layout mirroring is not supported, and the views and other elements that use the RTL language appear as they would for an LTR language.

## 2.2 Add an RTL language

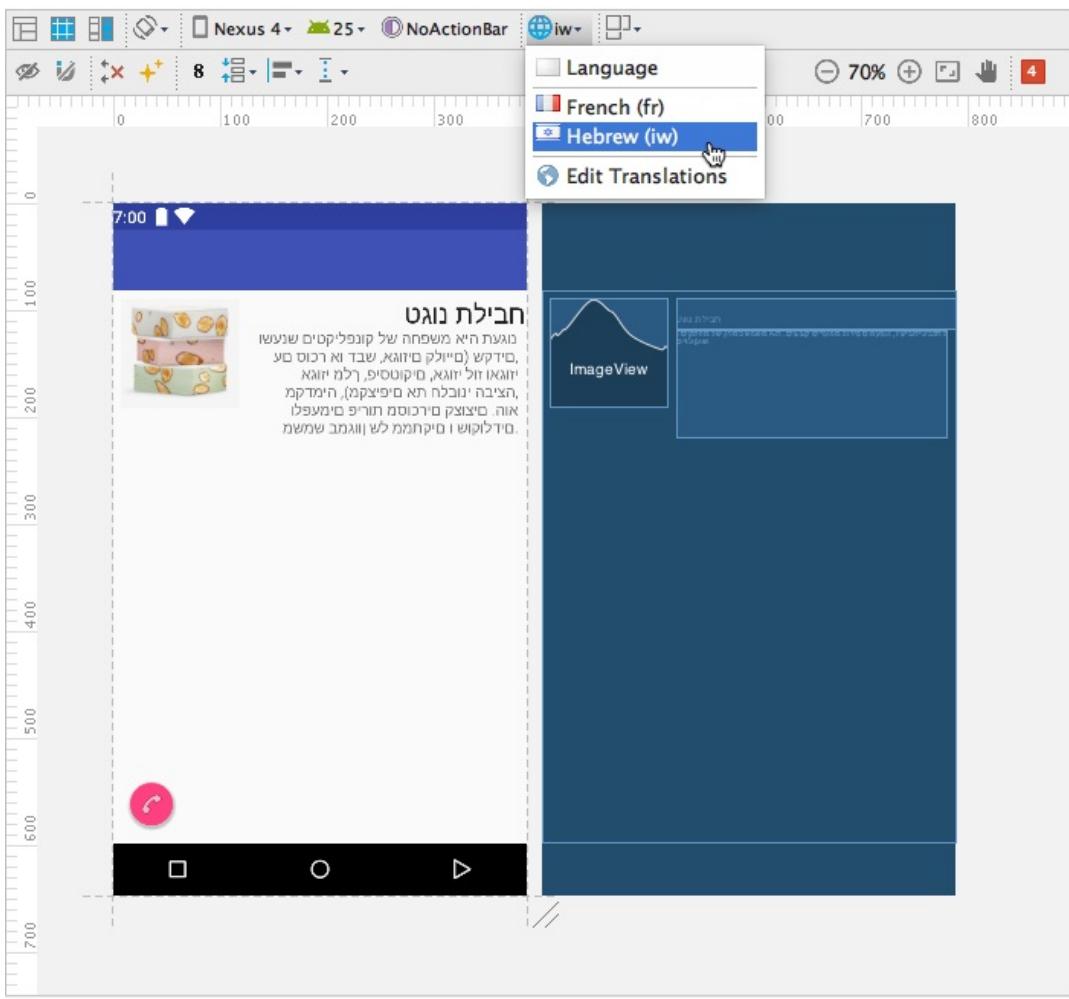
Add Hebrew to the LocaleText app:

1. Open the LocaleText app from the previous task in Android Studio.
2. Add Hebrew and translate the strings using the **Translations Editor** (see previous task for details). The translations appear in the new **Hebrew (iw)** column for the key.

The screenshot shows the Translations Editor interface with a table of string translations. The columns are Key, Untranslated..., Default Value, French (fr), and Hebrew (iw). The rows contain various app strings like 'action\_help', 'app\_name', etc., with their corresponding translations in French and Hebrew. A checkbox in the 'Untranslated...' column indicates if a translation is missing.

Key	Untranslated...	Default Value	French (fr)	Hebrew (iw)
action_help	<input type="checkbox"/>	Help	Aide	עזרה
app_name	<input type="checkbox"/>	LocaleText	Texte localisé	טקסט מוקומיז'
description	<input type="checkbox"/>	Nougat is a f...	Le nougat est ...	ים ושוקולדדים.
dial_log_message	<input checked="" type="checkbox"/>	Can't resolve		
help_text	<input type="checkbox"/>	If you need h...	Si vous avez ...	התמיכה שלנו.
nougat	<input type="checkbox"/>	Nougat Cand...	Paquet de bon	חבילת נוגט
support_phone	<input checked="" type="checkbox"/>	+141555512		

3. To see a preview of the layout in the new language, open `content_main.xml`, click the **Design** tab if it is not already selected, and choose **Hebrew (iw)** from the **Language** menu at the top of the Layout Editor. The layout reappears with a preview of what it will look like in Hebrew.



As shown in the figure, adding an RTL language can affect the layout, because Android Studio automatically supplies a declaration in the `<application>` element in the `AndroidManifest.xml` file that automatically turns on RTL layout support:

`android:supportsRtl="true"`. The RTL language characters are properly set from right to left, and the RTL text is justified to the right margin.

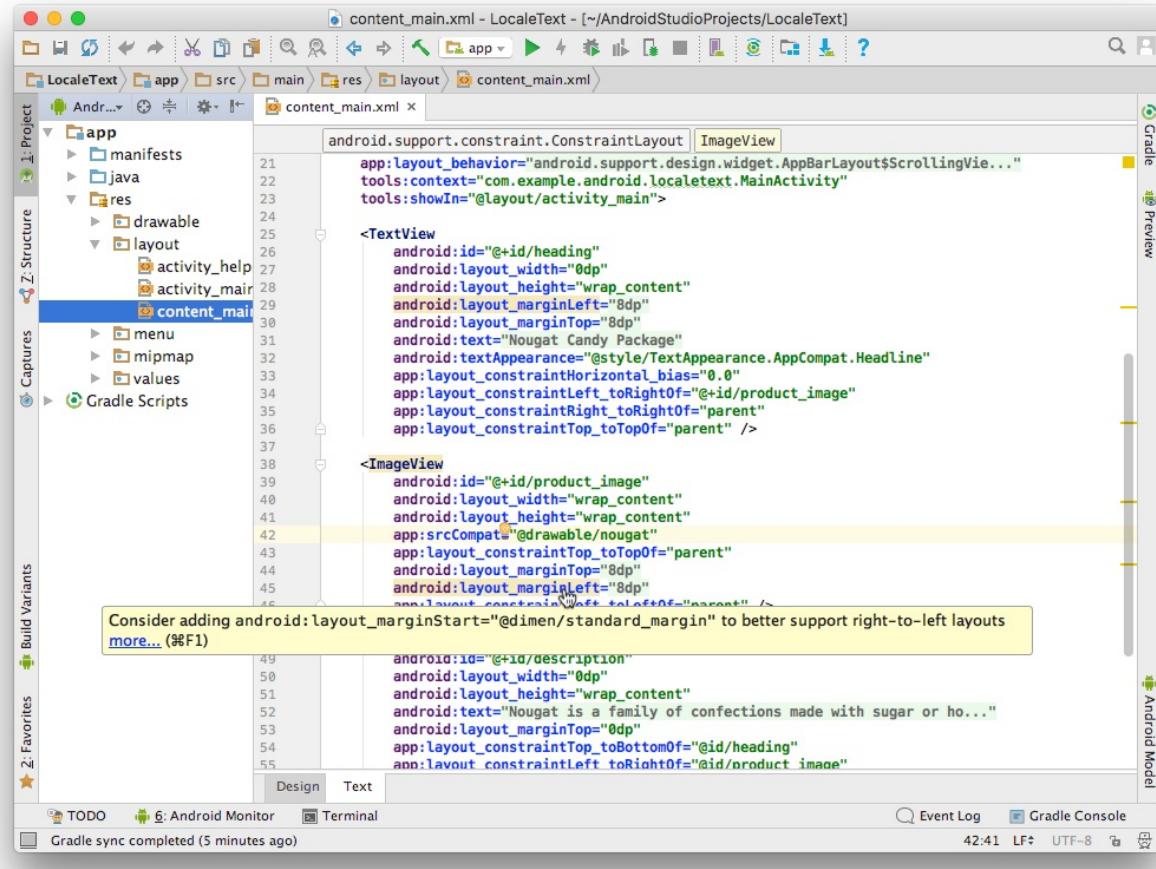
However, the UI elements—with the exception of the floating action button—are still in the same place in the layout, as if they were showing an LTR language. The floating action button moves to the opposite (left) side of the screen, as it should for an RTL language. But the `ImageView` is still constrained to the left margin, and the two `TextView` elements are constrained to the right side of the `ImageView`.

The `ImageView` and `TextView` elements should move to the opposite side of the screen, where a Hebrew reader would expect them to be. In the next step you will fix the layout to fully support an RTL language.

## 2.3 Modify the content layout for RTL languages

To adjust a layout so that it fully supports an RTL language, you need to add a "start" and "end" attribute for every "left" and "right" attribute in each layout. For example, if you use `android:layout_marginLeft`, add  `android:layout_marginStart :`

1. Open `content_main.xml` and look at the XML code.



Android Studio highlights "right" and "left" attributes (such as `layout_marginLeft` in the figure) and provides suggestions about including "start" and "end" attributes to better support right-to-left languages.

2. Add a "start" attribute for every "left" attribute, and an "end" attribute for every "right" attribute. For example, add the  `android:layout_marginStart` attribute to all uses of the `layout_marginLeft` attribute:

```

    android:layout_marginLeft="@dimen/standard_margin"
    android:layout_marginStart="@dimen/standard_margin"

```

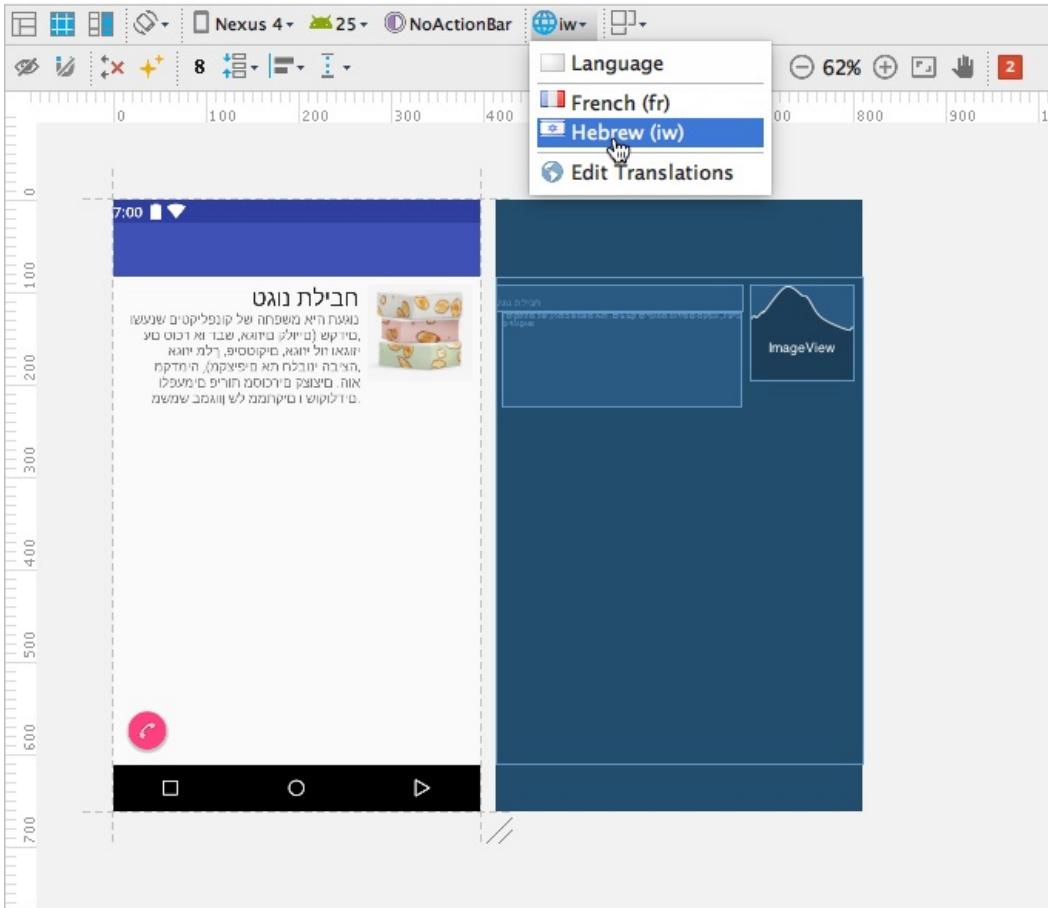
The "start" attribute defines the start of the view, which is the same as the "left" attribute for an LTR language; however, in RTL languages, the start side is the *right* side.

The "end" attribute defines the end of the view, which is the same as the "right" attribute for an LTR language; however, in RTL languages, the end side is the *left* side.

Be sure to add "start" and "end" to the constraint attributes. For example:

```
app:layout_constraintLeft_toRightOf="@+id/product_image"
app:layout_constraintStart_toEndOf="@+id/product_image"
```

- To see a preview of the layout in the new language, click the **Design** tab, and choose **Hebrew (iw)** in the **Language** menu at the top of the Layout Editor. The layout reappears with a preview of what it will look like, now that you have added the "start" and "end" attributes.



Compare this layout preview with the [layout preview in the previous task](#). As a result of adding the "start" and "end" attributes for margins and constraints, the layout now shows the image on the right side of the screen (the "start" side), and the `TextView` elements are constrained to its left ("end") side.

The following snippet is the revised XML code for the two `TextView` elements and the `ImageView` in `content_main.xml`:

```
ImageView in content_main.xml :
```

```
<TextView
    android:id="@+id/heading"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginLeft="@dimen/standard_margin"
    android:layout_marginStart="@dimen/standard_margin"
    android:layout_marginTop="@dimen/standard_margin"
    android:text="@string/nougat"
    android:textAppearance=
        "@style/TextAppearance.AppCompat.Headline"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintLeft_toRightOf="@+id/product_image"
    app:layout_constraintStart_toEndOf="@+id/product_image"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<ImageView
    android:id="@+id/product_image"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:srcCompat="@drawable/nougat"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_marginTop="@dimen/standard_margin"
    android:layout_marginLeft="@dimen/standard_margin"
    android:layout_marginStart="@dimen/standard_margin"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintStart_toStartOf="parent"/>

<TextView
    android:id="@+id/description"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="@string/description"
    android:layout_marginTop="0dp"
    app:layout_constraintTop_toBottomOf="@+id/heading"
    app:layout_constraintLeft_toRightOf="@+id/product_image"
    app:layout_constraintStart_toEndOf="@+id/product_image"
    android:layout_marginLeft="@dimen/standard_margin"
    android:layout_marginStart="@dimen/standard_margin"
    android:layout_marginRight="@dimen/standard_margin"
    android:layout_marginEnd="@dimen/standard_margin"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.0" >
</TextView>
```

## 2.4 Modify the help layout for RTL languages

Modify the `activity_help.xml` layout for the RTL language: Add a "start" attribute for every "left" attribute, and an "end" attribute for every "right" attribute, as done previously for `content.xml`, for the two `TextView` elements and the floating action button. The following snippet shows the XML code for these elements in `activity_help.xml`:

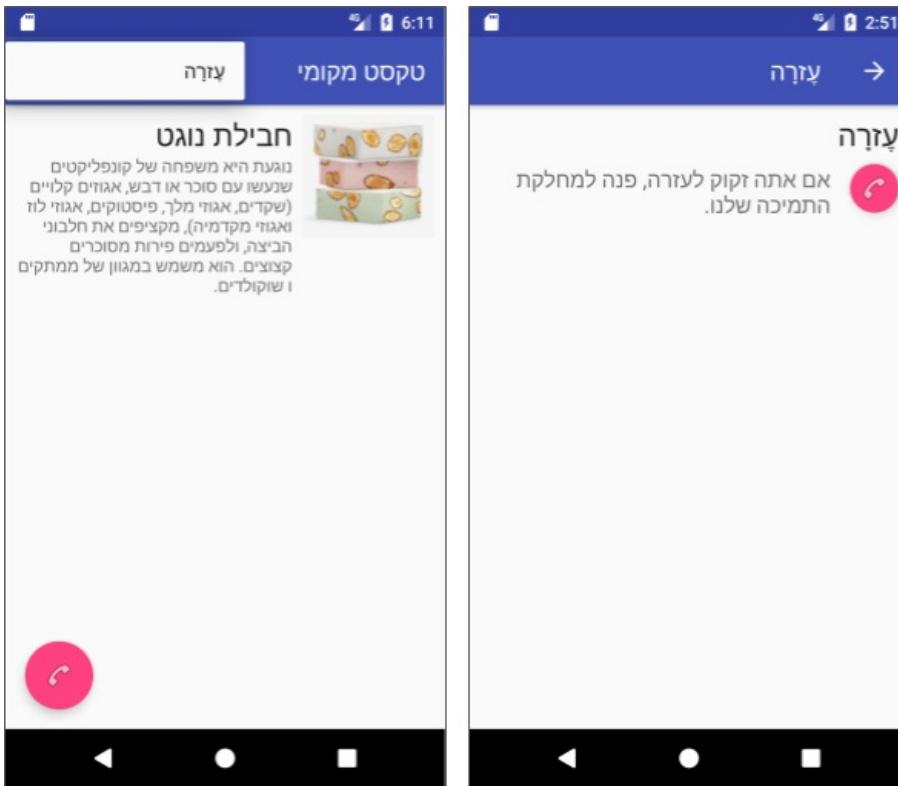
```
<TextView
    android:id="@+id/help_title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="@dimen/standard_margin"
    android:layout_marginStart="@dimen/standard_margin"
    android:layout_marginTop="@dimen/standard_margin"
    android:text="@string/action_help"
    android:textAppearance=
        "@style/TextAppearance.AppCompat.Headline"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:clickable="true"
    app:fabSize="mini"
    app:srcCompat="@android:drawable/ic_menu_call"
    android:layout_marginLeft="@dimen/standard_margin"
    android:layout_marginStart="@dimen/standard_margin"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    android:layout_marginTop="@dimen/standard_margin"
    app:layout_constraintTop_toBottomOf="@+id/help_title" />

<TextView
    android:id="@+id/help_body"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:padding="@dimen/standard_margin"
    android:text="@string/help_text"
    android:textAppearance="@style/TextAppearance.AppCompat.Medium"
    android:layout_marginLeft="@dimen/standard_margin"
    android:layout_marginStart="@dimen/standard_margin"
    android:layout_marginRight="@dimen/standard_margin"
    android:layout_marginEnd="@dimen/standard_margin"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/help_title"
    app:layout_constraintLeft_toRightOf="@+id/fab"
    app:layout_constraintStart_toEndOf="@+id/fab"
    app:layout_constraintHorizontal_bias="1.0" >
</TextView>
```

To see the changes, run the app on the device or emulator, and then change the language to Hebrew and run the app again.

The image now appears on the right of the screen, at the start of the text in the main activity. The help activity's layout is also mirrored for the RTL language.



Note also that the app name, the **Help** title for `HelpActivity`, and the **Up** button for `HelpActivity` (shown on the right side of the figure) are also moved to the opposite side of the layout, which is proper for RTL languages.

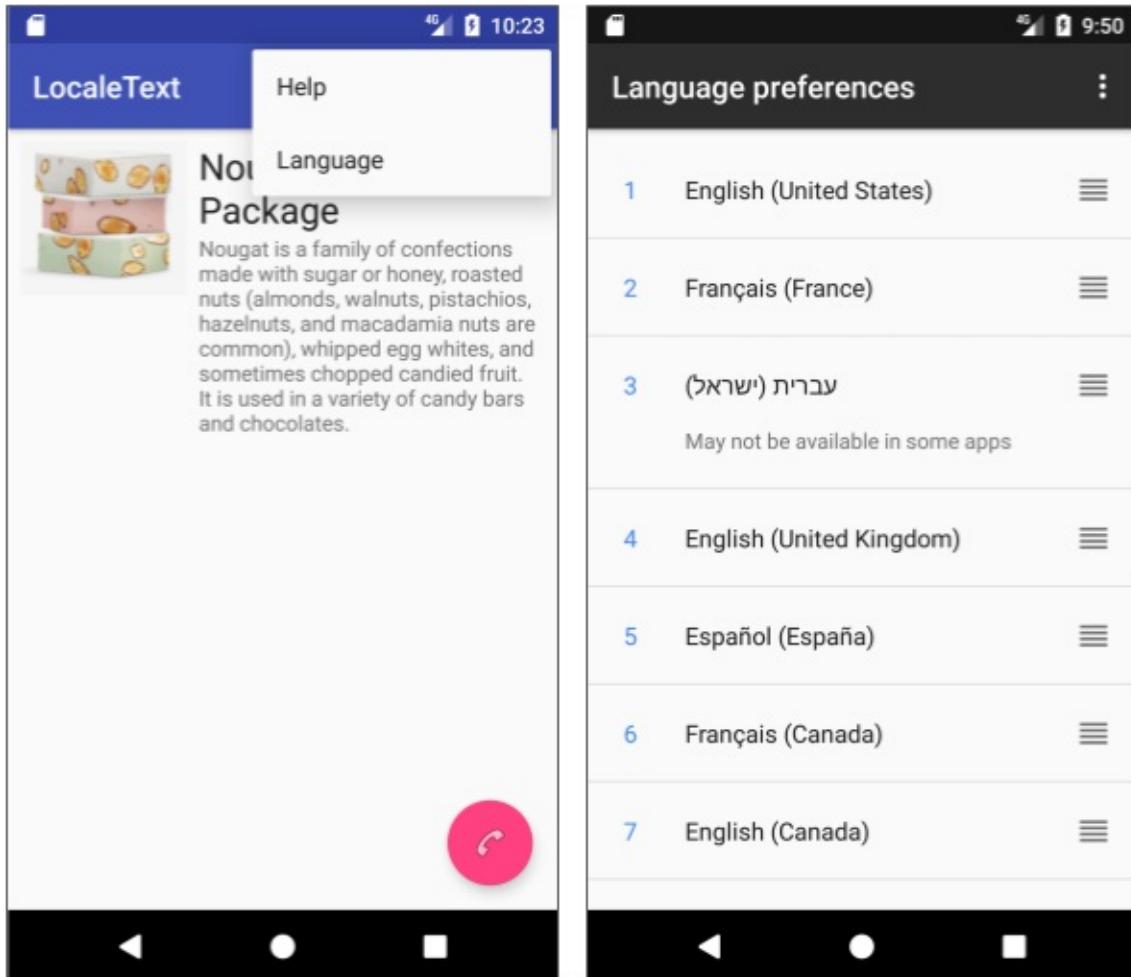
## Task 2 solution code

Android Studio project: [LocaleText1](#)

## Task 3. Add a Language option to the options menu

Would you like a quicker way than going back to the Settings app to change the language for testing this app? In this task, you add a **Language** option to the options menu in the LocaleText app. When tapped, this option uses an Intent to open the language list in the Settings app.

The **Language** menu option will make it easier for the user to open the language list without having to look for and launch the Settings app. The user can also immediately return to the LocaleText app and see the app in the new language by tapping the Back button.



### 3.1 Add the menu option

Open the LocaleText app project from the previous section (or download the [LocaleText1](#) app project).

Add the **Language** item to `menu_main.xml`, using `action_language` as the `id`. Set the `android:orderInCategory` attribute value to a higher number than `action_help` so that it appears below the **Help** option in the menu.

```
<item
    android:id="@+id/action_language"
    android:orderInCategory="110"
    android:title="Language"
    app:showAsAction="never" >
</item>
```

Extract the string resource for the option's `android:title` attribute ("Language"). Use the Translations Editor to provide that string resource for French and Hebrew.

## 3.2 Modify `onOptionsItemSelected()`

In `MainActivity`, change the code in the `onOptionsItemSelected()` method to a `switch` block that handles both **Help** and **Language**:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle options menu item clicks here.
    switch (item.getItemId()) {
        case R.id.action_help:
            showHelp();
            return true;
        case R.id.action_language:
            Intent languageIntent = new
                Intent(Settings.ACTION_LOCALE_SETTINGS);
            startActivity(languageIntent);
            return true;
        default:
            // Do nothing.
    }
    return super.onOptionsItemSelected(item);
}
```

The **Language** case uses an [Intent](#) with `Settings.ACTION_LOCALE_SETTINGS` to navigate directly to the language list.

## Task 3 solution code

Android Studio project: [LocaleText2](#)

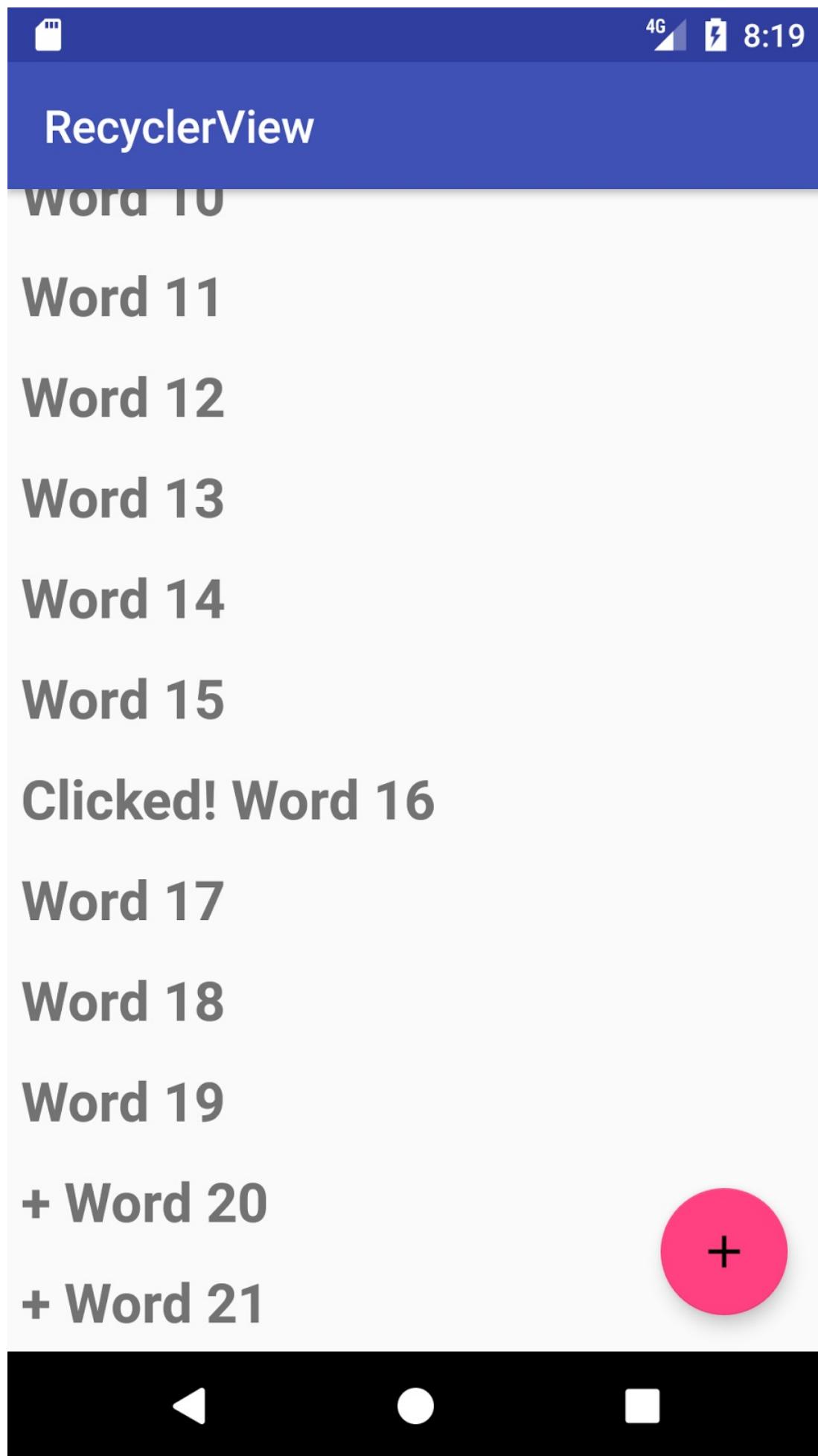
## Coding challenge

**Note:** All coding challenges are optional and not prerequisite for the material in the later chapters.

**Challenge:** Localize the UI elements of another app.

Developers often work in teams and are asked to update existing apps to localize them. In such cases you may not know everything about the code in the app, but you can apply localization best practices to update the code.

For this challenge, download the [RecyclerView\\_start](#) starter app, rename it to `RecyclerView`, and run the app. It shows a list of "Word" items. The floating action button adds a "Word" item each time you click it. Clicking on an item displays "Clicked!" with "Word" (as shown in the figure).



The following are the general steps for this challenge:

1. Add string resources to the project with translations for French (or any LTR language) and Hebrew (or any RTL language).

		Show only keys needing translations ?	<a href="#">Order a translation...</a>	
Key	Untranslated	Default Value	French (fr)	Hebrew (iw)
app_name	<input checked="" type="checkbox"/>	RecyclerView		
clicked	<input type="checkbox"/>	"Clicked!"	"cliquée"	"נלחץ"
word	<input type="checkbox"/>	Word %1\$d	mot %1\$d	"%1\$d מילה"

The challenge requires you to support a RTL language. The number that appears *after* "Word" for an LTR language ("Word 20") should appear *before* "Word" ("20 מילה") for an RTL language. To control placement, use a string format with an argument for the number, as described in the [Formatting and Styling](#) section of [String Resources](#).

2. Open `MainActivity`, find the code in the `onCreate()` method that puts the initial data into the word list (shown in the comment below), and replace it with code to use the `word` string resource:

```
// Original code: mWordList.addLast("Word " + i)
mWordList.addLast(String.format(getResources()
    .getString(R.string.word, i));
```

3. Within the `onCreate()` method, find the code for the floating action button in `onClick()` that adds a new word to the word list (shown in the comment below), and replace it to use the `word` string resource:

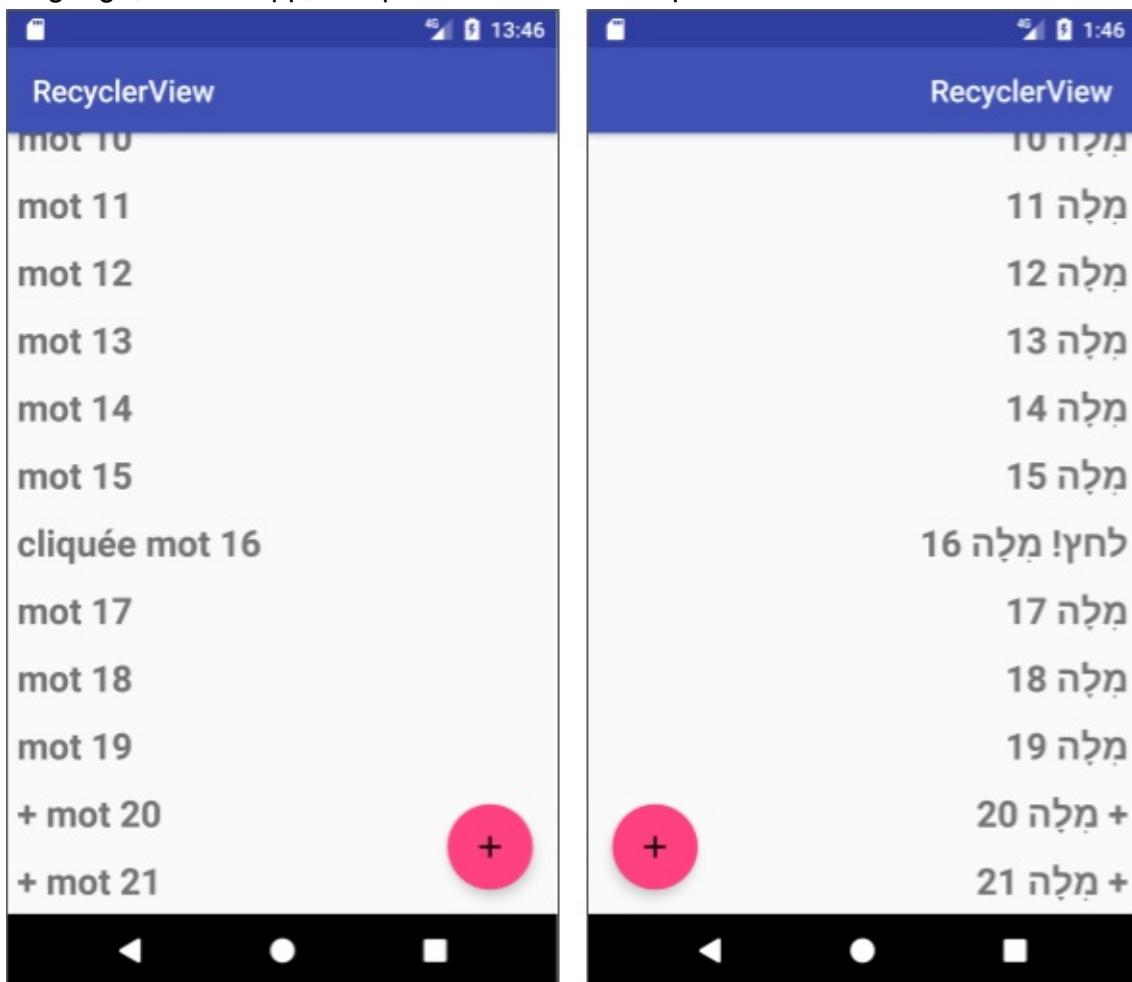
```
// Original code: mWordList.addLast("+ Word " + wordListSize)
mWordList.addLast("+ " +
    getResources().getString(R.string.word, wordListSize));
```

4. Open `WordListAdapter`, and use the `getString()` method to replace the hardcoded "Clicked!" with `R.string.clicked`. In the `WordViewHolder` class within `WordListAdapter`, change the `onClick()` method to the following:

```
public void onClick(View v) {
    String clickOutput = v.getContext().getString(clicked) +
        wordItemView.getText();
    wordItemView.setText(clickOutput);
}
```

5. Run the app and click the floating action button twice to add two more entries: "+ Word 20" and "+ Word 21." Click the **Word 16** item to ensure that it displays "Clicked! Word

16" as shown in the previous figure. Then change your device or emulator to another language, run the app, and perform the same steps.



## Challenge solution code

Android Studio project: [LocaleRecyclerView](#)

## Summary

- To prepare an app to support different languages, extract and save all strings as resources in `strings.xml`, including menu items, tabs, and any other navigation elements that use text.
- To use the Translations Editor, open the `strings.xml` file, and click the **Open editor** link in the top right corner.
- To add a language, click the globe button in the top left corner of the Translations Editor.
- To see a preview of the layout in the new language, open the layout XML file, click the **Design** tab, and choose the language in the **Language** menu at the top of the layout editor.

- To support right-to-left (RTL) languages with RTL layout mirroring, Android Studio automatically includes the `android:supportsRtl` attribute, set to `true`, as part of the `<application>` element in `AndroidManifest.xml` for each project.
- To support RTL languages in a layout, add the "start" and "end" XML attributes for all "left" and "right" attributes.
- Switch the preferred language in your device or emulator by choosing **Languages & input > Languages** in Settings. In Android 7 and newer you can add another language by clicking **Add a language**. Drag the preferred language to the top of the list.
- Use the default `strings.xml` file to define each and every string in the app, so that if a language or dialect is not available, the app shows the default language.

## Related concept

The related concept documentation is in [Languages and layouts](#).

## Learn more

Android developer documentation:

- [Supporting Different Languages and Cultures](#)
- [Localizing with Resources](#)
- [String Resources](#)
- [Localization checklist](#)
- [Language and Locale](#)
- [Testing for Default Resources](#)

Material Design: [Usability - Bidirectionality](#)

Android Developers Blog: [Native RTL support in Android 4.2](#)

Android Play Console: [Translate & localize your app](#)

Other:

- [Language Subtag Registry - IANA](#)
- [Country Codes](#)
- [ISO 3166 Country Codes](#)
- [Android locale codes and variants](#)

## 5.2: Using the locale to format information

### Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Use the locale's date format](#)
- [Task 2. Use the locale's number format](#)
- [Task 3. Use the locale's currency](#)
- [Solution code](#)
- [Coding challenge](#)
- [Challenge solution code](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

To provide the best experience for Android users in different regions, your app should handle not only text but also numbers, dates, times, and currencies in ways appropriate to those regions.

When users choose a language, they also choose a locale for that language, such as English (United States) or English (United Kingdom). Your app should change to show the formats for that locale: for dates, times, numbers, currencies, and similar information. Dates can appear in different formats (such as *dd / mm / yyyy* or *yyyy - mm - dd*), depending on the locale. Numbers appear with different punctuation, and currency formatting also varies.

### What you should already KNOW

You should be able to:

- Create and run apps in Android Studio.
- Add different languages and edit translations in the Translations Editor.
- Modify layouts to accommodate right-to-left (RTL) languages.

### What you will LEARN

You will learn how to:

- Use the current locale to set the format for the date and for numbers.
- Parse a number from a locale-formatted string.
- Show different currencies based on the locale.

## What you will DO

- Use `DateFormat` to format a date (a product's expiration date) according to the user-chosen locale.
- Use `NumberFormat` to format numbers and currencies according to the user-chosen locale.
- Use the current locale's country code to determine which currency to use.

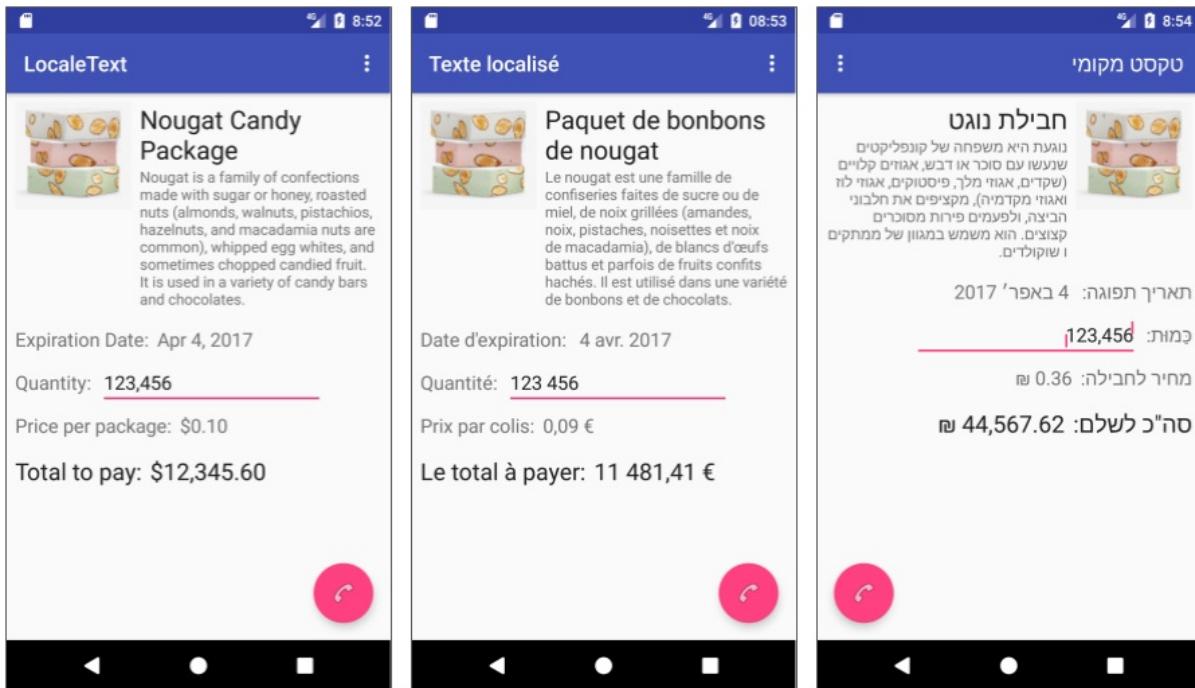
## App overview

The LocaleText app (from the lesson about using language resources) offers localized language resources, but there are other aspects of localization that need to be addressed. Dates, numbers, and currencies are the most important aspects.

You will use [LocaleText3\\_start](#) as the starter code. It already contains extra UI elements and strings translated into French and Hebrew, and the RTL layout adjustments ("start" and "end" attributes) that are described in the previous lesson. You will do the following:

- Convert a product expiration date to the format for the user's chosen locale.
- Format the user's quantity for the user's chosen locale, so that the quantity properly displays thousands and higher numbers with the right separators.
- Set the currency format for the language and locale, and use it to display the price.

The figure below shows the fully finished LocaleText3 app.



## Task 1. Use the locale's date format

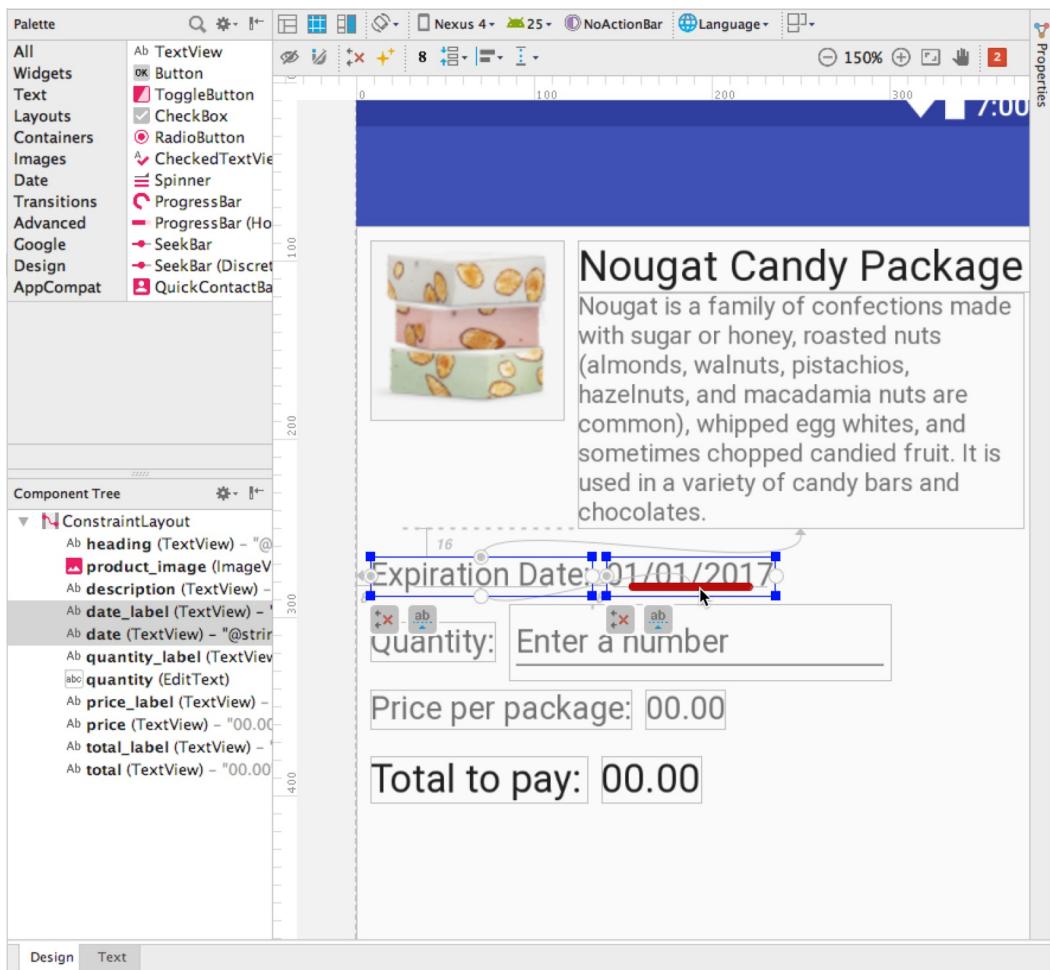
In this task you learn how to get the current `Date` and format it for current `Locale` using `DateFormat`. You add two views to the layout for the expiration date and its label, then localize the views by translating a string and adding RTL attributes to the layout.

### 1.1 Examine the layout

Download the `LocaleText3_start` app project, rename the project folder to `LocaleText3`, and open the project in Android Studio.

Open `content_main.xml` to see the layout and become familiar with the UI elements:

- `date_label` : Label for the expiration date.
- `date` : Expiration date.
- `quantity_label` : Label for the quantity.
- `quantity` : Quantity entered by the user.
- `price_label` : Label for the single-package price.
- `price` : Single-package price.
- `total_label` : Label for the total amount.
- `total` : Total amount, calculated by multiplying the `quantity` by the `price`.



Open the Translations Editor to see French and Hebrew translations for the new string resources. (For instructions on adding a language, see the lesson about using language resources.) You will not be translating the date, so the **Untranslatable** checkbox for the `date` key is already selected.

Since there is no translation for the `date` key, the default value will be used in the layout no matter what language and locale the user chooses. You will add code to format the date so that it appears in the locale's date format.

## 1.2 Use DateFormat to format the date

The code in `MainActivity.java` adds five days to the current date to get the expiration date. You will add code to format the expiration date for the locale.

**Note:** Throughout this lesson, places where you will add code are marked by `TODO` comments. After adding the code, you can delete or edit the `TODO` comments.

1. Open `MainActivity`, and find the the code at the end of the `onCreate()` method after the `fab.setOnClickListener()` section:

```
protected void onCreate(Bundle savedInstanceState) {  
    // ... Rest of the onCreate code.  
    fab.setOnClickListener(new View.OnClickListener() {  
        // ... Rest of the setOnClickListener code.  
    });  
    // Get the current date.  
    final Date myDate = new Date();  
    // Add 5 days in milliseconds to create the expiration date.  
    final long expirationDate = myDate.getTime() +  
        TimeUnit.DAYS.toMillis(5);  
    // Set the expiration date as the date to display.  
    myDate.setTime(expirationDate);  
    // TODO: Format the date for the locale.
```

This code adds five days to the current date to get the expiration date.

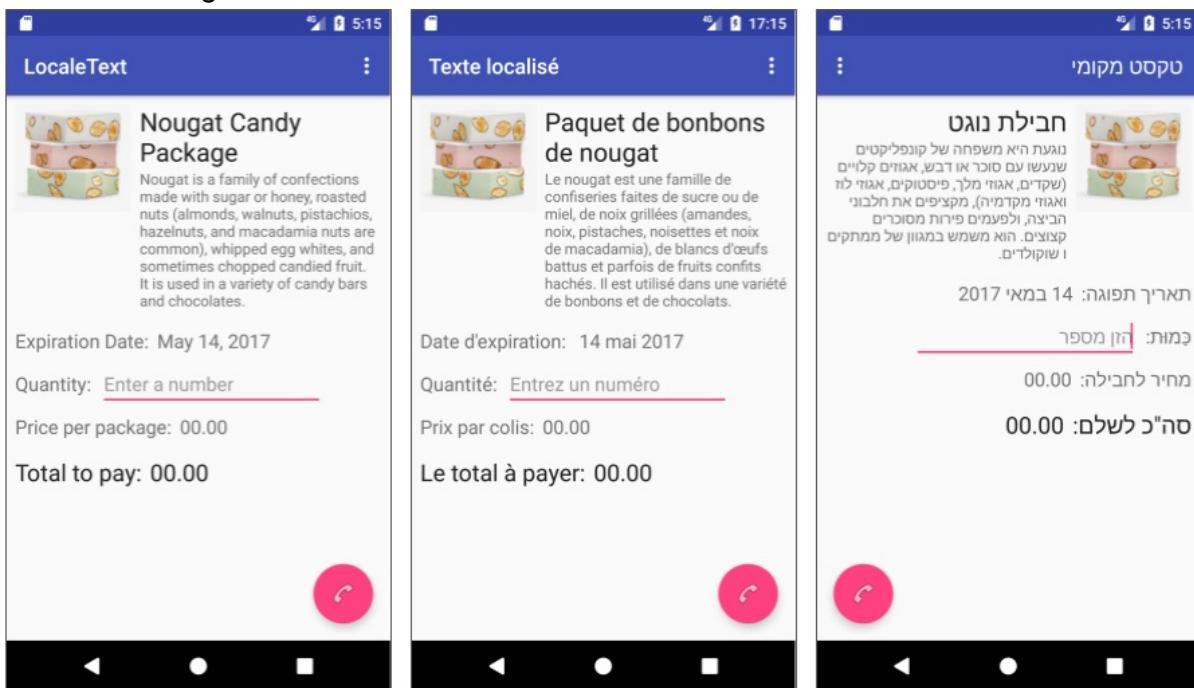
## 2. Add the following to format and display the date:

```
// TODO: Format the date for the locale.  
String myFormattedDate =  
    DateFormat.getDateInstance().format(myDate);  
// Display the formatted date.  
TextView expirationDateView = (TextView) findViewById(R.id.date);  
expirationDateView.setText(myFormattedDate);  
}
```

As you enter `DateFormat`, the `java.text.DateFormat` class should be imported. If it doesn't automatically import, click the red warning bulb in Android Studio and import it.

The `DateFormat.getDateInstance()` method gets the default formatting style for the user's selected language and locale. The `DateFormat.format()` method formats a date string.

3. Run the app, and switch languages. The date is formatted in each specific language as shown in the figure.



## Task 2. Use the locale's number format

Numbers appear with different punctuation in different locales. In U.S. English, the thousands separator is a comma, whereas in France, the thousands separator is a space, and in Spain the thousands separator is a period. The decimal separator also varies between period and comma.

Use the Java class `NumberFormat` to format numbers and to parse formatted strings to retrieve numbers. You will use the `quantity`, which is provided for entering an integer quantity amount.

`MainActivity` already includes `OnEditorActionListener`, which closes the keyboard when the user taps the **Done** key (the checkmark icon in a green circle):

You will add code to parse the quantity, convert it to a number, and then format the number according to the `Locale`.

### 2.1 Examine the listener code

Open `MainActivity`, and find the following listener code at the end of the `onCreate()` method. This is where you will put your code to get and format the quantity:

```
// Add an OnEditorActionListener to the EditText view.
enteredQuantity.setOnEditorActionListener(new
    EditText.OnEditorActionListener() {

    @Override
    public boolean onEditorAction(TextView v, int actionId,
                                  KeyEvent event) {
        if (actionId == EditorInfo.IME_ACTION_DONE) {
            // Close the keyboard.
            InputMethodManager imm = (InputMethodManager)
                v.getContext().getSystemService(
                    Context.INPUT_METHOD_SERVICE);

            imm.hideSoftInputFromWindow(v.getWindowToken(), 0);
            // TODO: Parse string in view v to a number.
        }
    }
});
```

The listener defines callbacks to be invoked when an action is performed on the editor. In this case, when the user taps the **Done** key, it triggers the `EditorInfo.IME_ACTION_DONE` action, and the callback closes the keyboard.

**Note:** The `EditorInfo.IME_ACTION_DONE` constant may not work for some smartphones that implement a proprietary soft keyboard that ignores `imeOptions` (as described in Stack Overflow tip, "[How do I handle ImeOptions' done button click?](#)"). For more information about setting input method actions, see [Specify the Input Method Action](#), and for details about the `EditorInfo` class, see [EditorInfo](#).

## 2.2 Use the locale's number format to parse the string to a number

The `quantity` value is a string, perhaps entered in a different language. In this step you write code to parse the string to a number, so that your code can use it in a calculation. If the value doesn't parse properly, you will throw an exception and show a message asking the user to enter a number.

Follow these steps:

1. Open `MainActivity`. At the top, declare `mNumberFormat` to get an instance of the number format for the user-chosen locale, and add a `TAG` for reporting an exception with the entered quantity:

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

After you enter `NumberFormat`, the expression appears in red. Click it and press Option-Return on a Mac, or Alt-Return on Windows, to choose `java.text.NumberFormat`.

The `NumberFormat.getInstance()` method returns a general-purpose number format for the user-selected language and locale.

2. In the listener code in `MainActivity`, change the `quantity` to a number (if the `view` is not empty) by using the `NumberFormat.parse()` method with `intValue()` to return an integer.

```
// Parse string in view v to a number.  
mInputQuantity = mNumberFormat.parse(v.getText()  
        .toString()).intValue();  
// TODO: Convert to string using locale's number format.
```

3. Android Studio displays a red bulb in the left margin of the `numberFormat.parse` statement because the statement requires an exception handler. Although the keyboard is restricted to a numeric keypad, the code still needs to handle an exception when parsing the string to convert it to a number. Click the bulb and choose **Surround with try/catch** to create a simple `try` and `catch` block to handle exceptions. Android Studio automatically imports `java.text.ParseException`.

```
// Parse string in view v to a number.  
try {  
    // Use the number format for the locale.  
    mInputQuantity = mNumberFormat.parse(v.getText()  
        .toString()).intValue();  
} catch (ParseException e) {  
    e.printStackTrace();  
}  
// TODO: Convert to string using locale's number format.
```

4. The exception handling is not yet finished. The best practice is to display a message to the user. The `TextEdit setError()` method provides a popup warning if, for some reason, a number was not entered. Change the code in the previous step to the following, using the string resource `enter_number` (provided in the `strings.xml` file and previously translated).

```
// Parse string in view v to a number.  
try {  
    // Use the number format for the locale.  
    mInputQuantity = mNumberFormat.parse(v.getText()  
        .toString()).intValue();  
    v.setError(null);  
} catch (ParseException e) {  
    Log.e(TAG, Log.getStackTraceString(e));  
    v.setError(getText(R.string.enter_number));  
    return false;  
}  
// TODO: Convert to string using locale's number format.
```

If the user runs the app and taps the **Done** key without entering a number, the `enter_number` message appears. Since this is a string resource, the translated version appears in French or Hebrew if the user chooses French or Hebrew for the device's language.

## 2.3 Convert the number to a string formatted for the locale

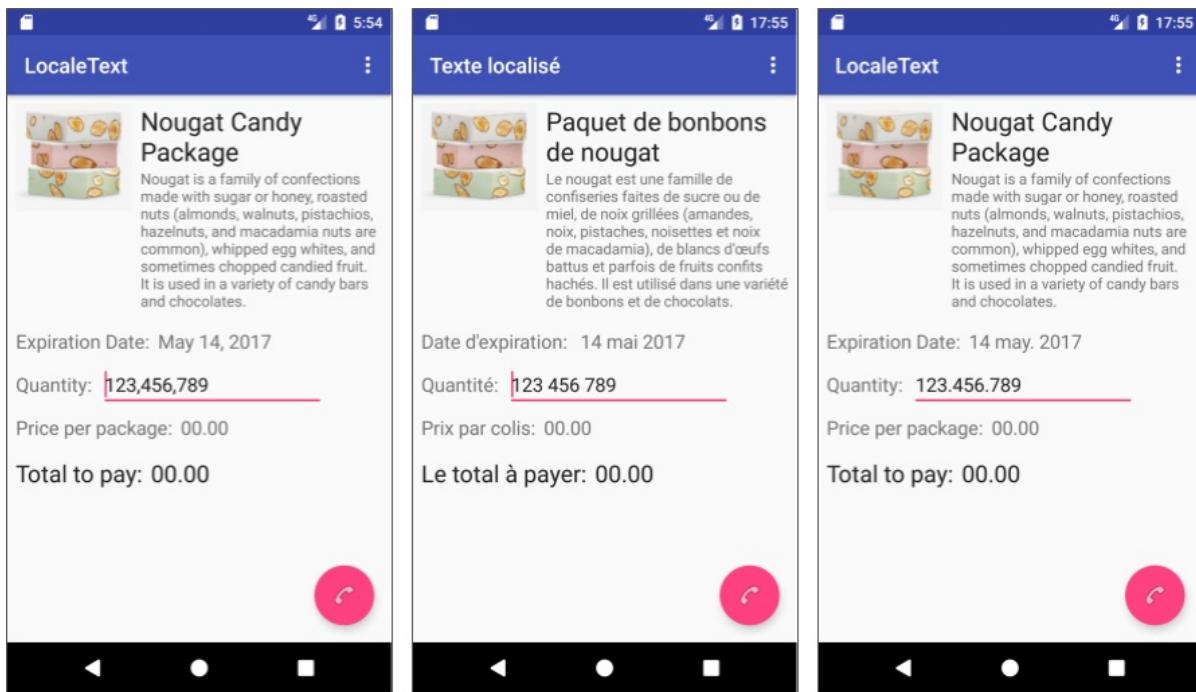
In this step you convert the number back into a string that is formatted correctly for the current locale.

In the listener code in `MainActivity`, add the following to convert the number to a string using the format for the current locale, and show the string:

```
// Convert to string using locale's number format.  
String myFormattedQuantity = mNumberFormat.format(mInputQuantity);  
// Show the locale-formatted quantity.  
v.setText(myFormattedQuantity);
```

Run the app:

1. Enter a quantity and tap the **Done** key to close the keyboard.
2. Switch the language. First choose **English (United States)**, run the app again, and enter a quantity again. The thousands separator is a comma.
3. Switch the language to **Français (France)**, run the app again, and enter a quantity again. The thousands separator is a space.
4. Switch the language to **Español (España)**, run the app again, and enter a quantity again. The thousands separator is a period. Note that even though you have no Spanish string resources (which is why the text appears in English), the date and the number are both formatted for the Spain locale.



## Task 3. Use the locale's currency

Currencies are different in some locales. Use the `NumberFormat` class to format currency numbers. The `NumberFormat.getCurrencyInstance()` method returns the currency format for the user-selected language and `Locale`, and the `NumberFormat.format()` method applies the format to create a string.

To demonstrate how to show amounts in a currency format for the user's chosen locale, you will add code that will show the price in the locale's currency. Given the complexities of multiple currencies and daily fluctuations in exchange rates, you may want to limit the currencies in an app to specific locales. To keep this example simple, you will use a fixed exchange rate for just the France and Israel locales, based on the U.S. dollar. The starter app already includes fixed (fake) exchange rates.

### 3.1 Get the current locale and its country code

You can retrieve the country code of the user-chosen locale to determine whether the country is one whose currency your app supports (that is, France or Israel). If it is, use the locale's currency format and exchange rate. For all other unsupported countries and the U.S., set the currency format to U.S. (dollar).

The starter app already includes variables for the France and Israel currency exchange rates. You will use this to calculate and show the price. Pricing information would likely come from a database or a web service, but to keep this app simple and focused on localization, a fixed price in U.S. dollars has already been added to the starter app.

1. Open `MainActivity` and find the fixed price and exchange rates at the top of the class:

```
// Fixed price in U.S. dollars and cents: ten cents.  
private double mPrice = 0.10;  
// Exchange rates for France (FR) and Israel (IL).  
private double mFrExchangeRate = 0.93; // 0.93 euros = $1.  
private double mIlExchangeRate = 3.61; // 3.61 new shekels = $1.
```

2. Add the following to the top of `MainActivity` to get an instance (`mCurrencyFormat`) of the currency for the user's chosen locale:

```
// Get locale's currency.  
private NumberFormat mCurrencyFormat =  
    NumberFormat.getCurrencyInstance();
```

The `NumberFormat.getCurrencyInstance()` method returns the currency format for the user-selected locale.

## 3.2 Calculate and show the price in different currencies

1. To calculate the price at a specific exchange rate, open `MainActivity` and add the following code to the `onCreate()` method after the `TODO` comment:

```
// TODO: Set up the price and currency format.  
String myFormattedPrice;  
String deviceLocale = Locale.getDefault().getCountry();  
// If country code is France or Israel, calculate price  
// with exchange rate and change to the country's currency format.  
if (deviceLocale.equals("FR") || deviceLocale.equals("IL")) {  
    if (deviceLocale.equals("FR")) {  
        // Calculate mPrice in euros.  
        mPrice *= mFrExchangeRate;  
    } else {  
        // Calculate mPrice in new shekels.  
        mPrice *= mIlExchangeRate;  
    }  
    // Use the user-chosen locale's currency format, which  
    // is either France or Israel.  
    myFormattedPrice = mCurrencyFormat.format(mPrice);  
} else {  
    // mPrice is the same (based on U.S. dollar).  
    // Use the currency format for the U.S.  
    mCurrencyFormat = NumberFormat.getCurrencyInstance(Locale.US);  
    myFormattedPrice = mCurrencyFormat.format(mPrice);  
}  
// TODO: Show the price string.
```

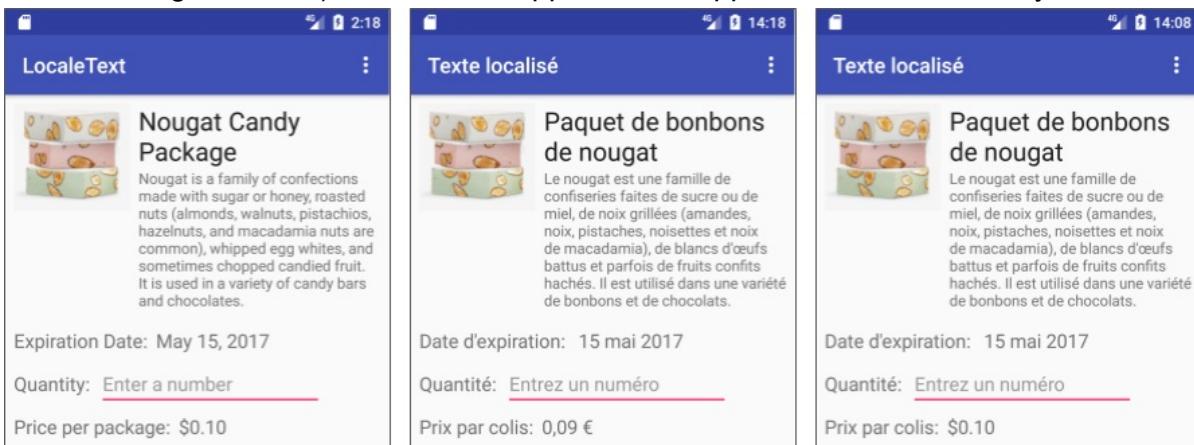
The `Locale.getDefault()` method gets the current value of the `Locale`, and the `Locale.getCountry()` method returns the country/region code for this `Locale`. Used together, they provide the country code that you need to check. The code `FR` is for France, and `IL` is for Israel. The code tests only for those locales, because all other locales, including the U.S., use the default currency (U.S. dollars).

The code then uses the currency format to create the string `myFormattedPrice`.

- After the above code, add code to show the price string:

```
// TODO: Show the price string.
TextView localePrice = (TextView) findViewById(R.id.price);
localePrice.setText(myFormattedPrice);
```

- Run the app. The "Price per package" appears in U.S. dollars (left side of the figure below) because the device or emulator is set to **English (United States)**.
- Change the language and locale to **Français (France)**, and navigate back to the app. The price appears in euros (center of the figure below).
- Change the language to **Français (Canada)**, and the price appears in U.S. dollars (right side of the figure below) because the app doesn't support Canadian currency.



When the user chooses the **Français (Canada)** locale, the language changes to French because French language resource strings are provided. The locale, however, is Canada. Since Canadian currency is not supported, the code uses the default locale's currency, which is the United States dollar. This demonstrates how the language and the locale can be treated differently.

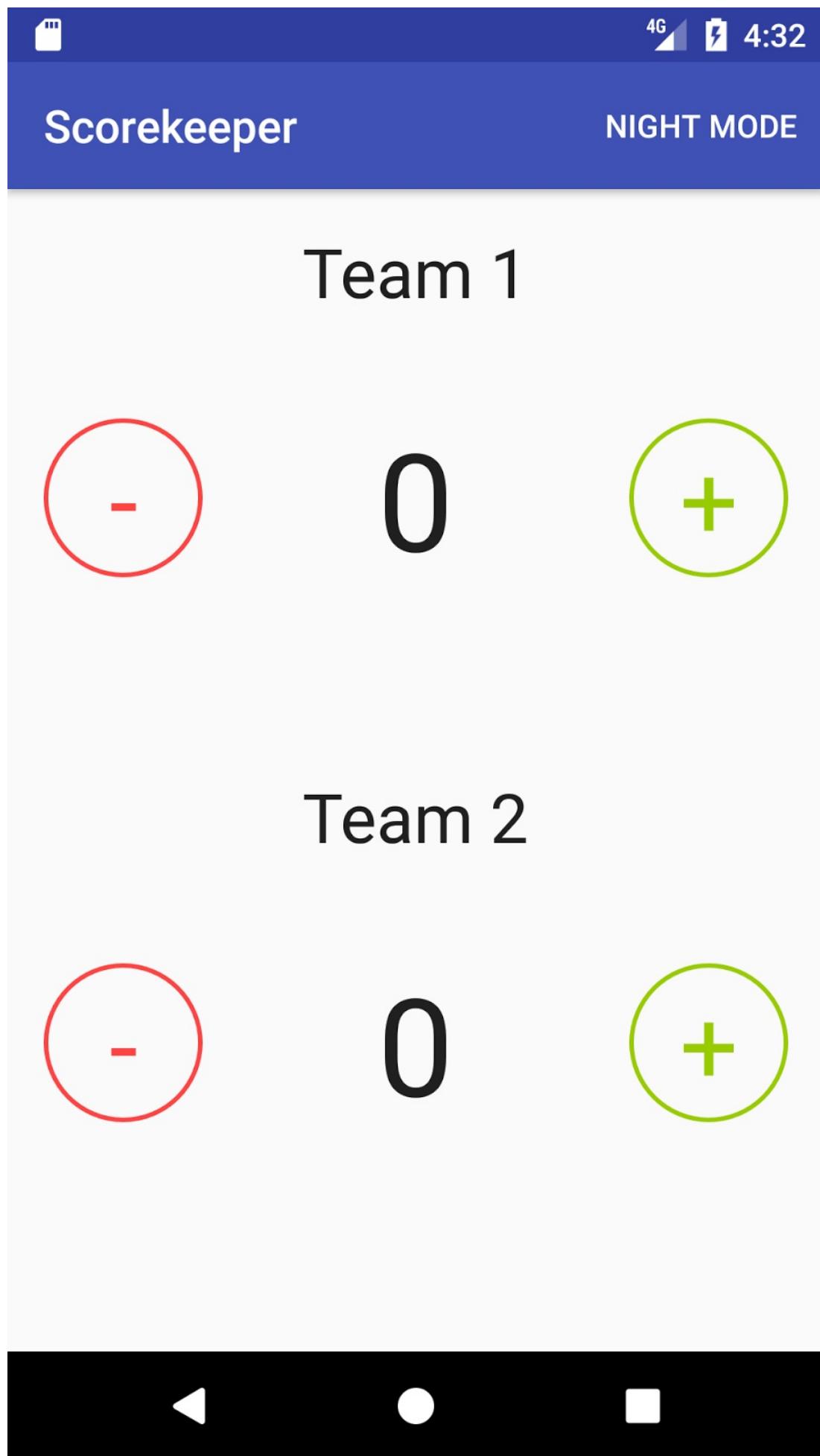
## Solution code

Android Studio project: [LocaleText3](#)

## Coding challenge

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

**Challenge:** This challenge demonstrates how to change colors and text styles based on the locale. Download the [Scorekeeper\\_start](#) app (shown below), and rename and refactor the project to `ScorekeepLocale`.



The Locales concept chapter explains how to change colors and styles for different locales.

In the Scorekeeper app, the text size is controlled by a style in `styles.xml` in the `values` directory, and the color for a `Button` background is set in an XML file in the `drawable` directory. An app can include multiple resource directories, each customized for a different language and locale. Android picks the appropriate resource directory depending on the user's choice for language and locale. For example, if the user chooses French, and French has been added to the app as a language using the Translations Editor (which creates the `values-fr` directory to hold it), then:

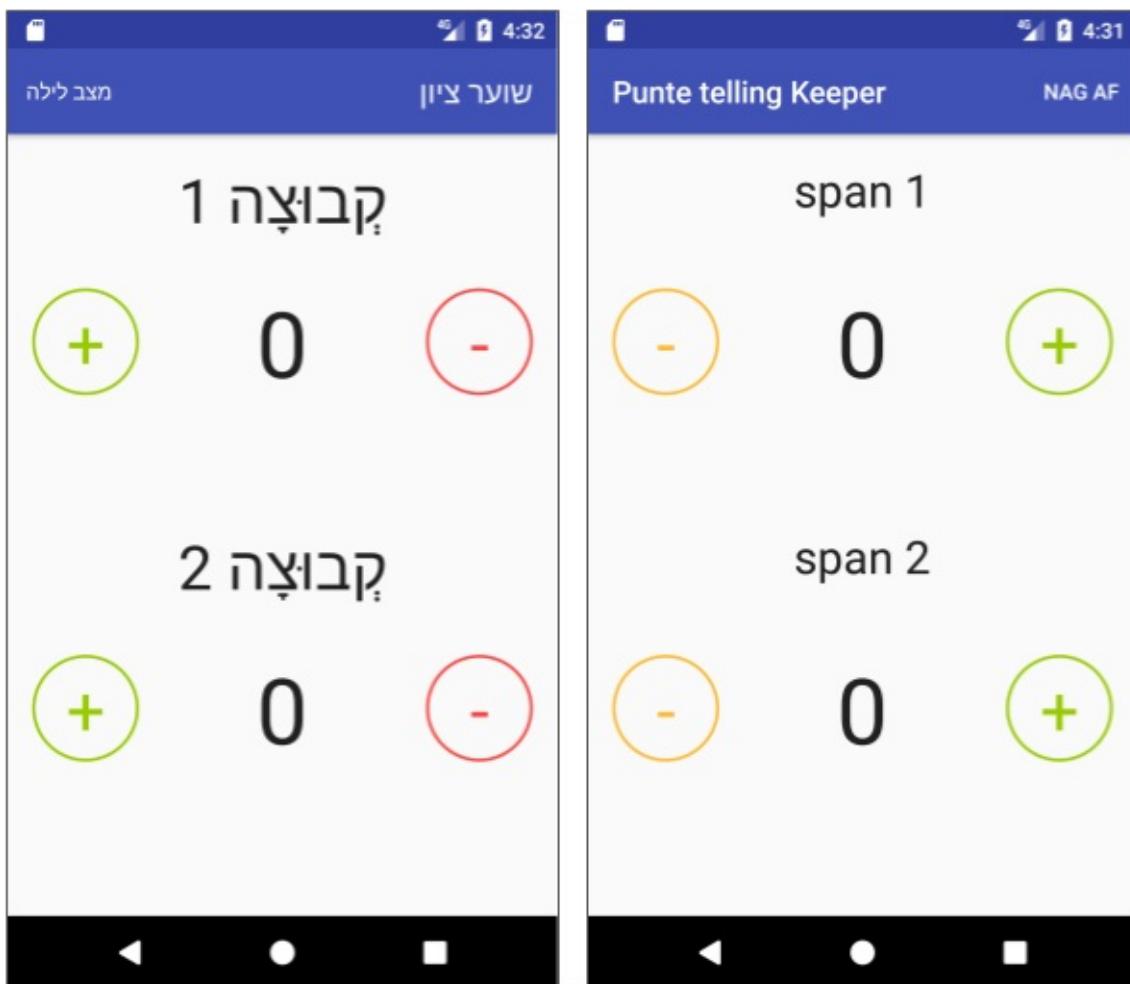
- The `strings.xml` file in `values-fr` is used rather than the `strings.xml` file in the default `values` directory, as you would expect.
- The `colors.xml` and `dimens.xml` files in `values-fr` (if they have been added to the directory) are used rather than the versions in the default `values` directory.
- The `drawables` in `drawable-fr` (if this directory has been added to the app) are used rather than the versions in the default `drawable` directory.

For this challenge, do the following:

- Add the Afrikaans and Hebrew languages.
- Change the text size for the team names to `40sp` in Hebrew only (as shown on the left side of the figure below).
- In South Africa, red is one of the colors of mourning, and is therefore not appropriate as the color for the minus buttons. Change the minus button color to light orange, but only for the Afrikaans language in the South Africa locale (as shown on the right side of the figure below).

#### Hints:

- The size is controlled in the `TeamText` style in `styles.xml` in the `values` directory. Add a new `styles.xml` file to the `values-iw` directory and edit it.
- The color for the `Button` background (the circle around the minus sign) is set in `minus_button_background.xml` in the `res>drawable` directory. To create a different language version, copy the `drawable` subdirectory and rename the copy **drawable-af-rZA**. You can then change the `minus_button_background.xml` file in the copied directory to change the color.
- The minus sign color is set in the `MinusButtons` style in `styles.xml`.
- Use `"@android:color/holo_orange_light"` for the new color.



## Challenge solution code

Android Studio project: [ScorekeepLocale](#)

## Summary

To format a date for current locale, use `DateFormat`.

- The `DateFormat getDateInstance()` method gets the default formatting style for the user's selected language and locale.
- The `DateFormat format()` method formats a date string.

Use the `NumberFormat` class to format numbers and to parse formatted strings to retrieve numbers.

- The `NumberFormat getInstance()` method returns a general-purpose number format for the user-selected language and locale.
- Use the `NumberFormat parse()` method to get an integer from a formatted number

string.

Use the `NumberFormat` class to format currency numbers.

- The `NumberFormat` `getCurrencyInstance()` method returns the currency format for the user-selected language and locale.
- The `NumberFormat` `format()` method applies the format to create a string.

Use the `Locale`:

- Get the current value of the default `Locale` with `Locale.getDefault()`.
- Get the country/region code by specifying the current `Locale` with `Locale.getCountry()`.

Use resource directories:

- Resources are defined within specially named directories inside the project's res directory. The path is `project_name/app/src/main/res/`.
- Add resource directories using the following general format:

```
<resource type>-<language code>[-r<country code>]
```

`<resource type>` : The resource subdirectory, such as `values` or `drawable`.

`<language code>` : The language code, such as `en` for English or `fr` for French.

`<country code>` : Optional: The country code, such as `us` for the U.S. or `FR` for France.

## Related concept

The related concept documentation is [Locales](#).

## Learn more

Android developer documentation:

- [Supporting Different Languages and Cultures](#)
- [Localizing with Resources](#)
- [Localization checklist](#)
- [Language and Locale](#)
- [Testing for Default Resources](#)

Material Design: [Usability - Bidirectionality](#)

Android Developers Blog:

- [Android Design Support Library](#)
- [Native RTL support in Android 4.2](#)

Android Play Console: [Translate & localize your app](#)

Other:

- [Language Subtag Registry - IANA](#)
- [Country Codes](#)
- [ISO 3166 Country Codes](#)
- [Android Locale Codes and Variants](#)
- [How do I use NumberFormat to format currencies?](#)
- [How to format international telephone numbers](#)
- [International Country Calling Codes](#)
- [ISO-3166-1 Standard \(Wikipedia\)](#)

# 6.1: Exploring accessibility in Android

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Explore TalkBack and text-to-speech](#)
- [Task 2. Explore other settings](#)
- [Solution code](#)
- [Coding challenge](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

Android apps should be usable by everyone, including people with disabilities.

Common disabilities that can affect a person's use of an Android device include blindness, low vision, color blindness, deafness or hearing loss, and restricted motor skills. When you develop your apps with accessibility in mind, you make the user experience better not only for users with these disabilities, but also for all of your other users.

In this lesson, you explore the features Android provides on the device to enable accessibility, including Google TalkBack (Android's screen reader) and other options in the Android framework.

**Note:** This practical describes the accessibility features in Android 7 (Nougat). Your device or version of Android may not have all these options, or the options may have different names.

## What you should already KNOW

You should be familiar with:

- How to navigate an Android device.

## What you will LEARN

You will learn how to:

- Turn on TalkBack and navigate the Google user interface with TalkBack enabled.
- Enable other accessibility settings to customize your device.

## What you will DO

- Experiment with the various accessibility settings on an Android device.

## App overview

For this lesson you use the built-in Android settings and apps. You do not build an app.

## Task 1. Explore TalkBack and text-to-speech

TalkBack is Android's built-in screen reader. With TalkBack enabled, the user can interact with their Android device without seeing the screen, because Android describes screen elements aloud. Users with visual impairments might rely on TalkBack to use your app.

In this task, you enable TalkBack to understand how screen readers work and how to navigate apps.

**Note:** By default, TalkBack is unavailable in the Android emulator and in some older devices. The source for the TalkBack feature is [available on GitHub](#). To test TalkBack on the emulator or on a device without TalkBack, download and build the source.

### 1.1 Turn on TalkBack

1. On an Android device or emulator, navigate to **Settings > Accessibility > TalkBack**.
2. Tap the **On/Off** toggle button to turn on TalkBack.
3. Tap **OK** to confirm permissions.
4. Confirm your device password, if asked.

If this is the first time you've run TalkBack, a tutorial launches. (The tutorial may not be available on older devices.) Use the tutorial to learn about:

- Explore by touch: TalkBack identifies every item you touch on the screen. You can touch items individually or move your finger over the screen. Swipe left or right to explore the items in tab (focus) order. The currently selected item has a green border around the view. To activate the selected item (the last item heard), double-tap it.
- Scrolling. Lists can be scrolled with a two finger scroll, or you can jump forward or

- back in a list with a side-by-side swipe.
- Finding global and local TalkBack menus.
- Setting the text navigation rate: Swipe up or down as TalkBack reads text to you a character, word, line, or paragraph at a time.
- Activating `EditText` views and entering text.

It may be helpful to navigate the tutorial with your eyes closed. To open the tutorial again in the future, navigate to **Settings > Accessibility > TalkBack > Settings > Launch TalkBack tutorial**.

## 1.2 Explore apps with TalkBack

With TalkBack enabled, explore the Camera and Calculator apps with these steps:

1. Tap the Home button, then double-tap the button to activate it and return to the Home screen.
2. Tap the Camera icon, then double-tap to activate it.
3. Explore the various buttons and options in the Camera app. TalkBack identifies each button and control by its *function* (for example, "Shutter Button" or "Zoom Out" button), not by its *appearance* ("plus button").
4. Tap the Home button, then double-tap the button to activate it and return to the Home screen.
5. Navigate to the Apps screen and activate the Calculator app.
6. Swipe right with one finger to navigate the views on the screen. Navigation in the app is not strictly left-to-right and top-to-bottom. All the numbers in the leftmost panel are identified before all the operations in the middle panel.
7. Keep swiping right to navigate to the advanced options screen, which is a panel in green that moves out from the right. Swipe left and double-tap to activate the main view and close the advanced operations. Note how these panel views do not have visible labels, but are identified by purpose in TalkBack.
8. Perform the calculation 3 times 5. TalkBack reads the numbers, the operation as the numbers are multiplied, and the result.
9. Navigate back to **Settings > Accessibility > TalkBack**, and turn off TalkBack. **TIP:** `</strong>` To scroll in TalkBack, use a two-finger swipe gesture.

## 1.3 Other speech settings

TalkBack is the most comprehensive of Android's screen reader functions. Besides TalkBack, Android provides other features for on-demand screen reading.

1. Navigate to **Settings > Accessibility > Select to Speak**. If you don't see Select to Speak, go to [Google Play](#) to download the latest version of TalkBack.

2. Tap the **On/Off** toggle button.
3. Tap **OK** to confirm permissions.
4. Confirm your device password, if asked.

A speech button appears in the lower right corner of the screen. Select to Speak works similarly to TalkBack, but Select to Speak works only on user-selected parts of the screen, and only by request.

5. Tap the speech button, then tap to select a portion of the text on the screen. Android reads that text to you.
6. Tap the Back button to return to the main **Accessibility** settings page.
7. Tap the speech button, then select a portion of the screen. Android reads all the text in the selection.
8. Tap **Select to Speak**. Turn the toggle button off.
9. Tap the Back button, then scroll down and tap **Accessibility shortcut**.

This option lets you use the device buttons to enable TalkBack by holding down the power button and then touching and holding two fingers on the screen. To disable TalkBack again, you must go to **Settings > Accessibility** and turn it off manually.

10. Tap the Back button, then scroll down to **Text-to-Speech output**. Tap that item.

The Google text-to-speech engine is used for TalkBack and other text-to-speech functions in apps that support speech-to-text. On this screen you can configure the settings for speech rate and pitch. You can also configure the speech language, change the language, and install new language packs. (To see all the options, tap the gear icon on the **Text-to-Speech output** screen.)

## Task 2. Explore other settings

TalkBack, Select to Speak, and the text-to-speech engine are helpful for users with visual impairments. Android provides several other accessibility features. In this task, you explore some of these other options.

### 2.1 Explore magnification, font, and display

1. On an Android device or emulator, navigate to **Settings > Accessibility > Magnification gesture**.
2. Tap the **On/Off** toggle button.

To use the magnification gesture, tap three times on the screen. Android zooms into the spot where you tapped, providing a magnified view. Tap again three times to exit magnification mode.

3. Tap the Back button to return to the accessibility settings.
4. Tap **Font size**. Move the slider all the way to the right to increase the font size to the largest setting. The preview at the top of the screen shows text at the chosen size.

A larger font size can make apps easier to read. A larger font size may cause app elements to also be larger or to wrap in unexpected ways.

**Note:** These font-size changes only affect apps that use the Android font system. In particular, the enlarged font size does not apply to the Chrome web browser. To enlarge the font for web pages, use **Chrome > Options menu > Settings > Accessibility > Text scaling**.

5. Return to the Home screen and examine the effect of the larger font in several apps. Try messages, email, calendar, or any of the apps you've created in this course. Note that apps that use their own internal fonts, such as some games, may not enlarge with this setting.
6. Navigate back to **Settings > Accessibility > Font size** and return the font size to the default.
7. Tap the Back button, select **Display size**, and move the slider all the way to the right.

Enlarging the display size makes all elements on the screen larger, including the text. Fewer elements fit on the screen. Enlarging the display size can make apps easier to read, and it also makes buttons and other interactive elements easier to tap.

8. Return the slider in the **Display size** settings to the default position.

## 2.2 Explore color and contrast

1. Tap **Settings > Accessibility > High contrast text**.

High contrast text fixes the display text to be either black or white, depending on the background, and outlines the text to emphasize it. High contrast text can make text easier to read.

**Note:** High contrast text is an experimental feature and may not work in all apps.

2. Return to the Home screen. See how text appears in messages, email, calendar, or any of the apps you've created in this course.
3. Return to **Settings > Accessibility** and turn **High contrast text** off again.
4. Scroll down to **Color inversion** and tap to turn it on.

Color inversion reverses the colors on your screen. For example, black text on a white screen becomes white text on a black screen. Color inversion can make some text easier to read and reduce eye strain.

5. Return to the Home screen and try some apps with inverted colors.
6. Swipe from the top of the screen to open the navigation drawer. Click the down arrow to open the quick settings.
7. Tap **Invert colors** to disable color inversion.
8. Navigate to **Settings > Accessibility > Color Correction**.

Color correction changes the system colors to help users with various forms of color blindness distinguish screen elements.

9. Tap the **On/Off** toggle button.
10. Navigate to the Home screen and note the differences in system colors.
11. Return to the **Color Correction** setting and tap to turn it off.

**High contrast text**, **Color inversion**, and **Color correction** are experimental features in Android. The features may not work in all apps and may affect app performance.

In the next practical, you learn how to support accessibility features in your own apps.

## Solution code

No project for this practical.

## Coding challenge

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

**Challenge:** Download and build the [RecyclerView app](#), if you have not already done so. Run the RecyclerView app with TalkBack enabled, and note where you could improve the label text and feedback.

## Summary

- An accessible app works well for all users, including users with low vision, blindness, deafness or hearing loss, cognitive impairments, or motor impairments.
- On an Android device, all accessibility features are available under **Settings > Accessibility**.

- With TalkBack enabled, users can interact with their Android device without seeing the screen, because Android describes various screen elements aloud. Users with visual impairments may rely on TalkBack when they use your app.
- The TalkBack "explore by touch" feature identifies the element under a user's finger.
- In TalkBack, swiping left or right identifies the next element to the user. Which element is next depends on the focus order, which is generally from left to right and top to bottom. If you don't want the default, you can define focus order for elements.
- Select To Speak works similarly to TalkBack, but only by request, and only for specific parts of the screen. To identify those parts of the screen, click the **Speech** button. Then tap or drag a selection on the screen.
- The Accessibility shortcut enables the user to turn on TalkBack quickly, without needing to navigate to the settings pages.
- The **Text-to-speech output** settings provide options for the Google text-to-speech engine, which TalkBack uses.
- Magnification gestures enable the user to magnify portions of the screen by tapping three times.
- Font and display-size settings can be used to enlarge the default system text or all app elements.
- The high-contrast-text setting fixes the display text to be either black or white, depending on the background, and outlines the text to emphasize it.
- The color-inversion setting reverses the colors on your screen.
- The color-correction setting changes the system colors to help users with various forms of color blindness distinguish screen elements.

## Related concept

The related concept documentation is in [Accessibility](#).

## Learn more

Android support documentation:

- [Android accessibility overview - Android Accessibility Help](#)
- [Get started on Android with TalkBack - Android Accessibility Help](#)
- [Select to Speak](#)
- [Magnification gestures](#)
- [Font size and display size](#)
- [High contrast text](#)
- [Accessibility shortcut](#)

- Text-to-speech output
- Color inversion
- Color correction

Other:

- [An Introduction to Android Accessibility Features - SitePoint](#)
- [What's New in Android Accessibility \(Google I/O '17\)](#)

## 6.2: Creating accessible apps

### Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Download and open the project
- Task 2. Add accessibility
- Task 3. Add an EditText view with a label
- Solution code
- Coding challenge
- Summary
- Related concept
- Learn more

Accessibility is a set of design, implementation, and testing techniques that enable your app to be usable by everyone, including people with disabilities.

Common disabilities that can affect a person's use of an Android device include blindness, low vision, color blindness, deafness or hearing loss, and restricted motor skills. When you develop your apps with accessibility in mind, you make the user experience better not only for users with these disabilities, but also for all of your other users.

Accessibility usually does not require a full overhaul of your app's design or code. Accessibility *does* require you to pay attention to details and test your app under the same conditions your users may encounter.

Android includes several accessibility-related features, to help you optimize your app's user interface (UI) for accessibility. In this lesson, you learn how to test your app and add features that enhance its accessibility.

## What you should already KNOW

You should be familiar with:

- Creating, building, and running apps in Android Studio.
- Creating views and layouts in both Android Studio's design editor and XML.

# What you will LEARN

You will learn how to:

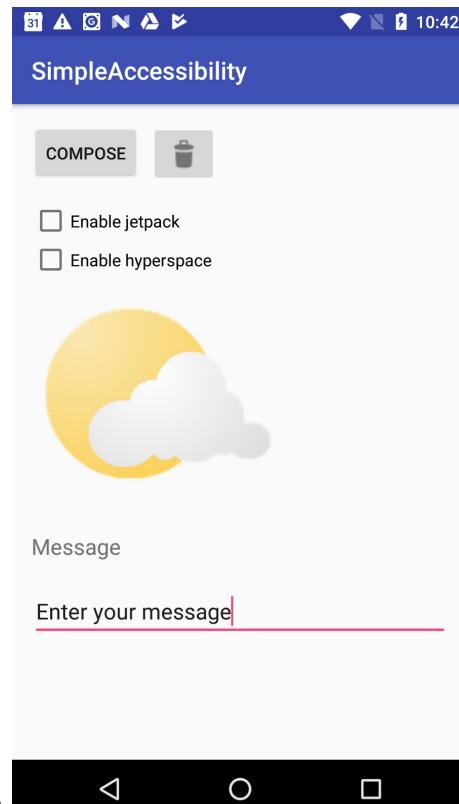
- Test your app for accessibility in a variety of ways.
- Add attributes to your XML layouts that assist with making your app more accessible for your users.

# What you will DO

- Download and run a starter app.
- Test the app with Google [TalkBack](#) (Android's screen reader) to identify possible accessibility issues. TalkBack gives spoken feedback so that you can use your device without looking at the screen.
- Run Android Studio's code inspector to identify missing accessibility-related attributes.
- Update the app to resolve the accessibility issues revealed in testing.

# App overview

The SimpleAccessibility app demonstrates how to add accessible features to your app's UI.



The app, when complete, looks like this:

None of the views have click handlers, so the app does not have any actual functionality. The focus for this app (and this lesson) is in the layout XML code and in the attributes that enable accessibility.

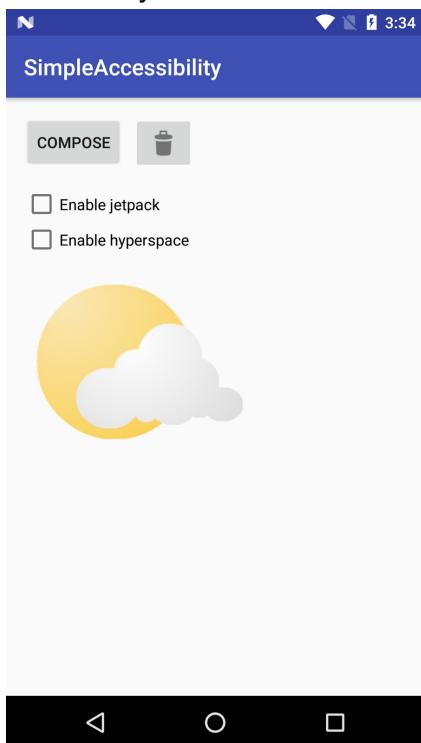
## Task 1. Download and open the project

In this task, you run the starter app for SimpleAccessibility and explore it in TalkBack.

### 1.1 Download and open the sample app

1. Download the [SimpleAccessibility-start](#) app and unzip the file.
2. Open the app in Android Studio.
3. Open `res/layout/activity_main.xml`.

The SimpleAccessibility app contains a number of views in a layout, including buttons (`Button` and `ImageButton` views), checkboxes, and an image. All the accessibility functions you learn about in this practical are implemented in the layout.



### 1.2 Test the app with TalkBack

TalkBack is Android's built-in screen reader. When TalkBack is on, the user can interact with their Android device without seeing the screen. Users with visual impairments may rely on TalkBack to use your app. In this task, you manually explore the app with TalkBack enabled, to expose possible accessibility problems.

**Note:** By default, TalkBack is unavailable in the Android emulator and in some older devices. The source for the TalkBack feature is [available on GitHub](#). To test TalkBack on the emulator or on a device without TalkBack, download and build the source.

To test the app with TalkBack enabled, use these steps:

1. On a device, navigate to **Settings > Accessibility > TalkBack**.
2. Tap the **On/Off** toggle button to turn on TalkBack.
3. Tap **OK** to confirm permissions. After TalkBack is on, you can navigate the Android UI in one of these ways:
  - Tap a UI element to hear the description for that element. Double-tap to select.
  - Swipe right or left to navigate through the elements in sequence. Double-tap anywhere to select.
  - Drag your finger over the screen to hear what's under your finger. Double-tap anywhere to select.
4. Build and run your app in Android Studio.

When your app starts with TalkBack enabled, no item is initially focused. TalkBack reads the title of the app.

5. Tap each element or swipe to hear the descriptions for the elements in sequence. Note the following:
  - The spoken feedback for the **Compose** button is the button title itself.
  - The feedback for the image button is "unlabelled button."
  - The feedback for a checkbox is the checkbox label and the current state of the checkbox (selected or cleared.)
  - No feedback is available for the partly cloudy image. The image is not even a focusable element.
6. Swipe from left to right to navigate between focusable elements on the screen. Note that the focus moves from top to bottom. The partly cloudy image is not focusable.

## Task 2. Add accessibility

Experimenting with TalkBack and the starter app in the previous task exposed problems in how the app's views are identified for vision-impaired users. In this task, you fix some of these problems.

### 2.1 Inspect and fix the code in Android Studio

Android Studio highlights places in your XML layout code that have potential accessibility problems, and makes suggestions for fixing those problems. For accessibility, Android Studio checks two things:

- Missing content descriptions on `ImageView` and `ImageButton` objects. Because these views do not have text associated with them, the TalkBack screen reader can't describe them without an explicit description.
- Missing label indicators (API 17 or higher). Text views are often used as labels for some other view in the layout, for example, as a label to indicate the content for an `EditText`. A label indicator specifies the other view that this view is a label for.

To inspect and fix your code for accessibility, use these steps:

1. In Android Studio, open the `res/layout/activity_main.xml` file, if it's not already open. Switch to the **Text** tab.
2. Note that the `ImageButton` and `ImageView` elements have yellow highlights on them. When you hover over the elements, the message reads, "Missing contentDescription attribute on image."
3. Add an `android:contentDescription` attribute to the `ImageButton` view, with the value "Discard." Extract this string into a resource.
4. Add an `android:contentDescription` attribute to the `ImageView`, with the value "Partly Cloudy." Extract this string into a resource.
5. Build and run your app.
6. Navigate your app with TalkBack turned on. The trash-can button now has a reasonable description, but you still can't focus on the partly cloudy image.

## 2.2 Add focus to views

In addition to readable descriptions, make sure that users can navigate your screen layouts using hardware or software-based directional controls such as D-pads, trackballs, keyboards, or on-screen gestures.

Most views are focusable by default, and the focus moves from view to view in your layout. The Android platform tries to figure out the most logical focus order based on each view's closest neighbor. Generally, views are focussed from left to right and top to bottom. In some cases, you want to make your views explicitly focusable or change the focus order to reflect how the user uses your app.

In the case of the SimpleAccessibility app, the partly cloudy image is not focusable, even with a content description added. Assuming that the partly cloudy image is there to show the current weather, you must add a focusable attribute to that image so that TalkBack can read the content description.

To make the partly cloudy image focusable, use these steps:

1. Find the partly cloudy `ImageView` element in the XML layout.
2. Add the `android:focusable` attribute to the `ImageView`:

```
    android:focusable="true"
```

3. Build and run the app. With TalkBack enabled, navigate to the partly cloudy image. TalkBack reads the image's content description.

**Note:** In addition to making an item focusable, you can add focus-related attributes to the views in your app. The attributes include `nextFocusUp`, `nextFocusDown`, `nextFocusLeft`, `nextFocusRight`, and `nextFocusForward`. See [Supporting Keyboard Navigation](#) for more information on how to navigate using input focus.

## Task 3. Add an `EditText` view with a label

In this task, you add an `EditText` view and an associated label (a `TextView`). `EditText` views and `TextView` labels are a special case for handling accessibility in your app.

### 3.1 Add views and test the app in TalkBack

1. Add a `TextView` to the app's layout below the `ImageView`. Give it the `android:text` attribute of "Message." Extract that string into a resource.
2. Add an `EditText` just below the `TextView`. Give it an ID of `@+id/edittext_message`, and the `android:hint` attribute "Enter your message." Extract that string into a resource.
3. Build and run the app and navigate to the new views. Experiment with entering text in TalkBack.

Note these things:

- For text views, TalkBack reads the text that's in the `android:text` attribute. Here, the text is "Message." You do not need content descriptions for text views.
- For `EditText` views, TalkBack reads the text that's in the `android:hint` attribute. Here, the text is "Enter your message." If `android:hint` does not exist, TalkBack reads the text that's in `android:text` instead.

In `EditText` views, it's better to use `android:hint` rather than `android:text` for the default text.

- When the app starts, the first `EditText` has the initial focus.

**TIP:** In TalkBack, you can move the cursor in an `EditText` view with the device's volume up and volume down buttons.

## 3.2 Add explicit labels for `EditText` views (API 17 and higher)

If you target your app to Android 4.2 (API level 17) or higher, use the `android:labelFor` attribute when labeling views that serve as content labels for other views. For readable descriptions, TalkBack prefers explicit labels (using `android:labelFor`) over attributes such as `android:text` or `android:hint`.

Explicit labels are only available in API 17 or higher, and Android Studio can highlight missing labels for `EditText` views as part of code inspection. The SimpleAccessibility app uses the default minimum SDK of 15. In this task, you change the API level to enable explicit labels. Then you add the appropriate label attribute.

1. Open the `build.gradle` (Module: `app`) file and change `minSdkVersion` from `15` to `17`.
2. Click **Sync Now** to rebuild the project.
3. In the `activity_main.xml` layout file, delete the `android:hint` attribute from the `EditText`.

The `EditText` element is now highlighted in yellow. When you hover over the element, the message reads, "No label views point to this text field with an `android:labelFor="@+id/edittext_message"` attribute."

4. Add the `android:labelFor` attribute to the `TextView` that serves as a label for this `EditText`:

```
    android:labelFor="@+id/edittext_message"
```

The highlight on the `EditText` disappears.

5. Build and run the app. TalkBack now reads the contents of the label to identify the `EditText`.

## Solution code

Android Studio project: [SimpleAccessibility](#)

## Coding challenge

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

**Challenge:** If the content of a view changes programmatically as the app runs, you need to update the content descriptions to reflect the new state.

- Modify the SimpleAccessibility app to include a click handler for the **Discard** (trash can) button. When the button is clicked, toggle between two images: the default trash can and a lock icon. (You can use `@android:drawable/ic_lock_lock` for the lock icon.)
- Use the `setContentDescription()` method to change the content description when the button image changes.
- Test the app in TalkBack. Verify that the correct content description appears for each image.

#### Hints:

Get an image drawable from the resources:

```
Drawable img = ContextCompat.getDrawable(this, R.drawable.my_drawable);
```

Get a string from the resources:

```
String str = getResources().getString(R.string.my_string);
```

Compare two drawables:

```
if (drawable1.getConstantState() == drawable2.getConstantState()) {  
    ...  
}
```

## Summary

- Adding accessibility to your app does not require significant code changes. You can add many accessibility features to an existing app through attributes in the XML layout. Use Android's TalkBack feature to test your app for users with low vision.
- Android Studio can highlight missing accessibility attributes (content descriptions and labels) in your app's layout.
- To provide readable descriptions of buttons, add `android:contentDescription` attributes to `ImageView` and `ImageButton` elements.
- To provide the ability to navigate your app's UI, use focus and focus order.
- Add `android:focusable` attributes to `ImageView` views that are not by default focusable.
- You don't need to add content descriptions or focus to decorative images.
- For `EditText` views, use `android:hint` instead of `android:contentDescription`.

- If your app supports API 17 or higher, use the `android:labelFor` attribute to indicate that a view is a label for some other view.

## Related concept

The related concept documentation is in [Accessibility](#).

## Learn more

Android support documentation:

- [Android accessibility overview - Android Accessibility Help](#)
- [Get started on Android with TalkBack - Android Accessibility Help](#)
- [Editable View labels - Android Accessibility Help](#)
- [Content labels - Android Accessibility Help](#)

Android developer documentation:

- [Accessibility Overview](#)
- [Making Apps More Accessible](#)
- [Accessibility Developer Checklist](#)
- [Building Accessible Custom Views](#)
- [Developing Accessible Applications](#)
- [Designing Effective Navigation](#)
- [Testing Your App's Accessibility](#)
- [Developing an Accessibility Service](#)

Material Design: [Usability - Accessibility](#)

Videos:

- [GTAC 2015: Automated Accessibility Testing for Android Applications](#)
- [Google I/O 2015 - Improve your Android app's accessibility](#)

Other:

- [Accessibility Testing on Android](#)
- [An Introduction to Android Accessibility Features - SitePoint](#)
- [Having Trouble Focusing? A Primer on Focus in Android](#)

# 7.1: Using the device location

## Contents:

- [Introduction](#)
- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Set up location services](#)
- [Task 2. Get the last known location](#)
- [Task 3. Get the location as an address](#)
- [Task 4. Receive location updates](#)
- [Coding challenge](#)
- [Solution code](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

Users are constantly moving around in the world, usually with their phones in their pockets. Advances in GPS and network technologies have made it possible to create Android apps with precise location awareness, using the location services APIs included in Google Play services.

When an app requests the device location, it impacts network and battery consumption, which impacts device performance. To improve the performance of your app, keep the frequency of location requests as low as possible.

In this practical, you learn how to access the device's last known location. You also learn how to convert a set of geographic coordinates (longitude and latitude) into a street address, and how to perform periodic location updates.

## What you should already KNOW

You should be familiar with:

- Creating, building, and running apps in Android Studio.
- The `Activity` lifecycle.
- Making data persistent across configuration changes.
- Requesting permissions at runtime.

- Using an `AsyncTask` to do background work.

## What you will LEARN

You will learn how to:

- Get the last known location of the device.
- Obtain a physical address from a set of coordinates (reverse geocoding).
- Perform periodic location updates.

## What you will DO

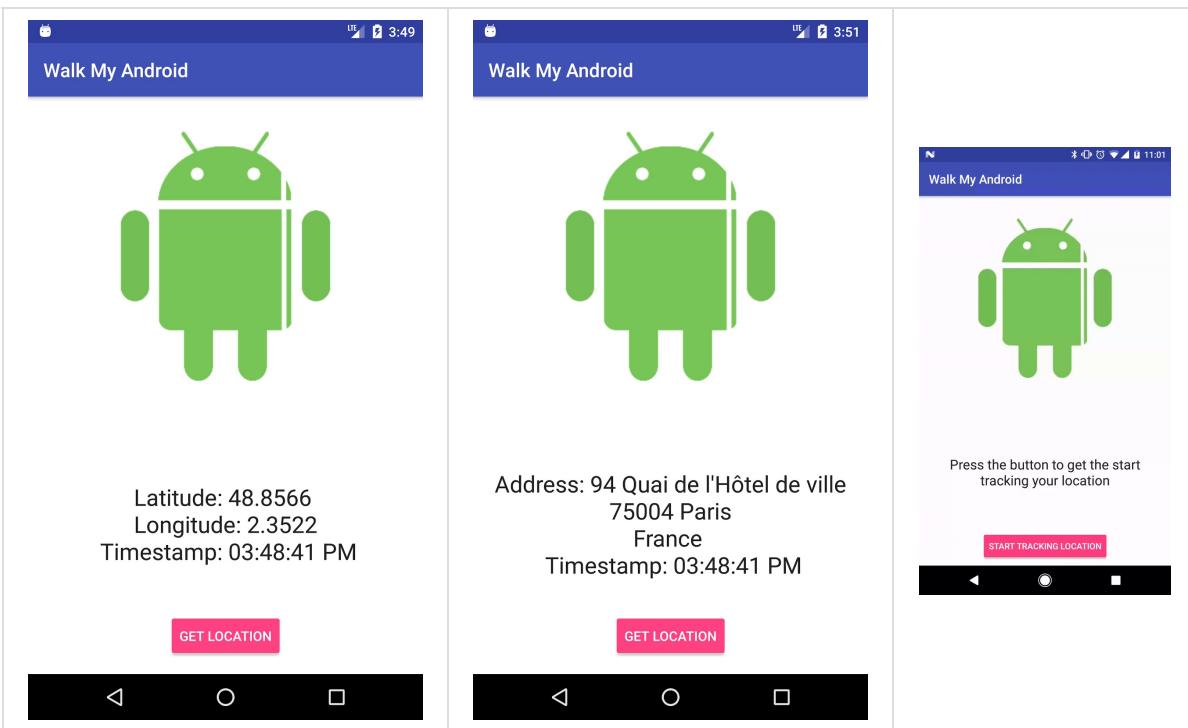
- Create the WalkMyAndroid app and have it display the device's last known location as latitude and longitude coordinates.
- Reverse geocode the latitude and longitude coordinates into a physical address.
- Extend the app to receive periodic location updates.

## App overview

The WalkMyAndroid app prompts the user to change the device location settings, if necessary, to allow the app to access precise location data. The app shows the device location as latitude and longitude coordinates, then shows the location as a physical address. The completed app does periodic location updates and shows an animation that gives the user a visual cue that the app is tracking the device's location.

You create the WalkMyAndroid app in phases:

- In tasks 1 and 2, you implement a button that gets the most recent location for the device and displays the coordinates and timestamp of the location in a `TextView`.
- In task 3, you turn the coordinates into a physical address, through a process called *reverse geocoding*.
- In task 4, you learn how to trigger periodic updates of the location.



## Task 1. Set up location services

Obtaining the location information for a device can be complicated: Devices contain different types of GPS hardware, and a satellite or network connection (cell or Wi-Fi) is not always available. Activating GPS and other hardware components uses power. (For more information about GPS and power use, see the chapter on performance.)

To find the device location efficiently without worrying about which provider or network type to use, use the `FusedLocationProviderClient` interface. Before you can use `FusedLocationProviderClient`, you need to set up Google Play services. In this task, you download the starter code and include the required dependencies.

### 1.1 Download the starter app

1. Download the starter app for this practical, [WalkMyAndroid-Starter](#).
2. Open the starter app in Android Studio, rename the app to WalkMyAndroid, and run it.
3. You might need to update your Android SDK Build-Tools. To do this, use the [Android SDK Manager](#).

The UI has a **Get Location** button, but tapping it doesn't do anything yet.

### 1.2 Set up Google Play services

Install the Google Repository and update the Android SDK Manager:

1. Open Android Studio.
2. Select **Tools > Android > SDK Manager**.
3. Select the **SDK Tools** tab.
4. Expand **Support Repository**, select **Google Repository**, and click **OK**.

Now you can include Google Play services packages in your app.

To add Google Play services to your project, add the following line of code to the `dependencies` section in your app-level `build.gradle` (Module: app) file:

```
compile 'com.google.android.gms:play-services-location:XX.X.X'
```

Replace `xx.x.x` with the latest version number for Google Play services, for example `11.0.2`. Android Studio will let you know if you need to update it. For more information and the latest version number, see [Add Google Play Services to Your Project](#).

**Note:** If your app references more than 65K methods, the app may fail to compile. To mitigate this problem, compile your app using only the Google Play services APIs that your app uses. To learn how to selectively compile APIs into your executable, see [Set Up Google Play Services](#) in the developer documentation. To learn how to enable an app configuration known as *multidex*, see [Configure Apps with Over 64K Methods](#).

Now that you have Google Play services installed, you're ready to connect to the [LocationServices API](#).

## Task 2. Get the last known location

The `LocationServices` API uses a fused location provider to manage the underlying technology and provides a straightforward API so that you can specify requirements at a high level, like high accuracy or low power. It also optimizes the device's use of battery power. In this step, you will use it to obtain the device's last known location.

### 2.1 Set location permission in the manifest

Using the `Location` API requires permission from the user. Android offers two location permissions:

- `ACCESS_COARSE_LOCATION`
- `ACCESS_FINE_LOCATION`

The permission you choose determines the accuracy of the location returned by the API. For this lesson, use the `ACCESS_FINE_LOCATION` permission, because you want the most accurate location information possible.

Add the following element to your manifest file, above the `<application>` element:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

## 2.2 Request permission at runtime

Starting with Android 6.0 (API level 23), it's not always enough to include a permission statement in the manifest. For "dangerous" permissions, you also have to request permission programmatically, at runtime. To request location permission at runtime:

1. Create an `OnClickListener` for the **Get Location** button in `onCreate()`.
2. Create a method stub called `getLocation()` that takes no arguments and doesn't return anything. Invoke the `getLocation()` method from the button's `onClick()` method.
3. In the `getLocation()` method, check for the `ACCESS_FINE_LOCATION` permission.
  - If the permission has not been granted, request it.
  - If the permission has been granted, display a message in the logs. (The code below shows a `TAG` variable, which you declare later, in Task 3.1.)

For a refresher on runtime permissions, check out [Requesting Permissions at Run Time](#).

```
private void getLocation() {  
    if (ActivityCompat.checkSelfPermission(this,  
        Manifest.permission.ACCESS_FINE_LOCATION)  
        != PackageManager.PERMISSION_GRANTED) {  
        ActivityCompat.requestPermissions(this, new String[]  
            {Manifest.permission.ACCESS_FINE_LOCATION},  
            REQUEST_LOCATION_PERMISSION);  
    } else {  
        Log.d(TAG, "getLocation: permissions granted");  
    }  
}
```

4. In your `MainActivity` class, define an integer constant `REQUEST_LOCATION_PERMISSION`. This constant is used to identify the permission request when the results come back in the `onRequestPermissionsResult()` method. It can be any integer greater than `0`.
5. Override the `onRequestPermissionsResult()` method. If the permission was granted, call `getLocation()`. Otherwise, show a `Toast` saying that the permission was denied.

```

@Override
public void onRequestPermissionsResult(int requestCode,
    @NonNull String[] permissions, @NonNull int[] grantResults) {
    switch (requestCode) {
        case REQUEST_LOCATION_PERMISSION:
            // If the permission is granted, get the location,
            // otherwise, show a Toast
            if (grantResults.length > 0
                && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                getLocation();
            } else {
                Toast.makeText(this,
                    R.string.location_permission_denied,
                    Toast.LENGTH_SHORT).show();
            }
            break;
    }
}

```

- Run the app. Clicking the button requests permission from the user. If permission is granted, you see a log statement in the console.

After you grant permission, subsequent clicks on the **Get Location** button have no effect. Because you already granted permission, the app doesn't need to ask for permission again, even if you close and restart the app.

## 2.3 Get the last known location

The `getLastLocation()` method doesn't actually make a location request. It simply returns the location most recently obtained by the `FusedLocationProviderClient` class.

If no location has been obtained since the device was restarted, the `getLastLocation()` method may return `null`. Usually, the `getLastLocation()` method returns `Location` object that contains a timestamp of when this location was obtained.

To get the last known location:

- In `strings.xml`, add a string resource called `location_text`. Use `location_text` to display the latitude, longitude, and timestamp of the last known location.

```

<string name="location_text">"Latitude: %1$.4f \n Longitude: %2$.4f
\n Timestamp: %3$tr"</string>

```

**Note:** If you aren't familiar with string replacement and formatting, see [Formatting and Styling](#) and the [Formatter](#) documentation.

2. In your `MainActivity` class, create a member variable of the `Location` type called `mLastLocation`.
3. Find the location `TextView` by ID (`textview_location`) in `onCreate()`. Assign the `TextView` to a member variable called `mLocationTextView`.
4. Create a member variable of the `FusedLocationProviderClient` type called `mFusedLocationClient`.
5. Initialize `mFusedLocationClient` in `onCreate()` with the following code:

```
mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this);
```

The `getLastLocation()` method returns a `Task` that results in a `Location` object (after the Task's `onSuccess()` callback method is called, signifying that the `Task` was successful).

Retrieve the latitude and longitude coordinates of a geographic location from the resulting `Location` object:

6. Replace the log statement in the `getLocation()` method with the following code snippet. The code obtains the device's most recent location and assigns it to `mLastLocation`.

- If the returned location is not `null`, set the `TextView` to show the coordinates and time stamp of the `Location` object.
- If the location returned is `null`, the fused location provider has not obtained a location since the device was restarted. Display a message in the `TextView` that says that the location is not available.

```
mFusedLocationClient.getLastLocation().addOnSuccessListener(  
    new OnSuccessListener<Location>() {  
        @Override  
        public void onSuccess(Location location) {  
            if (location != null) {  
                mLastLocation = location;  
                mLocationTextView.setText(  
                    getString(R.string.location_text,  
                        mLastLocation.getLatitude(),  
                        mLastLocation.getLongitude(),  
                        mLastLocation.getTime()));  
            } else {  
                mLocationTextView.setText(R.string.no_location);  
            }  
        }  
    }
```

7. Run the app. You now see the latest location that is stored in the fused location provider.

## Testing location on an emulator

If you test the WalkMyAndroid app on an emulator, use a system image that supports Google APIs or Google Play:

1. In Android Studio, create a new virtual device and select hardware for it.
2. In the **System Image** dialog, choose an image that says "Google APIs" or "Google Play" in the **Target** column.

To update the fused location provider on an emulator:

1. The emulator appears on your screen with a vertical menu to the right of the virtual device. To access emulator options, click the ... icon at the bottom of this vertical menu.
2. Click **Location**.
3. Enter or change the coordinates in the **Longitude** and **Latitude** fields.
4. Click **Send** to update the fused location provider.

Clicking **Send** does not affect the location returned by `getLastLocation()`, because `getLastLocation()` uses a local cache that the emulator tools do not update.

When you test the app on an emulator, the `getLastLocation()` method might return `null`, because the fused location provider doesn't update the location cache after the device is restarted. If `getLastLocation()` returns `null` unexpectedly:

1. Start the Google Maps app and accept the terms and conditions, if you haven't already.
2. Use the steps above to update the fused location provider. Google Maps will force the local cache to update.
3. Go back to your app and click the **Get Location** button. The app updates with the new location.

Later in this lesson, you learn how to force the fused location to update the cache using periodic updates.

## Task 3. Get the location as an address

Your app displays the device's most recent location, which usually corresponds to its current location, using latitude and longitude coordinates. Although latitude and longitude are useful for calculating distance or displaying a map position, in many cases the address of the location is more useful. For example, if you want to let your users know where they are or what is close by, a street address is more meaningful than the geographic coordinates of the location.

The process of converting from latitude/longitude to a physical address is called *reverse geocoding*. The `getFromLocation()` method provided by the `Geocoder` class accepts a latitude and longitude and returns a list of addresses. The method is synchronous and requires a network connection. It may take a long time to do its work, so do not call it from the main user interface (UI) thread of your app.

In this task, you subclass an `AsyncTask` to perform reverse geocoding off the main thread. When the `AsyncTask` completes its process, it updates the UI in the `onPostExecute()` method.

(Using an `AsyncTask` means that when your main `Activity` is destroyed when the device orientation changes, you will no longer be able to update the UI. To handle this, you make the location tracking state persistent in Task 4.5.)

## 3.1 Create an `AsyncTask` subclass

Recall that an `AsyncTask` object is used to perform work off the main thread. An `AsyncTask` object contains one required override method, `doInBackground()` which is where the work is performed. For this use case, you need another override method, `onPostExecute()`, which is called on the main thread after `doInBackground()` finishes. In this step, you set up the boilerplate code for your `AsyncTask`.

`AsyncTask` objects are defined by three generic types:

- Use the `Params` type to pass parameters into the `doInBackground()` method. For this app, the passed-in parameter is the `Location` object.
- Use the `Progress` type to mark progress in the `onProgressUpdate()` method. For this app, you are not interested in the `Progress` type, because reverse geocoding is typically quick.
- Use the `Results` type to publish results in the `onPostExecute()` method. For this app, the published result is the returned address String.

To create a subclass of `AsyncTask` that you can use for reverse geocoding:

1. Create a new class called `FetchAddressTask`, and use it to subclass `AsyncTask`.

Parameterize the `AsyncTask` using the three types described above:

```
private class FetchAddressTask extends AsyncTask<Location, Void, String> {}
```

2. In Android Studio, this class declaration is underlined in red, because you have not implemented the required `doInBackground()` method. Press **Alt + Enter (Option + Enter on a Mac)** on the highlighted line and select **Implement methods**. (Or select **Code > Implement methods**.)

Notice that the method signature for `doInBackground()` includes a parameter of the `Location` type, and returns a `String`; this comes from parameterized types in the class declaration.

3. Override the `onPostExecute()` method by going to the menu and selecting **Code > Override Methods** and selecting `onPostExecute()`. Again notice that the passed-in parameter is automatically typed as a String, because this what you put in the `FetchAddressTask` class declaration.
4. Create a constructor for the `AsyncTask` that takes a `Context` as a parameter and assigns it to a member variable.

Your `FetchAddressTask` now looks something like this:

```
private class FetchAddressTask extends AsyncTask<Location, Void, String> {
    private final String TAG = FetchAddressTask.class.getSimpleName();
    private Context mContext;

    FetchAddressTask(Context applicationContext) {
        mContext = applicationContext;
    }

    @Override
    protected String doInBackground(Location... locations) {
        return null;
    }
    @Override
    protected void onPostExecute(String address) {
        super.onPostExecute(address);
    }
}
```

## 3.2 Convert the location into an address string

In this step, you complete the `doInBackground()` method so that it converts the passed-in `Location` object into an address string, if possible. If there is a problem, you show an error message.

1. Create a `Geocoder` object. This class handles both geocoding (converting from an address into coordinates) and reverse geocoding:

```
Geocoder geocoder = new Geocoder(mContext,
    Locale.getDefault());
```

2. Obtain a `Location` object. The passed-in parameter is a Java `varargs` argument that can contain any number of objects. In this case we only pass in one `Location` object,

so the desired object is the first item in the `varargs` array:

```
Location location = params[0];
```

3. Create an empty `List` of `Address` objects, which will be filled with the address obtained from the `Geocoder`. Create an empty `String` to hold the final result, which will be either the address or an error:

```
List<Address> addresses = null;
String resultMessage = "";
```

4. You are now ready to start the geocoding process. Open up a `try` block and use the following code to attempt to obtain a list of addresses from the `Location` object. The third parameter specifies the maximum number of addresses that you want to read. In this case you only want a single address:

```
try {
    addresses = geocoder.getFromLocation(
        location.getLatitude(),
        location.getLongitude(),
        // In this sample, get just a single address
        1);
}
```

5. Open a `catch` block to catch `IOException` exceptions that are thrown if there is a network error or a problem with the `Geocoder` service. In this `catch` block, set the `resultMessage` to an error message that says "Service not available." Log the error and `result message`:

```
catch (IOException ioException) {
    // Catch network or other I/O problems
    resultMessage = mContext
        .getString(R.string.service_not_available);
    Log.e(TAG, resultMessage, ioException);
}
```

6. Open another `catch` block to catch `IllegalArgumentException` exceptions. Set the `resultMessage` to a string that says "Invalid coordinates were supplied to the Geocoder," and log the error and `result message`:

```

        catch (IllegalArgumentException illegalArgumentException) {
            // Catch invalid latitude or longitude values
            resultMessage = mContext
                .getString(R.string.invalid_lat_long_used);
            Log.e(TAG, resultMessage + ". " +
                  "Latitude = " + location.getLatitude() +
                  ", Longitude = " +
                  location.getLongitude(), illegalArgumentException);
        }
    }
}

```

7. You need to catch the case where `Geocoder` is not able to find the address for the given coordinates. In the `try` block, check the address list and the `resultMessage` string. If the address list is empty or `null` and the `resultMessage` string is empty, then set the `resultMessage` to "No address found" and log the error:

```

if (addresses == null || addresses.size() == 0) {
    if (resultMessage.isEmpty()) {
        resultMessage = mContext
            .getString(R.string.no_address_found);
        Log.e(TAG, resultMessage);
    }
}

```

8. If the address list is not empty or `null`, the reverse geocode was successful.

The next step is to read the first address into a string, line by line:

- Create an empty `ArrayList` of `Strings`.
- Iterate over the `List` of `Address` objects and read them into the new `ArrayList` line by line.
- Use the `TextUtils.join()` method to convert the list into a string. Use the `\n` character to separate each line with the new-line character:

Here is the code:

```

else {
    // If an address is found, read it into resultMessage
    Address address = addresses.get(0);
    ArrayList<String> addressParts = new ArrayList<>();

    // Fetch the address lines using getAddressLine,
    // join them, and send them to the thread
    for (int i = 0; i <= address.getMaxAddressLineIndex(); i++) {
        addressParts.add(address.getAddressLine(i));
    }

    resultMessage = TextUtils.join("\n", addressParts);
}

```

9. At the bottom of `doInBackground()` method, return the `resultMessage` object.

### 3.3 Display the result of the FetchAddressTask object

When `doInBackground()` completes, the `resultMessage` string is automatically passed into the `onPostExecute()` method. In this step you update the member variables of `MainActivity` with new values and display the new data in the `TextView` using a passed in interface.

1. Create a new string resource with two replacement variables.

```
<string name="address_text">"Address: %1$s \n Timestamp: %2$tr"</string>
```

2. Create an interface in `FetchAddressTask` called `OnTaskCompleted` that has one method, called `onTaskCompleted()`. This method should take a string as an argument:

```
interface OnTaskCompleted {  
    void onTaskCompleted(String result);  
}
```

3. Add another parameter to the constructor for the `OnTaskCompleted` interface and assign it to a member variable:

```
private OnTaskCompleted mListener;  
  
FetchAddressTask(Context applicationContext, OnTaskCompleted listener) {  
    mContext = applicationContext;  
    mListener = listener;  
}
```

4. In the `onPostExecute()` method, call `onTaskCompleted()` on the `mListener` interface, passing in the result string:

```
@Override  
protected void onPostExecute(String address) {  
    mListener.onTaskCompleted(address);  
    super.onPostExecute(address);  
}
```

5. Back in the `MainActivity`, have the activity implement the interface you created and override the required `onTaskCompleted()` method.

6. In this method, updated the `TextView` with the resulting address and the current time:

```
@Override  
public void onTaskCompleted(String result) {  
    // Update the UI  
    mLocationTextView.setText(getString(R.string.address_text,  
        result, System.currentTimeMillis()));  
}
```

7. In the `getLocation()` method, inside the `onSuccess()` callback, replace the lines that assigns the passed-in location to `mLastLocation` and sets the `TextView` with the following line of code. This code creates a new `FetchAddressTask` and executes it, passing in the `Location` object. You can also remove the now unused `mLastLocation` member variable.

```
// Start the reverse geocode AsyncTask  
new FetchAddressTask(MainActivity.this,  
    MainActivity.this).execute(location);
```

8. At the end of the `getLocation()` method, show loading text while the `FetchAddressTask` runs:

```
mLocationTextView.setText(getString(R.string.address_text,  
    getString(R.string.loading),  
    System.currentTimeMillis()));
```

9. Run the app. After briefly loading, the app displays the location address in the `TextView`.

## Task 4. Receive location updates

Up until now, you've used the `FusedLocationProviderClient.getLastLocation()` method, which relies on other apps having already made location requests. In this task, you learn how to:

- Track the device location using periodic location requests.
- Make the tracking state persistent.
- Show an animation to give a visual cue that the device location is being tracked.
- Check the device location settings. You need to know whether location services are turned on, and whether they are set to the accuracy that your app needs.

### 4.1 Set up the UI and method stubs

If your app relies heavily on device location, using the `getLastLocation()` method may not be sufficient, because `getLastLocation()` relies on a location request from a different app and only returns the last value stored in the provider.

To make location requests in your app, you need to:

1. Create a `LocationRequest` object that contains the requirements for your location requests. The requirements include update frequency, accuracy, and so on. You do this step in [4.2 Create the LocationRequest object](#), below.
2. Create a `LocationCallback` object and override its `onLocationResult()` method. The `onLocationResult()` method is where your app receives location updates. You do this step in [4.3 Create the LocationCallback object](#).
3. Call `requestLocationUpdates()` on the `FusedLocationProviderClient`. Pass in the `LocationRequest` and the `LocationCallback`. You do this step in [4.4 Request location updates](#).

The user has no way of knowing that the app is making location requests, except for a tiny icon in the status bar. In this step, you use an animation (included in the starter code) to add a more obvious visual cue that the device's location is being tracked. You also change the button text to show the user whether location tracking is on or off.

To indicate location tracking to the user:

1. Declare the member variables `mAndroidImageView` (of type `ImageView`) and `mRotateAnim` (of type `AnimatorSet`).
2. In the `onCreate()` method, find the Android `ImageView` by ID and assign it to `mAndroidImageView`. Then find the animation included in the starter code by ID and assign it to `mRotateAnim`. Finally set the Android `ImageView` as the target for the animation:

```
mAndroidImageView = (ImageView) findViewById(R.id.imageview_android);

mRotateAnim = (AnimatorSet) AnimatorInflater.loadAnimator
    (this, R.animator.rotate);

mRotateAnim.setTarget(mAndroidImageView);
```

3. In the `strings.xml` file:
  - Change the button text to "Start Tracking Location." Do this for both the portrait and the landscape layouts.
  - Change the `TextView` text to "Press the button to start tracking your location."
4. Refactor and rename the `getLocation()` method to `startTrackingLocation()`.
5. Create a private method stub called `stopTrackingLocation()` that takes no arguments

- and returns `void`.
6. Create a boolean member variable called `mTrackingLocation`. Boolean primitives default to `false`, so you do not need to initialize `mTrackingLocation`.
  7. Change the `onClick()` method for the button's `onClickListener`:
    - If `mTrackingLocation` is `false`, call `startTrackingLocation()`.
    - If `mTrackingLocation` is `true`, call `stopTrackingLocation()`.

```
@Override
public void onClick(View v) {
    if (!mTrackingLocation) {
        startTrackingLocation();
    } else {
        stopTrackingLocation();
    }
}
```

8. At the end of the `startTrackingLocation()` method, start the animation by calling `mRotateAnim.start()`. Set `mTrackingLocation` to `true` and change the button text to "Stop Tracking Location".
9. In the `stopTrackingLocation()` method, check if you are tracking the location. If you are, stop the animation by calling `mRotateAnim.end()`, set `mTrackingLocation` to `false`, change the button text back to "Start Tracking Location" and reset the location `TextView` to show the original hint.

```
/**
 * Method that stops tracking the device. It removes the location
 * updates, stops the animation and reset the UI.
 */
private void stopTrackingLocation() {
    if (mTrackingLocation) {
        mTrackingLocation = false;
        mLocationButton.setText(R.string.start_tracking_location);
        mLocationTextView.setText(R.string.textview_hint);
        mRotateAnim.end();
    }
}
```

## 4.2 Create the LocationRequest object

The `LocationRequest` object contains setter methods that determine the frequency and accuracy of location updates. For now, we're only interested in the following parameters:

- **Interval:** The `setInterval()` method defines the desired update interval in milliseconds. For this app, use 10 seconds (10000 milliseconds).
- **Fastest interval:** The fused location provider attempts to make location request more

efficient by batching requests from different apps. This means that you may receive updates faster than what you set in `setInterval()`, which can cause problems if your UI is not ready for updates. To limit the rate of location updates, use the `setFastestInterval()` method. In this app, use 5 seconds (5000 milliseconds)

- **Priority:** Use this parameter with one of the [priority constants](#) to specify a balance between power consumption and accuracy. (Greater accuracy requires greater power consumption.) For this app, use the `PRIORITY_HIGH_ACCURACY` constant to prioritize accuracy.

To create the `LocationRequest` object:

1. Create a method called `getLocationRequest()` that takes no arguments and returns a `LocationRequest`.
2. Set the interval, fastest interval, and priority parameters.

```
private LocationRequest getLocationRequest() {  
    LocationRequest locationRequest = new LocationRequest();  
    locationRequest.setInterval(10000);  
    locationRequest.setFastestInterval(5000);  
    locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);  
    return locationRequest;  
}
```

## 4.3 Create the LocationCallback object

When your app requests a location update, the fused location provider invokes the `LocationCallback.onLocationResult()` callback method. The incoming argument contains a list `Location` objects containing the location's latitude and longitude.

To create a `LocationCallback` object:

1. At the bottom of `onCreate()`, create a new `LocationCallback` object and assign it to a member variable called `mLocationCallback`.
2. Override the `onLocationResult()` method.

```
mLocationCallback = new LocationCallback() {  
    @Override  
    public void onLocationResult(LocationResult locationResult) {  
    }  
};
```

## 4.4 Request location updates

You now have the required `LocationRequest` and `LocationCallback` objects to request periodic location updates. When your app receives the `LocationResult` objects in `onLocationResult()`, use the `FetchAddressTask` to reverse geocode the `Location` object into an address:

1. To request periodic location updates, replace the call to `getLastLocation()` in `startTrackingLocation()` (along with the `OnSuccessListener`) with the following method call. Pass in the `LocationRequest` and `LocationCallback`:

```
mFusedLocationClient.requestLocationUpdates  
    (getLocationRequest(), mLocationCallback,  
     null /* Looper */);
```

2. In the `stopTrackingLocation()` method, call `removeLocationUpdates()` on `mFusedLocationClient`. Pass in the `LocationCallback` object.

```
mFusedLocationClient.removeLocationUpdates(mLocationCallback);
```

3. In the `onLocationResult()` callback, check `mTrackingLocation`. If `mTrackingLocation` is `true`, execute `FetchAddressTask()`, and use the `LocationResult.getLastLocation()` method to obtain the most recent `Location` object.

```
@Override  
public void onLocationResult(LocationResult locationResult) {  
    // If tracking is turned on, reverse geocode into an address  
    if (mTrackingLocation) {  
        new FetchAddressTask(MainActivity.this, MainActivity.this)  
            .execute(locationResult.getLastLocation());  
    }  
}
```

4. In `onTaskComplete()`, where the UI is updated, wrap the code in an `if` statement that checks the `mTrackingLocation` boolean. If the user turns off the location updates while the `AsyncTask` is running, the results are not displayed to the `TextView`.
5. Run the app. Your app tracks your location, updating the location approximately every ten seconds.

Testing the location-update functionality on an emulator can be tough: the UI will say "Loading" until you send a new location, and seeing the timing of the set interval is impossible. You can use a [GPX file](#) to simulate different locations over time. For testing, you can use the `places_gps_data.gpx` GPX file, which contains several locations:

1. Download the `places_gps_data.gpx` file.
2. Open your emulator, click the ... icon at the bottom of this vertical settings menu, and select the **Location** tab.
3. Click **Load GPX/KML** and select the downloaded file.

4. Change the duration of each item to 10 seconds, and click the play button. If you start tracking when the GPX file is playing, you see a changing address displayed in the UI.

Right now, the app continues to request location updates until the user clicks the button, or until the `Activity` is destroyed. To conserve power, stop location updates when your `Activity` is not in focus (in the [paused state](#)) and resume location updates when the `Activity` regains focus:

1. Override the `Activity` object's `onResume()` and `onPause()` methods.
2. In `onResume()`, check `mTrackingLocation`. If `mTrackingLocation` is `true`, call `startTrackingLocation()`.
3. In `onPause()`, check `mTrackingLocation`. If `mTrackingLocation` is `true`, call `stopTrackingLocation()` but set `mTrackingLocation` to `true` so the app continues tracking the location when it resumes.
4. Run the app and turn on location tracking. Exiting the app stops the location updates when the activity is not visible.

## 4.5 Make the tracking state persistent

If you run the app and rotate the device, the app resets to its initial state. The `mTrackingLocation` boolean is not persistent across configuration changes, and it defaults to `false` when the `Activity` is recreated. This means the UI defaults to the initial state.

In this step, you use the saved instance state to make `mTrackingLocation` persistent so that the app continues to track location when there is a configuration change.

1. Override the `Activity` object's `onSaveInstanceState()` method.
2. Create a string constant called `TRACKING_LOCATION_KEY`. You use this constant as a key for the `mTrackingLocation` boolean.
3. In `onSaveInstanceState()`, save the state of the `mTrackingLocation` boolean by using the `putBoolean()` method:

```
@Override  
protected void onSaveInstanceState(Bundle outState) {  
    outState.putBoolean(TRACKING_LOCATION_KEY, mTrackingLocation);  
    super.onSaveInstanceState(outState);  
}
```

4. In `onCreate()`, restore the `mTrackingLocation` variable *before* you create the `LocationCallback` instance (because the code checks for the `mTrackingLocation` boolean before starting the `FetchAddressTask`):

```
if (savedInstanceState != null) {  
    mTrackingLocation = savedInstanceState.getBoolean(  
        TRACKING_LOCATION_KEY);  
}
```

- Run the app and start location tracking. Rotate the device. A new `FetchAddressTask` is triggered, and the device continues to track the location.

## Coding challenge

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

**Challenge:** Extend the location `TextView` to include the distance traveled from the first location obtained. (See the `distanceTo ()` method.)

## Solution code

[WalkMyAndroid-Solution](#)

## Summary

- Location information is available through the `FusedLocationProviderClient`.
- Using location services requires location permissions.
- Location permissions are categorized as "dangerous permissions," so you must include them in the manifest *and* request them at runtime.
- Use the `getLastLocation()` method to obtain the device's last known location from the `FusedLocationProviderClient`.
- The process of converting a set of coordinates (longitude and latitude) into a physical address is called *reverse geocoding*. Reverse geocoding is available through the `Geocoder` class' `getFromLocation()` method.
- The `getFromLocation()` method is synchronous and may take a while to complete, so you should not use it on the main thread.
- Use a `LocationRequest` to specify the accuracy and frequency requirements of your location updates.
- Provided that the device settings are appropriate, use the `FusedLocationProviderClient` to request periodic location updates with `requestLocationUpdates()`.
- Stop the location updates with the `FusedLocationProviderClient` `removeLocationUpdates()` method.

## Related concept

The related concept documentation is in [7.1 C: Location services](#).

## Learn more

Android developer documentation:

- [Making Your App Location-Aware](#)
- [Location](#)
- [Geocoder](#)
- [FusedLocationProviderClient](#)

# 8.1: Using the Places API

## Contents:

- [Introduction](#)
- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Sign up and obtain API keys](#)
- [Task 2. Get details on the current place](#)
- [Task 3. Add the place-picker UI](#)
- [Coding challenge](#)
- [Solution code](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

The Location APIs can provide timely and accurate location information, but these APIs only return a set of geographic coordinates. In [7.1 Using the device location](#), you learned how to make geographic coordinates more useful by reverse geocoding them into physical addresses.

But what if you want to know more about a location, like the type of place it is? In this practical, you use the Google Places API for Android to obtain details about the device's current location. You also learn about the place picker and the place-autocomplete APIs. The place picker and autocomplete let users search for places rather than having your app detect the device's current place.

## What you should already KNOW

You should be familiar with:

- Creating, building, and running apps in Android Studio.
- The activity lifecycle.
- Including external libraries in your `build.gradle` file.
- Creating responsive layouts with `ConstraintLayout`.

## What you will LEARN

You will learn how to:

- Get details about the user's current place.
- Launch the place-picker UI so the user can select a place.
- Include a place-autocomplete fragment to allow the user to search for places.

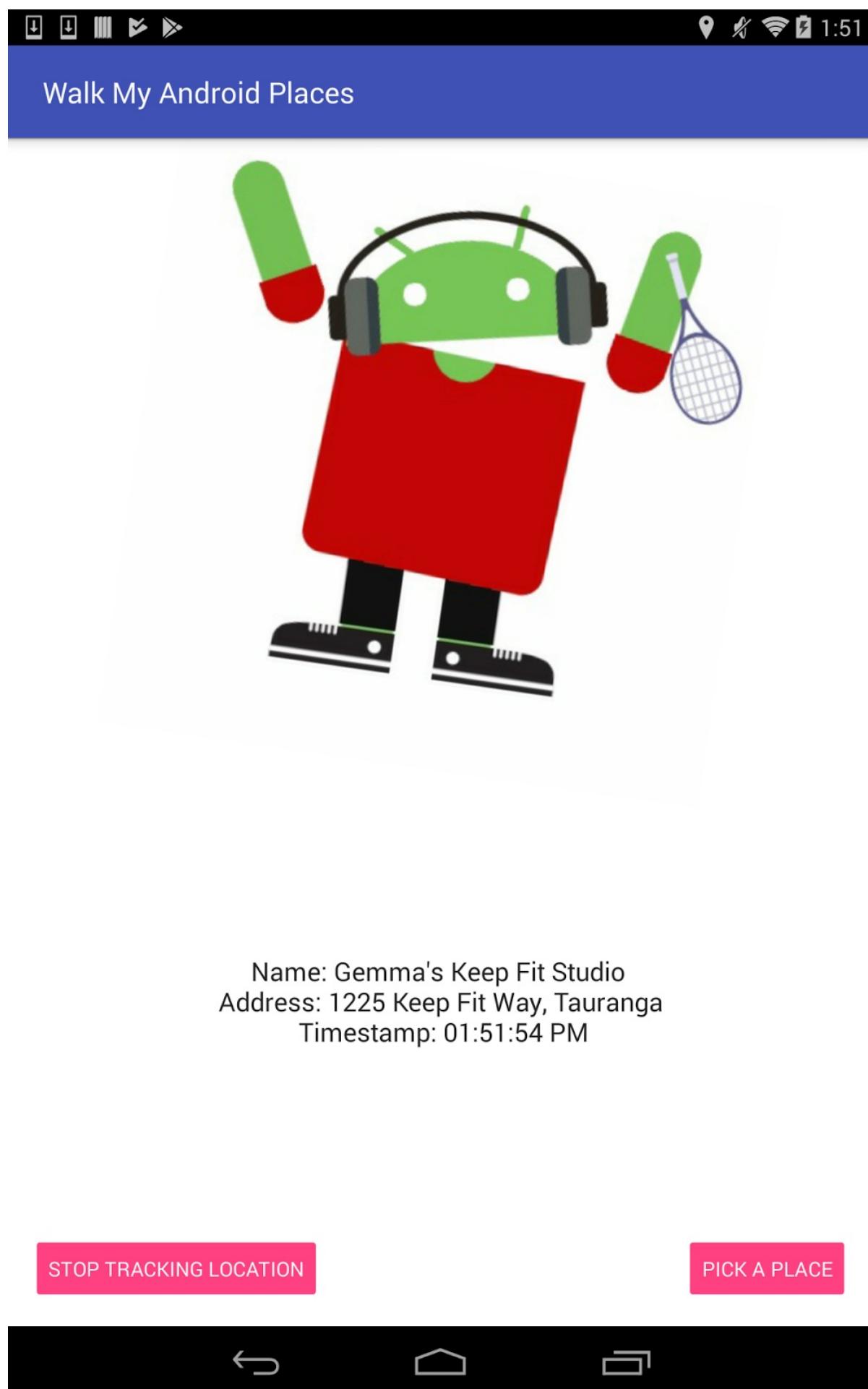
## What you will DO

- Get an API key from the Google API Console and register the key to your app.
- Get the name of the place where the device is located.
- If the place is a school, gym, restaurant, or library, change an image in the UI to reflect the place type.
- Add a button to allow the user to select a place.
- Add a search bar that autocompletes a user's search for a place.

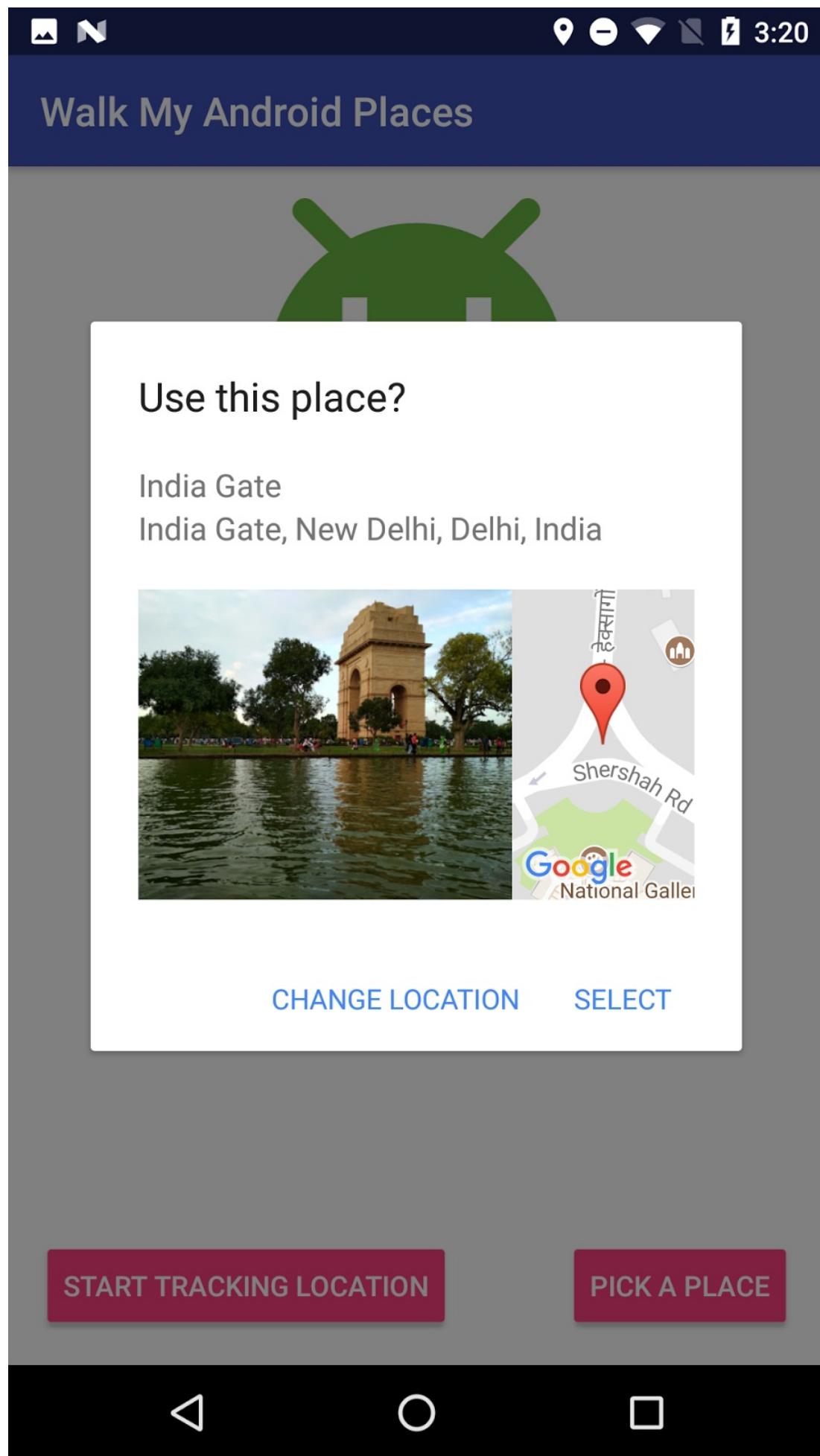
## App overview

The app for this practical extends the WalkMyAndroid app from the previous practical in two ways:

1. Get details about the current location of the device, including the [place type](#) and [place name](#). Display the name in the label `TextView` and change the Android robot image to reflect the place type.



2. Add a **Pick a Place** button that launches the [place-picker UI](#), allowing the user to select a place.



# Task 1. Sign up and obtain API keys

In this task you set up the starter app, [WalkMyAndroidPlaces-Starter](#), with an API key. Every app that uses the Google Places API for Android must have an API key. The key is linked to the app by the app's package name and by a digital certificate that's unique to the app.

To set up an API key in your app, you need to do the following:

- Get information about your app's digital certificate.
- Get an API key. To do this, you register a project in the [Google API Console](#) and add the Google Places API for Android as a service for the project.
- Add the key to your app by adding a `meta-data` element to your `AndroidManifest.xml` file.

## 1.1 Get your app's certificate information

The API key is based on a short form of your app's digital certificate, known as the certificate's *SHA-1 fingerprint*. This fingerprint uniquely identifies the app, and identifies you as the app's owner, for the lifetime of the app.

You might have two certificates:

- A *debug certificate*, which is the certificate you use for this practical. The Android SDK tools generate a debug certificate when you do a debug build. Don't attempt to publish an app that's signed with a debug certificate. The debug certificate is described in more detail in [Sign your debug build](#).
- A *release certificate*, which you don't need for this practical. The Android SDK tools generate a release certificate when you do a release build. You can also generate a release certificate using the `keytool` utility. Use the release certificate when you're ready to release your app to the world. For information about release certificates, see [Signup and API Keys](#).

For this practical, make sure that you use the debug certificate.

To view the debug certificate's fingerprint:

1. Locate your debug `keystore` file, which is named `debug.keystore`. By default, the file is stored in the same directory as your Android Virtual Device (AVD) files:

- macOS and Linux: `~/.android/`
- Windows Vista and Windows 7: `c:\Users\your_user_name.android\`

The file is created the first time you build your project.

## 2. List the SHA-1 fingerprint:

- For Linux or macOS, open a terminal window and enter the following:

```
keytool -list -v -keystore ~/.android/debug.keystore -alias androiddebugkey -storepass android -keypass android
```

- For Windows Vista and Windows 7, run:

```
keytool -list -v -keystore "%USERPROFILE%\.android\debug.keystore"\ -alias androiddebugkey -storepass android -keypass android
```

You should see output similar to the following:

```
Alias name: androiddebugkey
Creation date: Jan 01, 2013
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 4aa9b300
Valid from: Mon Jan 01 08:04:04 UTC 2013 until: Mon Jan 01 18:04:04 PST 2033
Certificate fingerprints:
MD5: AE:9F:95:D0:A6:86:89:BC:A8:70:BA:34:FF:6A:AC:F9
SHA1: BB:0D:AC:44:D3:21:E1:41:07:71:9C:62:90:AF:A4:66:6E:44:5D:95
Signature algorithm name: SHA1withRSA
Version: 3
```

The line that begins with `SHA1` contains the certificate's SHA-1 fingerprint. The fingerprint is the sequence of 20 two-digit hexadecimal numbers separated by colons.

- Copy this value to your clipboard. You need it in the next set of steps.
- Download the starter code for this practical, [WalkMyAndroidPlaces-Starter](#).
- Open Android Studio and open your `AndroidManifest.xml` file. Note the `package` string in the `manifest` tag. You need the package name in future steps.

**Note:** To protect your `keystore` file and key, don't enter the `storepass` or `keypass` arguments on the command line unless you're confident of your computer's security. For example, on a public computer, someone could look at your terminal window history or list of running processes, get the password, and have write-access to your signing certificate. That person could modify your app or replace your app with their own.

## 1.2 Get an API key from the Google API Console

1. Go to the [Google API Console](#).
2. Create or select a project. You can reuse the same project and API key for multiple apps.
3. From the Dashboard page, click **ENABLED APIs AND SERVICES**. The API Library opens.
4. Search for "Places" and select **Google Places API for Android**.
5. Click **ENABLE**.
6. If a warning appears telling you to create credentials, click **Create credentials**. Otherwise, open the Credentials page and click **Create credentials**.
7. If you see a **Find out what kind of credentials you need** step, skip the prompts. Click directly on the **API key** link.
8. Name the key whatever you like.

Restrict the key to Android apps:

1. Follow the prompts to restrict the key to **Android apps**.
2. Click **Add package name and fingerprint**.
3. Add your app's SHA-1 fingerprint to the key. (You copied this fingerprint to your clipboard in [1.1 Get your app's certificate information](#).) Also enter the package name from Android Studio.

For example:

```
BB:0D:AC:74:D3:21:E1:43:67:71:9B:62:91:AF:A1:66:6E:44:5D:75  
com.example.android.walkmyandroidplaces
```

4. Save your changes.

On the Credentials page, your new Android-restricted API key appears in the list of API keys for your project. An API key is a string of characters, something like this:

```
AIzaSyBdV1-cTCSwYZrZ95SuvNw0dbMuDt1KG0
```

5. Copy the API key to your clipboard. You'll paste the key into your app manifest in the next step.

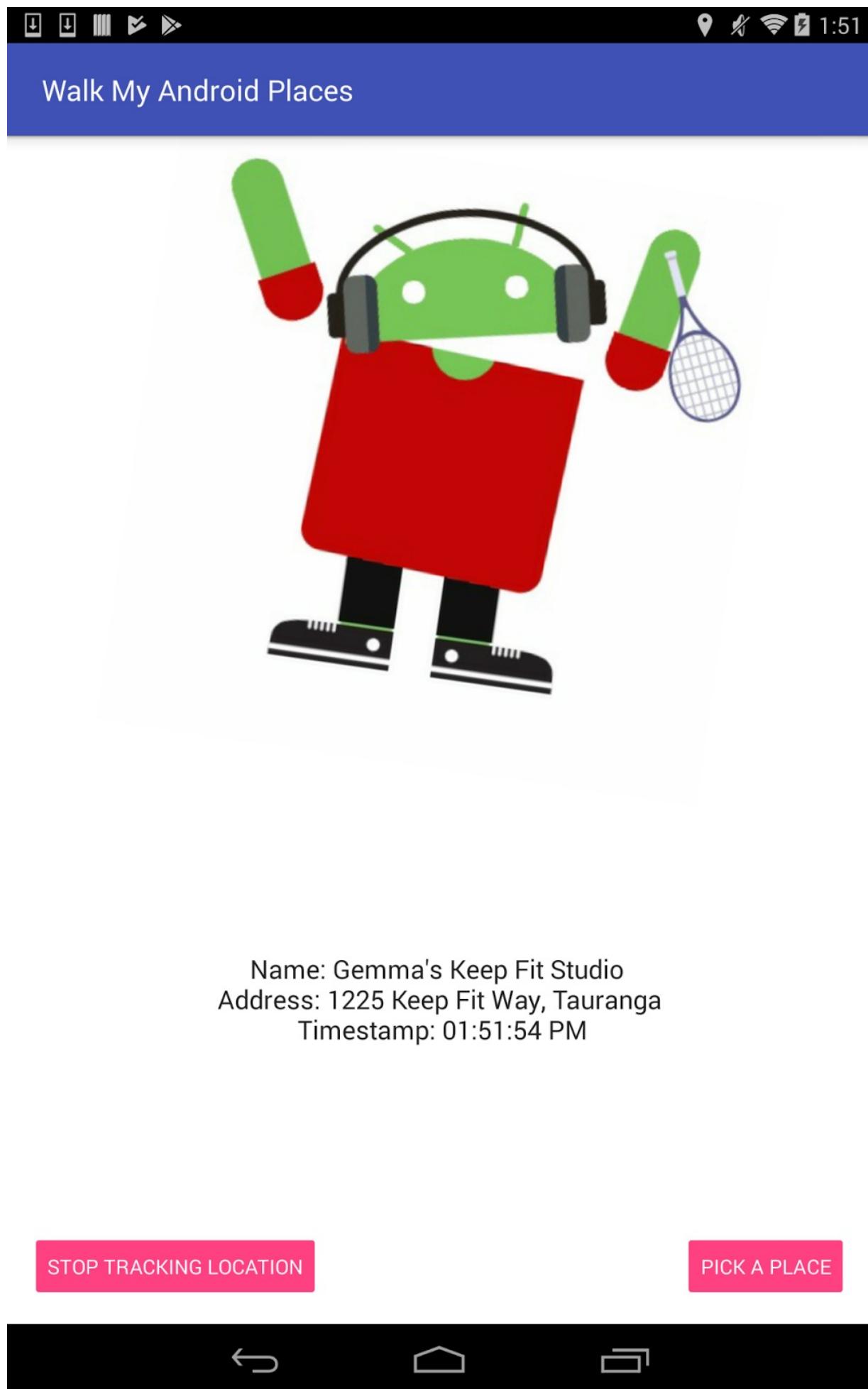
## 1.3 Add the key to the manifest

Add your API key to your `AndroidManifest.xml` file as shown in the following code. Replace `YOUR_API_KEY` with your own API key:

```
<application>
    ...
    <meta-data
        android:name="com.google.android.geo.API_KEY"
        android:value="YOUR_API_KEY"/>
    ...
```

## Task 2. Get details on the current place

Now that you have your API key set up, you're ready to start using the Places API. In this task, you use the `PlaceDetectionClient.getCurrentPlace()` method to obtain the place name and place type for the device's current location. You use these place details to enhance the WalkMyAndroidPlaces UI.



## 2.1 Add the Places API to your project

To use the Places API, you need to add it to the app-level `build.gradle` file. You also need to connect to the API using the `GoogleApiClient` class.

1. Add the following statement to your app-level `build.gradle` file. Replace `xx.x.x` with the appropriate support library version. For the latest version number, see [Add Google Play Services to Your Project](#).

```
compile 'com.google.android.gms:play-services-places:xx.x.x'
```

2. In the `MainActivity`, in the `onCreate()` method, initialize a `PlaceDetectionClient` object. You use this object to get information about the device's current location.

```
mPlaceDetectionClient = Places.getPlaceDetectionClient(this, null);
```

**Note:** To use the `PlaceDetectionClient` interface, your app needs the `ACCESS_FINE_LOCATION` permission. You don't need to add this permission, because the `WalkMyAndroidPlaces-Starter` code includes it.

## 2.2 Get the place name

In this step, you extend the `WalkMyAndroidPlaces` app to show the place name associated with the current device location.

1. In the `MainActivity`, create a `String` member variable called `mLastPlaceName`. This member variable will hold the name of the device's most probable location.
2. In `strings.xml`, modify the `address_text` string to include the place name as an additional variable:

```
<string name="address_text">"Name: %1$s \n Address: %2$s \n Timestamp: %3$tr"</string>
```

3. In the `onClick()` method for the **Start Tracking Location** button, add another argument to the `setText()` call. Pass in the `loading` string so that the `TextView` label shows "Loading..." for the `Name` and `Address` lines:

```
mLocationTextView.setText(getString(R.string.address_text,
    getString(R.string.loading), // Name
    getString(R.string.loading), // Address
    new Date())); // Timestamp
```

When the `AsyncTask` returns an `Address`, the `onTaskCompleted()` method is called. In

the `onTaskComplete()` method, obtain the current place name and update the `TextView`. To get the current place name, call `PlaceDetectionClient.getCurrentPlace()`.

The `getCurrentPlace()` method returns a `Task` object. A `Task` object represents an asynchronous operation and contains a parameterized type that's returned when the operation completes in its `onComplete()` callback. In this case, the parameterized type is a `PlaceLikelihoodBufferResponse`.

The returned `PlaceLikelihoodBufferResponse` instance is a list of `Place` objects, each with a "likelihood" value between 0 and 1. The likelihood value indicates the likelihood that the device is at that place. The strategy shown below is to use a loop to determine the place that has the highest likelihood, then display that place name.

1. In the `onTaskCompleted()` method, call `getCurrentPlace()` on your `PlaceDetectionClientApi` instance. Store the result in a local variable. Because you are interested in all places, you can pass in `null` for the `PlaceFilter`:

```
Task<PlaceLikelihoodBufferResponse> placeResult =  
    mPlaceDetectionClient.getCurrentPlace(null);
```

2. The `getCurrentPlace()` method call is underlined in Android Studio, because the method may throw a `SecurityException` if you don't have the right location permissions. Have the `onTaskCompleted()` method throw a `SecurityException` to remove the warning.
3. Add an `OnCompleteListener` to the `placeResult`:

```
placeResult.addOnCompleteListener  
    (new OnCompleteListener<PlaceLikelihoodBufferResponse>() {  
        @Override  
        public void onComplete(@NonNull  
            Task<PlaceLikelihoodBufferResponse> task) {  
    }});
```

4. Create an if/else statement to check whether the `Task` was successful:

```
if (task.isSuccessful()) {  
} else {  
{
```

5. If the `Task` was successful, call `getResult()` on the task to obtain the `PlaceLikelihoodBufferResponse`. Initialize an integer to hold the maximum value. Initialize a `Place` to hold the highest likelihood `Place` object.

```

if (task.isSuccessful()) {
    PlaceLikelihoodBufferResponse likelyPlaces = task.getResult();
    float maxLikelihood = 0;
    Place currentPlace = null;
}

```

6. Iterate over each `PlaceLikelihood` object and check whether it has the highest likelihood so far. If it does, update the `maxLikelihood` and `currentPlace` objects:

```

if (task.isSuccessful()) {
    PlaceLikelihoodBufferResponse likelyPlaces = task.getResult();
    float maxLikelihood = 0;
    Place currentPlace = null;
    for (PlaceLikelihood placeLikelihood : likelyPlaces) {
        if (maxLikelihood < placeLikelihood.getLikelihood()) {
            maxLikelihood = placeLikelihood.getLikelihood();
            currentPlace = placeLikelihood.getPlace();
        }
    }
}

```

7. If the `currentPlace` is not `null`, update the `TextView` with the result. The following code should all be within the `if (task.isSuccessful())` loop:

```

if (currentPlace != null) {
    mLocationTextView.setText(
        getString(R.string.address_text,
        currentPlace.getName(), result //This is the address from the AsyncTask,
        System.currentTimeMillis()));
}

```

8. After you use the place information, release the buffer:

```
likelyPlaces.release();
```

9. In the `else` block for the case where the `task` is not successful, show an error message instead of a place name:

```

else {
    mLocationTextView.setText(
        getString(R.string.address_text,
        "No Place name found!",
        result, System.currentTimeMillis()));
}

```

10. Run the app. You see the place name along with the address in the label `TextView`.

## 2.3 Get the place type

The `Place` object you obtained from the `PlaceLikelihood` object contains a lot more than just the name of the place. The object can also include the place type, rating, price level, website URL, and more. (For a list of all the fields, see the [Place reference](#).)

In this step, you change the image of the Android robot to reflect the [place type](#) of the current location. The starter code includes a bitmap image for a "plain" Android robot, plus images for four other Android robots, one for each of these place types: school, gym, restaurant, and library.

If you want to add support for other place types, use the [Androidify](#) tool to create a custom Android robot image and include the image in your app.

1. In `MainActivity`, create a `setAndroidType()` method that uses a `Place` object. Have the method assign the appropriate drawable to the `ImageView`, based on the place type:

```
private void setAndroidType(Place currentPlace) {  
    int drawableID = -1;  
    for (Integer placeType : currentPlace.getPlaceTypes()) {  
        switch (placeType) {  
            case Place.TYPE_SCHOOL:  
                drawableID = R.drawable.android_school;  
                break;  
            case Place.TYPE_GYM:  
                drawableID = R.drawable.android_gym;  
                break;  
            case Place.TYPE_RESTAURANT:  
                drawableID = R.drawable.android_restaurant;  
                break;  
            case Place.TYPE_LIBRARY:  
                drawableID = R.drawable.android_library;  
                break;  
        }  
    }  
  
    if (drawableID < 0) {  
        drawableID = R.drawable.android_plain;  
    }  
    mAndroidImageView.setImageResource(drawableID);  
}
```

**Note:** The `setAndroidType()` method is where you can add support for more place types. All you need to do is add another `case` to the `switch` statement and select the appropriate drawable. For a list of supported place types, see [Place Types](#).

2. In the `onComplete()` callback where you obtain the current `Place` object, call `setAndroidType()`. Pass in the `currentPlace` object.

3. Run your app. Unless you happen to be in one of the supported place types, you don't see any difference in the Android robot image. To get around this, run the app on an emulator and follow the steps below to set up fake locations.

## How to test location-based features on an emulator

Testing location-based features on an emulator can be challenging. The `FusedLocationProviderClient` and the Places API have to use a location that you provide through the emulator settings.

To simulate a location, use a GPX file, which provides a set of GPS coordinates over time:

1. Download the [WalkMyAndroidPlaces-gpx](#) file. The file contains five locations. The first location doesn't correspond to any of the supported place types. The other four locations have the place types that the WalkMyAndroidPlaces app supports: school, gym, restaurant, library.
2. Start an emulator of your choice.
3. To navigate to your emulator settings, select the three dots at the bottom of the menu next to the emulator, then select the **Location** tab.
4. In the bottom right corner, click **Load GPX/KML**. Select the file you downloaded.

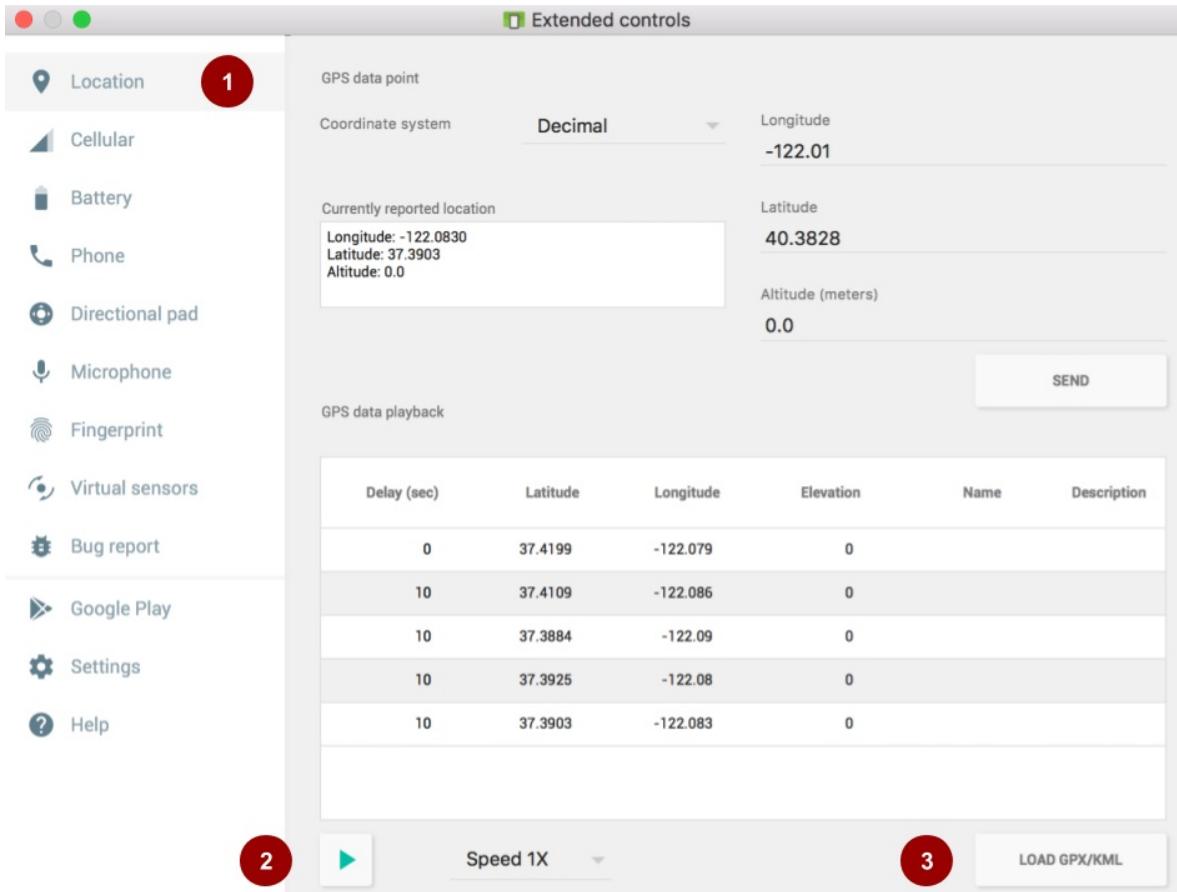
Five locations load in the GPS data playback window.

5. Notice the **Delay** column. By default, the emulator changes the location every 2 seconds. Change the delay to 10 seconds for each item except the first item, which should load immediately and have a delay of 0.

(A delay of 10 seconds makes sense because your location updates happen approximately every 10 seconds. Recall the `LocationRequest` object, in which the interval is set to 10,000 milliseconds, or 10 seconds.)

6. Run the WalkMyAndroid app on the emulator to start tracking the device location.
7. Use the play button in the bottom left corner of the emulator **Location** tab to deliver the GPX file's location information to your app. The location `TextView` and the Android robot image should update every 10 seconds to reflect the new locations as they are "played" by the GPX file!

The screenshot below shows the **Location** tab (1) for emulator location settings, the GPS data playback button (2), and the **Load GPX/KML** button (3).

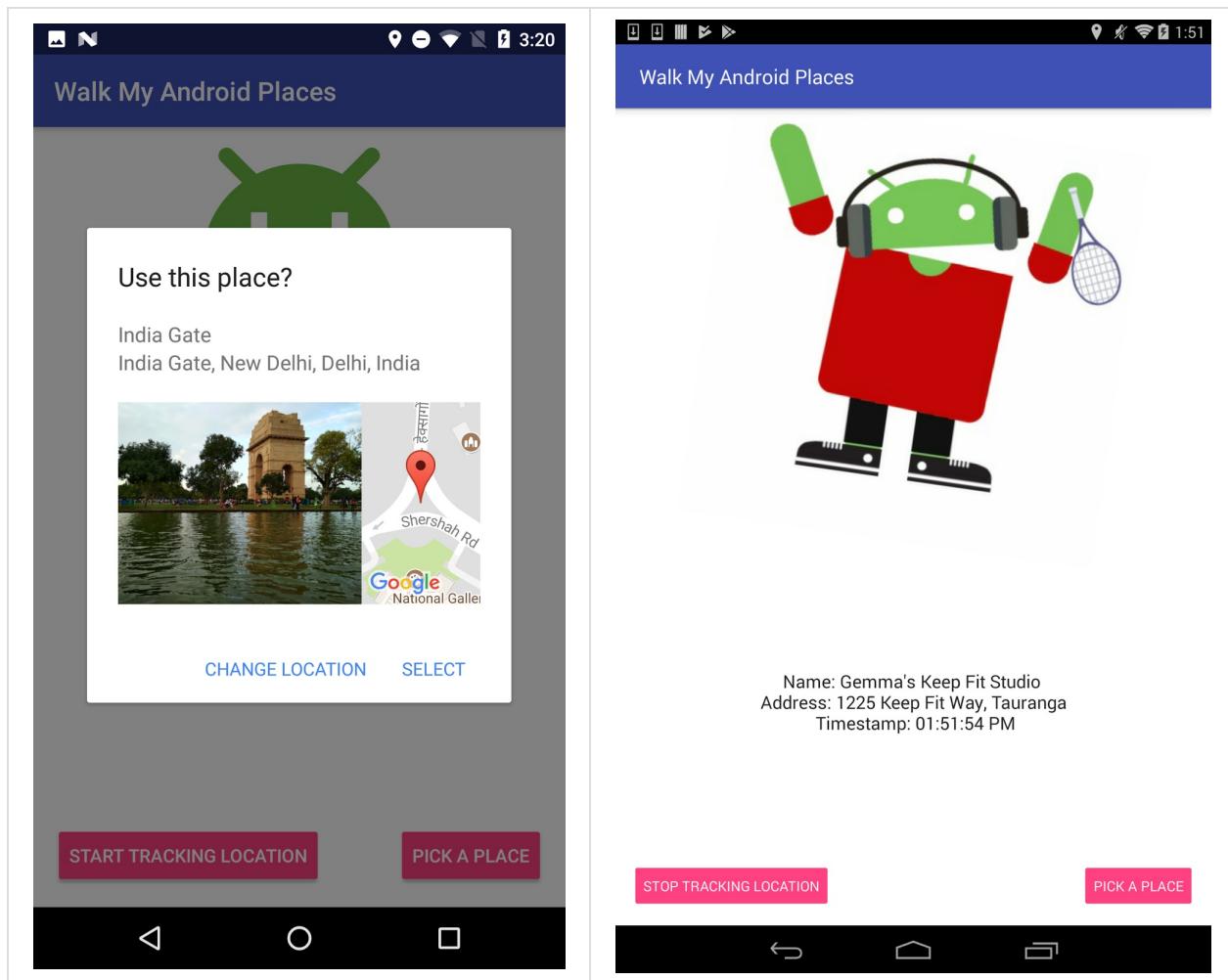


## Task 3. Add the place-picker UI

At this point, the `WalkMyAndroidPlaces` app only looks for places in the device's current location, but there are many use cases where you want the user to select a location from a map. For example, if you create an app to help users decide on a restaurant, you want to show restaurants in the area that the user selects, not just restaurants in the user's current location.

Having the user select a location from a map seems complicated: you need a map with places of interest already on it, and you need a way for the user to search for a place and select a place. Fortunately, the Place API includes the [place picker](#), a UI that greatly simplifies the work.

In this task, you add a button that launches the place-picker UI. The place-picker UI lets the user select a place, and it displays the place information in the UI, as before. (It displays the relevant Android robot image, if available, and it displays the place name, address, and update time.)



### 3.1 Add a PlacePicker button

The [place-picker UI](#) is a dialog where the user can search for a place and select a place. To add the place picker to your app, use the `PlacePicker.IntentBuilder` intent to launch a special `Activity`. Get the result in your activity's `onActivityResult()` method:

1. Add a button next to the **Start Tracking Location** button. Use "Pick a Place" as the button's text in both the portrait and the landscape layout files.
2. Add a click handler that executes the following code to start the `PlacePicker`. Create an arbitrary constant called `REQUEST_PICK_PLACE` that you'll use later to obtain the result. (This constant should be different from your permission-check integer.)

```

private static final int REQUEST_PICK_PLACE = 2;

PlacePicker.IntentBuilder builder = new PlacePicker.IntentBuilder();
try {
    startActivityForResult(builder.build(MainActivity.this), REQUEST_PICK_PLACE);
} catch (GooglePlayServicesRepairableException | GooglePlayServicesNotAvailableException e) {
    e.printStackTrace();
}

```

- Run your app. When you click **Pick a Place**, a `PlacePicker` dialog opens. In the dialog, you can search for and select any place.

The next step is to get whatever place the user selects in `onActivityResult()`, then update the `TextView` and Android robot image .

## 3.2 Obtain the selected place

The result from the `PlacePicker` is sent automatically to the activity's `onActivityResult()` override method. Use the passed-in `requestCode` integer to get the result. The result should match the integer you passed into the `startActivityForResult()` method.

- To override the `onActivityResult()` method, select **Code > Override Methods** in Android Studio. Find and select the `onActivityResult()` method, then click **OK**.
- In `onActivityResult()`, create an `if` statement that checks whether the `requestCode` matches your request integer.

To get the `Place` object that was selected from the `PlacePicker`, use the `PlacePicker.getPlace()` method.

- If the `resultCode` is `RESULT_OK`, get the selected place from the data `Intent` using the `PlacePicker.getPlace()` method. Pass in the application context and the data `Intent`, which the system passes into `onActivityResult()`.

If the `resultCode` isn't `RESULT_OK`, show a message that says that a place was not selected.

```

if (resultCode == RESULT_OK) {
    Place place = PlacePicker.getPlace(this, data);
} else {
    mLocationTextView.setText(R.string.no_place);
}

```

- After you get the `Place` object, call `setAndroidType()`. Pass in your obtained object.
- Update the label `TextView` with the place name and address from the `Place` object.

Set the update time to be the current time:

```
mLocationTextView.setText(  
        getString(R.string.address_text, place.getName(),  
        place.getAddress(), System.currentTimeMillis()));
```

6. Run the app. You can now use the `PlacePicker` to choose any place in the world, provided that the place exists in the Places API. To test your app's functionality, search for one of the place types for which you have an Android robot image.

**Note:** When the user selects a location using the `PlacePicker`, this data is not persisted using the `savedInstanceState`. For this reason, when you rotate the device, the app resets to the initial state.

## Coding challenge

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

**Challenge:** Add a `place autocomplete` search dialog UI element to your `Activity`. The place autocomplete dialog lets the user search for a place without launching the `PlacePicker`.

## Solution code

[WalkMyAndroidPlaces-Solution](#)

## Summary

- To use the Google Places API for Android, you must create an API key that's restricted to Android apps. You do this in the [Google API Console](#). In your project in the API Console, you also need to enable the Google Places API for Android.
- Include the API key in a metadata tag in your `AndroidManifest.xml` file.
- The `Place` object contains information about a specific geographic location, including the place name, address, coordinates, and more.
- Use the `PlaceDetectionClient.getCurrentPlace()` method to get information about the device's current location.
- `PlaceDetectionClient.getCurrentPlace()` returns a `PlaceLikelihoodBuffer` in a `Task`.
- The `PlaceLikelihoodBuffer` contains a list of `PlaceLikelihood` objects that represent likely places. For each place, the result includes the likelihood that the place is the right one.

- The Places API includes the `PlacePicker`, which you use to include the place-picker UI in your app. The place-picker UI is a dialog that lets the user search for places and select places.
- Launch the `PlacePicker` using `startActivityForResult()`. Pass in an `Intent` created with `PlacePicker.IntentBuilder()`.
- Retrieve the selected place in `onActivityResult()` by calling `PlacePicker.getPlace()`. Pass in the activity context and the data `Intent`.

## Related concept

The related concept documentation is in [8.1: Places API](#).

## Learn more

Android developer documentation:

- [Set Up Google Play Services](#)
- [Places API for Android](#)
- [Place Picker](#)
- [Current Place](#)
- [Place](#)
- [PlaceDetectionClient](#)

# 9.1: Adding a Google Map to your app

## Contents:

- [Introduction](#)
- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Set up the project and get an API Key](#)
- [Task 2. Add map types and markers](#)
- [Task 3. Style your map](#)
- [Task 4. Enable location tracking and Street View](#)
- [Coding challenge](#)
- [Solution code](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

Building apps with Google Maps allows you to add features to your app such as satellite imagery, robust UI controls, location tracking, and location markers. You can add value to the standard Google Maps by showing information from your own data set, such as the locations of well-known fishing or climbing areas. You can also create games tied to the real world, like Pokemon Go.

In this practical, you create a Google Maps app called `wander`.

## What you should already KNOW

You should be familiar with:

- Basic functionality of Google Maps.
- Runtime permissions.
- Creating, building, and running apps in Android Studio.
- Including external libraries in your `build.gradle` file.

## What you will LEARN

You will learn how to:

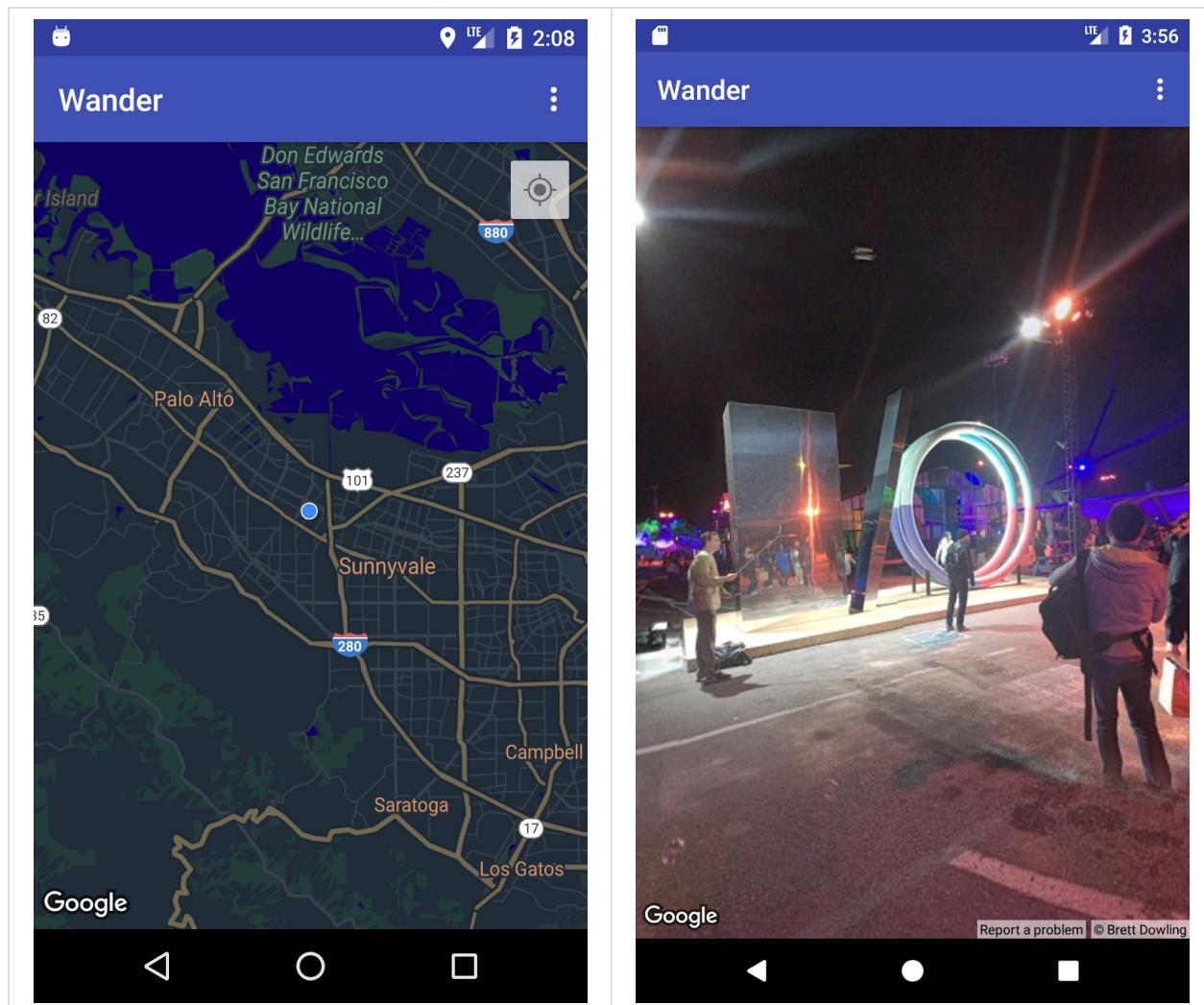
- Integrate a Google Map in your app.
- Display different map types.
- Style the Google Map.
- Add markers to your map.
- Enable the user to place a marker on a point of interest (POI).
- Enable location tracking.
- Enable Google Street View.

## What you will DO

- Get an API key from the Google API Console and register the key to your app.
- Create the `wander` app, which has an embedded Google Map.
- Add custom features to your app such as markers, styling, and location tracking.
- Enable location tracking and Street View in your app.

## App overview

In this practical you create the `wander` app, which is a styled Google Map. The `wander` app allows you to drop markers onto locations, see your location in real time, and look at Street View panoramas.



## Task 1. Set up the project and get an API Key

The Google Maps API, like the Places API, requires an API key. To obtain the API key, you register your project in the Google API Console. The API key is tied to a digital certificate that links the app to its author. For more about using digital certificates and signing your app, see [Sign Your App](#).

In this practical, you use the API key for the debug certificate. The debug certificate is insecure by design, as described in [Sign your debug build](#). Published Android apps that use the Google Maps API require a second API key: the key for the release certificate. For more information about obtaining a release certificate, see [Get API Key](#).

Android Studio includes a Google Maps Activity template, which generates helpful template code. The template code includes a `google_maps_api.xml` file containing a link that simplifies obtaining an API key.

**Note:** If you wish to build the Activity without using the template, follow the steps in the [API key guide](#) to obtain the API key without using the link in the template.

## 1.1 Create the Wander project with the Maps template

1. Create a new Android Studio project.
2. Name the new app "Wander". Accept the defaults until you get to the **Add an Activity** page.
3. Select the **Google Maps Activity** template.
4. Leave the default **Activity Name** and **Layout Name**.
5. Change the **Title** to "Wander" and click **Finish**.

Android Studio creates several maps-related additional files:

- `google_maps_api.xml`

You use this configuration file to hold your API key. The template generates two `google_maps_api.xml` files: one for debug and one for release. The file for the API key for the debug certificate is located in `src/debug/res/values`. The file for the API key for the release certificate is located in `src/release/res/values`. In this practical we only use the debug certificate.

- `activity_maps.xml`

This layout file contains a single fragment that fills the entire screen. The `SupportMapFragment` class is a subclass of the `Fragment` class. You can include `SupportMapFragment` in a layout file using a `<fragment>` tag in any `ViewGroup`, with an additional attribute:

```
    android:name="com.google.android.gms.maps.SupportMapFragment"
```

- `MapsActivity.java`

The `MapsActivity.java` file instantiates the `SupportMapFragment` class and uses the class's `getMapAsync()` method to prepare the Google Map. The activity that contains the `SupportMapFragment` must implement the `OnMapReadyCallback` interface and that interface's `onMapReady()` method. The `getMapAsync()` method returns a `GoogleMap` object, signifying that the map is loaded.

**Note:** If you test the `wander` app on an emulator, use a system image that includes Google APIs and Google Play. Select an image that shows **Google Play** in the **Target** column of the virtual-devices list.

Run the app and notice that the map fails to load. If you look in the logs, you see a message saying that your API key is not properly set up. In the next step, you obtain the API key to make the app display the map.

## 1.2 Obtain the API key

1. Open the debug version of the `google_maps_api.xml` file.

The file includes a comment with a long URL. The URL's parameters include specific information about your app.

2. Copy and paste the URL into a browser.
3. Follow the prompts to create a project in the Google API Console. Because of the parameters in the provided URL, the API Console knows to automatically enable the Google Maps Android API
4. Create an API key and click **Restrict Key** to restrict the key's use to Android apps. The generated API key should start with `AIza`.
5. In the `google_maps_api.xml` file, paste the key into the `google_maps_key` string where it says `YOUR_KEY_HERE`.
6. Run your app. You have an embedded map in your activity, with a marker set in Sydney, Australia. (The Sydney marker is part of the template, and you change it later.)

**Note:** The API key may take up to 5 minutes to take effect.

## Task 2. Add map types and markers

Google Maps include several map types: normal, hybrid, satellite, terrain, and "none." In this task you add an app bar with an options menu that allows the user to change the map type. You move the map's starting location to your own home location. Then you add support for markers, which indicate single locations on a map and can include a label.

### 2.1 Add map types

The type of map that your user wants depends on the kind of information they need. When using maps for navigation in your car, it's helpful to see street names clearly. When you are hiking, you probably care more about how much you have to climb to get to the top of the mountain. In this step, you add an app bar with an options menu that allows the user to change the map type.

1. To create a new menu XML file, right-click on your `res` directory and select **New > Android Resource File**.
2. In the dialog, name the file `map_options`. Choose **Menu** for the resource type. Click **OK**.
3. Replace the code in the new file with the following code to create the map options. The "none" map type is omitted, because "none" results in the lack of any map at all.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item android:id="@+id/normal_map"
        android:title="@string/normal_map"
        app:showAsAction="never"/>
    <item android:id="@+id/hybrid_map"
        android:title="@string/hybrid_map"
        app:showAsAction="never"/>
    <item android:id="@+id/satellite_map"
        android:title="@string/satellite_map"
        app:showAsAction="never"/>
    <item android:id="@+id/terrain_map"
        android:title="@string/terrain_map"
        app:showAsAction="never"/>
</menu>
```

4. Create string resources for the `title` attributes.
5. In the `MapsActivity` file, change the class to extend the `AppCompatActivity` class instead of extending the `FragmentActivity` class. Using `AppCompatActivity` will show the app bar, and therefore it will show the menu.
6. In `MapsActivity`, override the `onCreateOptionsMenu()` method and inflate the `map_options` file:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.map_options, menu);
    return true;
}
```

7. To change the map type, use the `setMapType()` method on the `GoogleMap` object, passing in one of the map-type constants.

Override the `onOptionsItemSelected()` method. Paste the following code to change the map type when the user selects one of the menu options:

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Change the map type based on the user's selection.
        switch (item.getItemId()) {
            case R.id.normal_map:
                mMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
                return true;
            case R.id.hybrid_map:
                mMap.setMapType(GoogleMap.MAP_TYPE_HYBRID);
                return true;
            case R.id.satellite_map:
                mMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
                return true;
            case R.id.terrain_map:
                mMap.setMapType(GoogleMap.MAP_TYPE_TERRAIN);
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
}
```

8. Run the app. Use the menu in the app bar to change the map type. Notice how the map's appearance changes.

## 2.2 Move the default map location

By default, the `onMapReady()` callback includes code that places a marker in Sydney, Australia, where Google Maps was created. The default callback also animates the map to pan to Sydney. In this step, you make the map pan to your home location without placing a marker, then zoom to a level you specify.

1. In the `onMapReady()` method, remove the code that places the marker in Sydney and moves the camera.
2. Go to [www.google.com/maps](http://www.google.com/maps) in your browser and find your home.
3. Right-click on the location and select **What's here?**

Near the bottom of the screen, a small window pops up with location information, including latitude and longitude.

4. Create a new `LatLng` object called `home`. In the `LatLng` object, use the coordinates you found from Google Maps in the browser.
5. Create a `float` variable called `zoom` and set the variable to your desired initial zoom level. The following list gives you an idea of what level of detail each level of zoom shows:
  - 1 : World

- 5 : Landmass/continent
- 10 : City
- 15 : Streets
- 20 : Buildings

6. Create a `CameraUpdate` object using `CameraUpdateFactory.newLatLngZoom()`, passing in your `LatLang` object and `zoom` variable. Pan and zoom the camera by calling `moveCamera()` on the `GoogleMap` object, passing in the new `CameraUpdate` object:

```
mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(home, zoom));
```

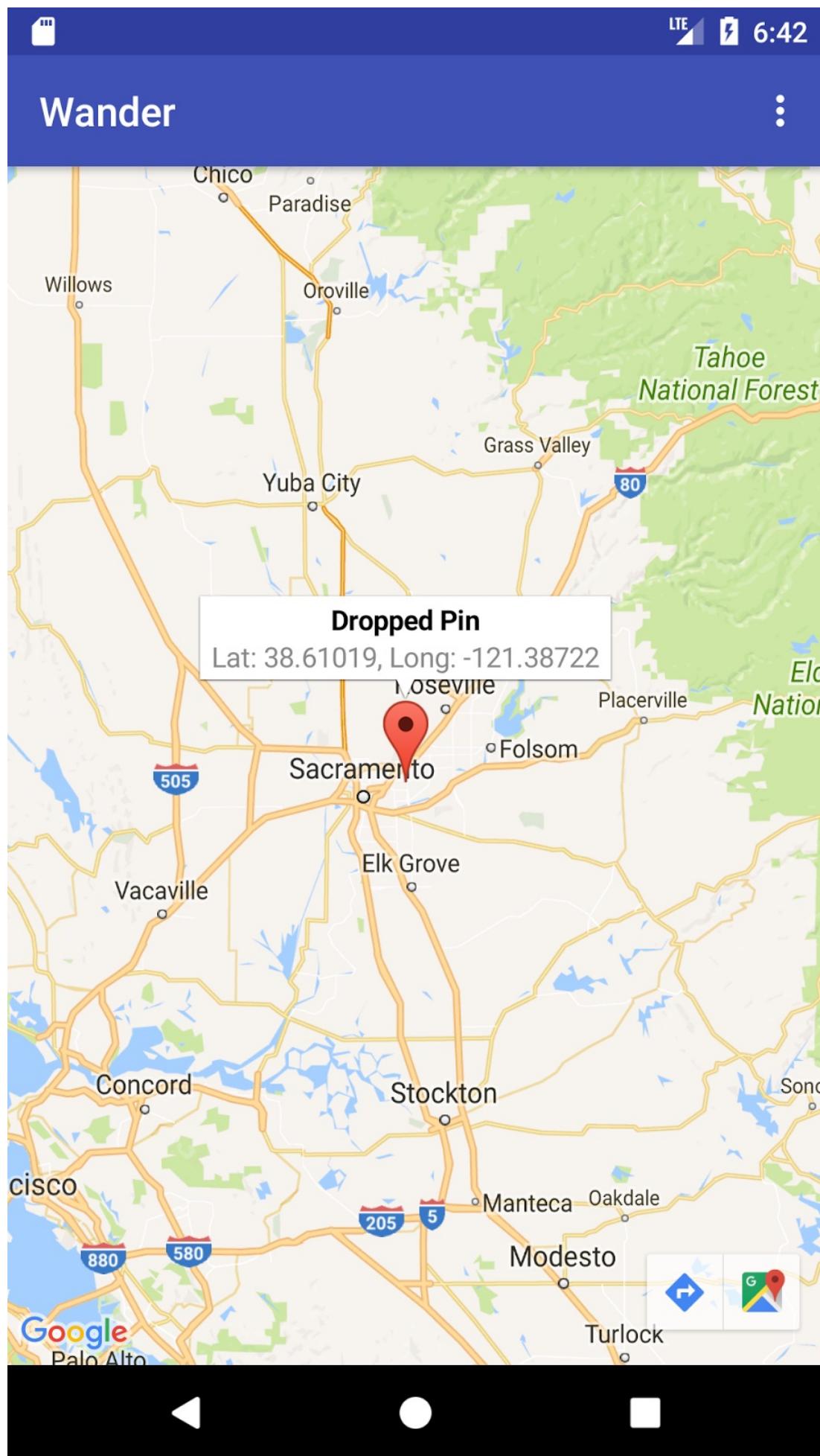
7. Run the app. The map should pan to your home and zoom into the desired level.

## 2.3 Add map markers

Google Maps can single out a location using a marker, which you create using the `Marker` class. The default marker uses the standard Google Maps icon: 

You can extend markers to show contextual information in `info windows`.

In this step, you add a marker when the user performs a touch & hold on a location on the map. You then add an `InfoWindow` that displays the coordinates of the marker when the marker is tapped.



1. Create a method stub in `MapsActivity` called `setMapLongClick()` that takes a `final GoogleMap` as an argument and returns `void`:

```
private void setMapLongClick(final GoogleMap map) {}
```

2. Use the `GoogleMap` object's `setOnMapLongClickListener()` method to place a marker where the user touches & holds. Pass in a new instance of `OnMapLongClickListener` that overrides the `onMapLongClick()` method. The incoming argument is a `LatLng` object that contains the coordinates of the location the user pressed:

```
private void setMapLongClick(final GoogleMap map) {  
    map.setOnMapLongClickListener(new GoogleMap.OnMapLongClickListener() {  
        @Override  
        public void onMapLongClick(LatLng latLng) {  
        }  
    });  
}
```

3. Inside `onMapLongClick()`, call the `addMarker()` method. Pass in a new `MarkerOptions` object with the position set to the passed-in `LatLng`:

```
map.addMarker(new MarkerOptions().position(latLng));
```

4. Call `setMapLongClick()` at the end of the `onMapReady()` method. Pass in `mMap`.
5. Run the app. Touch & hold on the map to place a marker at a location.
6. Tap the marker, which centers it on the screen.

Navigation buttons appear at the bottom-left side of the screen, allowing the user to use the Google Maps app to navigate to the marked position.

To add an info window for the marker:

1. In the `MarkerOptions` object, set the `title` field and the `snippet` field.
2. In `onMapLongClick()`, set the `title` field to "Dropped Pin." Set the `snippet` field to the location coordinates inside the `addMarker()` method.

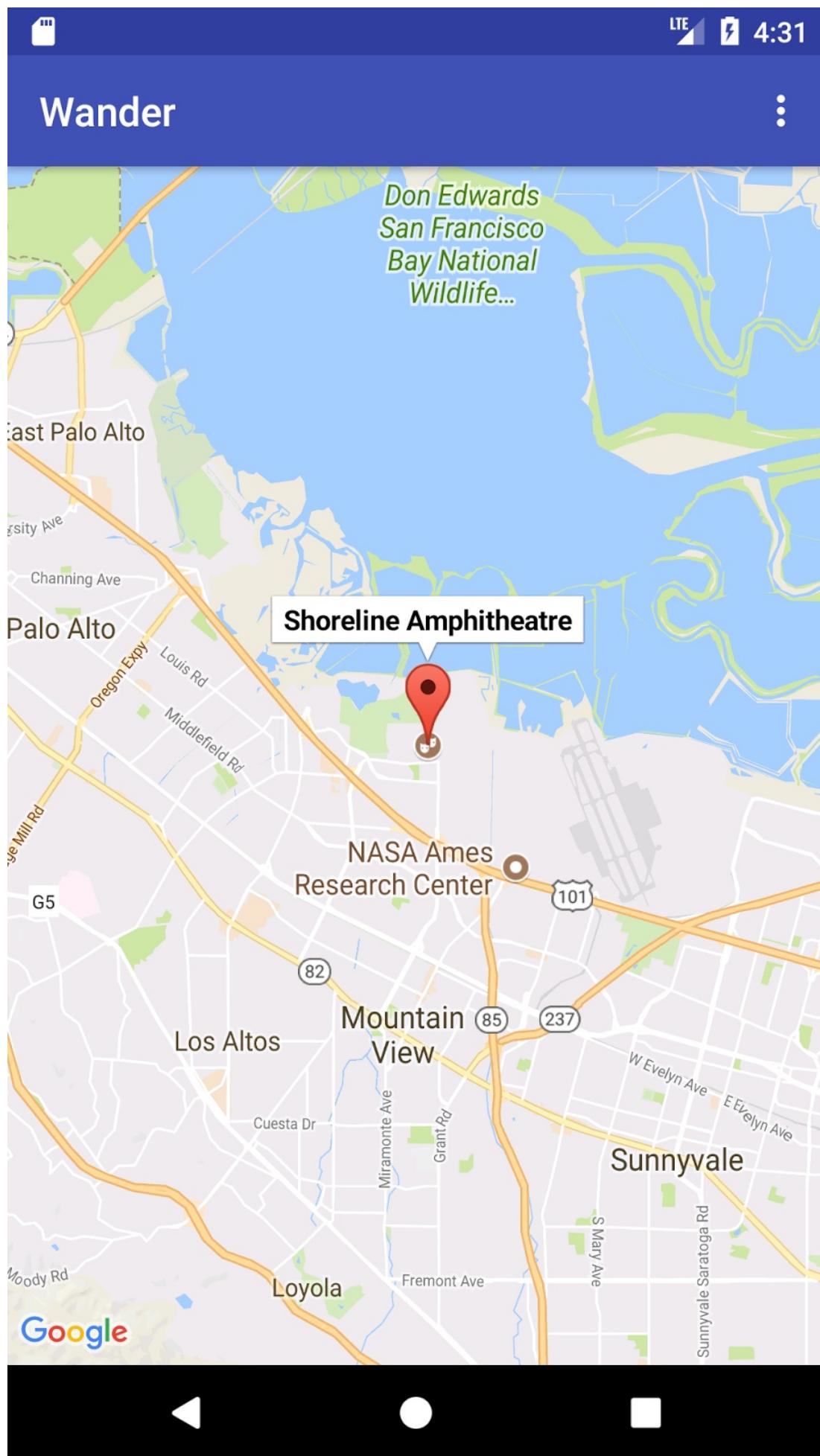
```
map.setOnMapLongClickListener(new GoogleMap.OnMapLongClickListener() {  
    @Override  
    public void onMapLongClick(LatLng latLng) {  
        String snippet = String.format(Locale.getDefault(),  
            "Lat: %1$.5f, Long: %2$.5f",  
            latLng.latitude,  
            latLng.longitude);  
  
        map.addMarker(new MarkerOptions()  
            .position(latLng)  
            .title(getString(R.string.dropped_pin))  
            .snippet(snippet));  
    }  
});
```

3. Run the app. Touch & hold on the map to drop a location marker. Tap the marker to show the info window.

## 2.4 Add POI listener

By default, points of interest (POIs) appear on the map along with their corresponding icons. POIs include parks, schools, government buildings, and more. When the map type is set to `normal`, business POIs also appear on the map. Business POIs represent businesses such as shops, restaurants, and hotels.

In this step, you add a `GoogleMap.OnPoiClickListener` to the map. This click-listener places a marker on the map immediately, instead of waiting for a touch & hold. The click-listener also displays the info window that contains the POI name.



1. Create a method stub in `MapsActivity` called `setPoiClick()` that takes a `final GoogleMap` as an argument, and returns `void`:

```
private void setPoiClick(final GoogleMap map) {}
```

2. In the `setPoiClick()` method, set an `OnPoiClickListener` on the passed-in `GoogleMap`:

```
map.setOnPoiClickListener(new GoogleMap.OnPoiClickListener() {
    @Override
    public void onPoiClick(PointOfInterest poi) {
    }
});
```

3. In the `onPoiClick()` method, place a marker at the POI location. Set the title to the name of the POI. Save the result to a variable called `poiMarker`.

```
public void onPoiClick(PointOfInterest poi) {
    Marker poiMarker = mMap.addMarker(new MarkerOptions()
        .position(poi.latLng)
        .title(poi.name));
}
```

4. Call `showInfoWindow()` on `poiMarker` to immediately show the info window.

```
poiMarker.showInfoWindow();
```

5. Call `setPoiClick()` at the end of `onMapReady()`. Pass in `mMap`.

6. Run your app and find a POI such as a park. Tap on the POI to place a marker on it and display the POI's name in an info window.

## Task 3. Style your map

You can customize Google Maps in many ways, giving your map a unique look and feel.

You can customize a `MapFragment` object using the available [XML attributes](#), as you would customize any other fragment. However, in this step you customize the look and feel of the *content* of the `MapFragment`, using methods on the `GoogleMap` object. You use the online [Styling Wizard](#) to add a style to your map and customize your markers. You also add a `GroundOverlay` to your home location that scales and rotates with the map.

### 3.1 Add a style to your map

To create a customized style for your map, you generate a JSON file that specifies how features in the map are displayed. You don't have to create this JSON file manually: Google provides the [Styling Wizard](#), which generates the JSON for you after you visually style your map. In this practical, you style the map for "night mode," meaning that the map uses dim colors and low contrast for use at night.

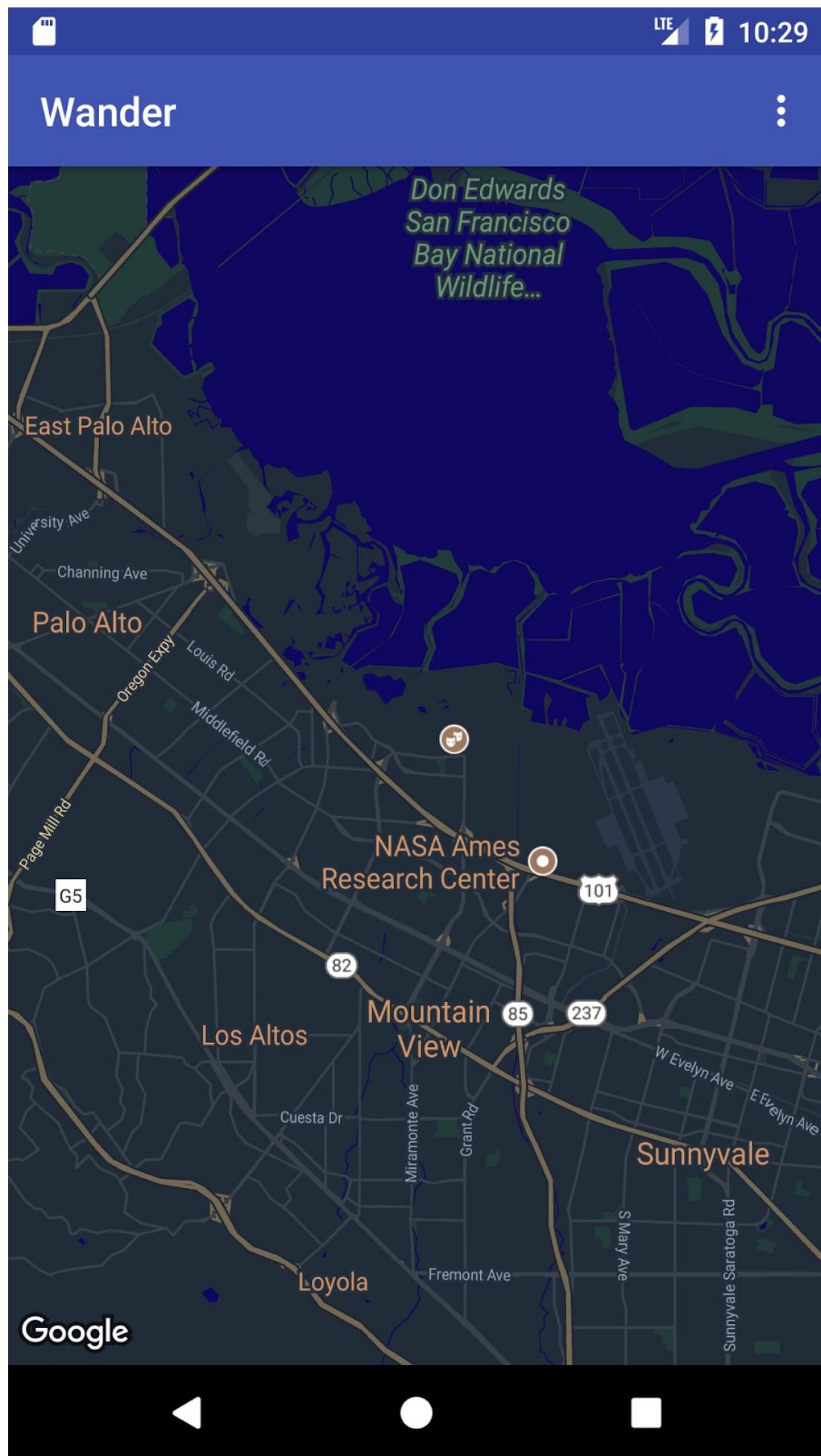
**Note:** Styling only applies to maps that use the `normal` map type.

1. Navigate to <https://mapstyle.withgoogle.com/> in your browser.
2. Select **Create a Style**.
3. Select the **Night** theme.
4. Click **More Options** at the bottom of the menu.
5. At the bottom of the **Feature type** list, select **Water > Fill**. Change the color of the water to a dark blue (for example, #160064).
6. Click **Finish**. Copy the JSON code from the resulting pop-up window.
7. In Android Studio, create a resource directory called `raw` in the `res` directory. Create a file in `res/raw` called `map_style.json`.
8. Paste the JSON code into the new resource file.
9. To set the JSON style to the map, call `setMapStyle()` on the `GoogleMap` object. Pass in a `MapStyleOptions` object, which loads the JSON file. The `setMapStyle()` method returns a boolean indicating the success of the styling. If the file can't be loaded, the method throws a `Resources.NotFoundException`.

Copy the following code into the `onMapReady()` method to style the map. You may need to create a `TAG` string for your log statements:

```
try {  
    // Customize the styling of the base map using a JSON object defined  
    // in a raw resource file.  
    boolean success = googleMap.setMapStyle(  
        MapStyleOptions.loadRawResourceStyle(  
            this, R.raw.map_style));  
  
    if (!success) {  
        Log.e(TAG, "Style parsing failed.");  
    }  
} catch (Resources.NotFoundException e) {  
    Log.e(TAG, "Can't find style. Error: ", e);  
}
```

10. Run your app. The new styling should be visible when the map is in `normal` mode.



## 3.2 Style your marker

You can personalize your map further by styling the map markers. In this step, you change the default red markers to match the night mode color scheme.

1. In the `onMapLongClick()` method, add the following line of code to the `MarkerOptions()` constructor to use the default marker but change the color to blue:

```
.icon(BitmapDescriptorFactory.defaultMarker  
    (BitmapDescriptorFactory.HUE_BLUE))
```

2. Run the app. The markers you place are now shaded blue, which is more consistent with the night-mode theme of the app.

Note that POI markers are still red, because you didn't add styling to the `onPoiClick()` method.

## 3.3 Add an overlay

One way you can customize the Google Map is by drawing on top of it. This technique is useful if you want to highlight a particular type of location, such as popular fishing spots. Three types of overlays are supported:

- Shapes: You can add `polylines`, `polygons`, and `circles` to the map.
- `TileOverlay` objects: A tile overlay defines a set of images that are added on top of the base map tiles. Tile overlays are useful when you want to add extensive imagery to the map. A typical tile overlay covers a large geographical area.
- `GroundOverlay` objects: A ground overlay is an image that is fixed to a map. Unlike markers, ground overlays are oriented to the Earth's surface rather than to the screen. Rotating, tilting, or zooming the map changes the orientation of the image. Ground overlays are useful when you wish to fix a single image at one area on the map

In this step, you add a ground overlay in the shape of an Android to your home location.

1. Download [this Android image](#) and save it in your `res/drawable` folder.
2. In `onMapReady()`, after the call to move the camera to the home position, create a `GroundOverlayOptions` object. Assign the object to a variable called `homeOverlay`:

```
GroundOverlayOptions homeOverlay = new GroundOverlayOptions();
```

3. Use the `BitmapDescriptorFactory.fromResource()` method to create a `BitmapDescriptor` object from the above image. Pass the object into the `image()` method of the `GroundOverlayOptions` object:

```
GroundOverlayOptions homeOverlay = new GroundOverlayOptions()  
    .image(BitmapDescriptorFactory.fromResource(R.drawable.android));
```

4. Set the `position` property for the `GroundOverlayOptions` object by calling the `position()` method. Pass in the `home` `LatLng` object and a `float` for the width in meters of the desired overlay. For this example, a width of 100 m works well:

```
GroundOverlayOptions homeOverlay = new GroundOverlayOptions()  
    .image(BitmapDescriptorFactory.fromResource(R.drawable.android))  
    .position(home, 100);
```

5. Call `addGroundOverlay()` on the `GoogleMap` object. Pass in your `GroundOverlayOptions` object:

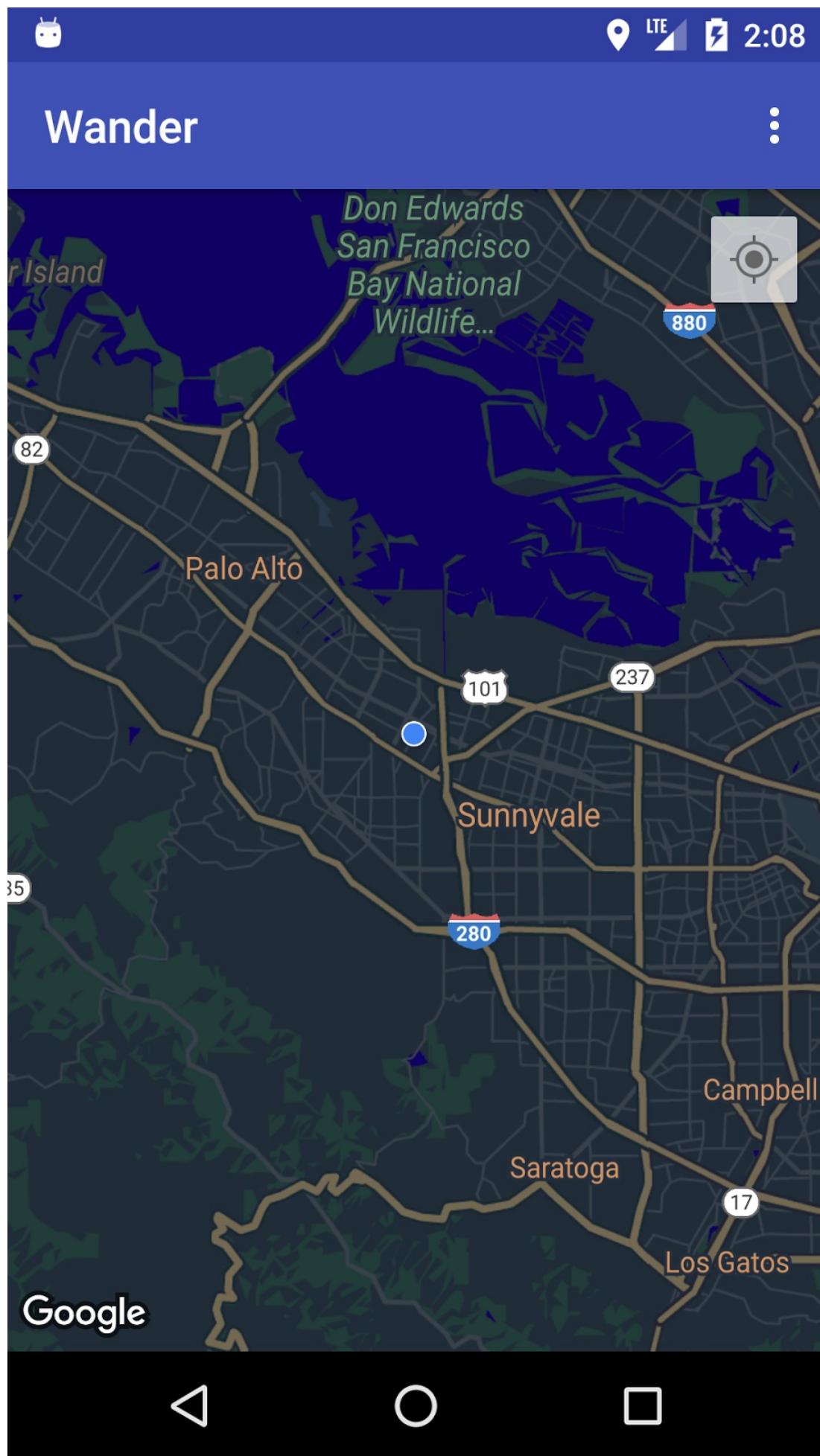
```
mMap.addGroundOverlay(homeOverlay);
```

6. Run the app. Zoom in on your home location, and you see the Android image as an overlay.

## Task 4. Enable location tracking and Street View

Users often use Google Maps to see their current location, and you can obtain device location using the [Location Services API](#). To display the device location on your map without further use of `Location` data, you can use the [location-data layer](#).

The location-data layer adds a **My Location** button to the top-right side of the map. When the user taps the button, the map centers on the device's location. The location is shown as a blue dot if the device is stationary, and as a blue chevron if the device is moving.



You can provide additional information about a location using Google Street View, which is a navigable panorama photo of a given location.

In this task, you enable the location-data layer and Street View so that when the user taps the info window for the POI marker, the map goes into to Street View mode.

## 4.1 Enable location tracking

Enabling location tracking in Google Maps requires a single line of code. However, you must make sure that the user has granted location permissions (using the runtime-permission model).

In this step, you request location permissions and enable the location tracking.

1. In the `AndroidManifest.xml` file, verify that the `FINE_LOCATION` permission is already present. Android Studio inserted this permission when you selected the Google Maps template.
2. To enable location tracking in your app, create a method in the `MapsActivity` called `enableMyLocation()` that takes no arguments and doesn't return anything.
3. In the `enableMyLocation()` method, check for the `ACCESS_FINE_LOCATION` permission. If the permission is granted, enable the location layer. Otherwise, request the permission:

```
private void enableMyLocation() {
    if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_FINE_LOCATION)
        == PackageManager.PERMISSION_GRANTED) {
        mMap.setMyLocationEnabled(true);
    } else {
        ActivityCompat.requestPermissions(this, new String[]
            {Manifest.permission.ACCESS_FINE_LOCATION},
            REQUEST_LOCATION_PERMISSION);
    }
}
```

4. Call `enableMyLocation()` from the `onMapReady()` callback to enable the location layer.
5. Override the `onRequestPermissionsResult()` method. If the permission is granted, call `enableMyLocation()`:

```

@Override
public void onRequestPermissionsResult(int requestCode,
    @NonNull String[] permissions,
    @NonNull int[] grantResults) {
    // Check if location permissions are granted and if so enable the
    // location data layer.
    switch (requestCode) {
        case REQUEST_LOCATION_PERMISSION:
            if (grantResults.length > 0
                && grantResults[0]
                == PackageManager.PERMISSION_GRANTED) {
                enableMyLocation();
                break;
            }
    }
}

```

- Run the app. The top-right corner now contains the **My Location** button, which displays the device's current location.

**Note:** When you run the app on an emulator, the location may not be available. If you haven't used the emulator settings to set a location, the location button will be unavailable.

## 4.2 Enable Street View

Google Maps provides Street View, which is a panoramic view of a location with controls for navigating along a designated path. [Street View does not have global coverage](#).

In this step, you enable a Street View panorama that is activated when the user taps a POI's info window. You need to do two things:

- Distinguish POI markers from other markers, because you want your app's functionality to work *only* on POI markers. This way, you can start Street View when the user taps a POI info window, but not when the user taps any other type of marker.

The `Marker` class includes a `setTag()` method that allows you to attach data. (The data can be anything that extends from `Object`). You will set a tag on the markers that are created when users click POIs.

- When the user taps a tagged info window in an `OnInfoWindowClickListener`, replace the `MapFragment` with a `StreetViewPanoramaFragment`. (The code below uses `SupportMapFragment` and `SupportStreetViewPanoramaFragment` to support Android versions below API 12.)

If any of the fragments change at runtime, you must add them in the containing `Activity` class, and not statically in XML.

## Tag the POI marker

1. In the `onPoiClick()` callback, call `setTag()` on `poiMarker`. Pass in any arbitrary string:

```
poiMarker.setTag("poi");
```

## Replace the static SupportMapFragment with a runtime instance

2. Open `activity_maps.xml` and change the element to a frame layout that will serve as the container for your fragments:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/fragment_container"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" />
```

3. In `onCreate()` in `MapsActivity`, remove the code that finds the `SupportMapFragment` by ID, because there is no longer a static `SupportMapFragment` in the XML. Instead, create a new runtime instance of the `SupportMapFragment` by calling

```
SupportMapFragment.newInstance() :
```

```
SupportMapFragment mapFragment = SupportMapFragment.newInstance();
```

4. Add the fragment to the `FrameLayout` using a fragment transaction with the `FragmentManager`:

```
getSupportFragmentManager().beginTransaction()  
    .add(R.id.fragment_container, mapFragment).commit();
```

5. Keep the line of code that triggers the asynchronous loading of the map:

```
mapFragment.getMapAsync(this);
```

## Set an OnInfoWindowClickListener and check the marker tag

6. Create a method stub in `MapsActivity` called `setInfoWindowClickListener()` that takes a `GoogleMap` as an argument and returns `void`:

```
private void setInfoWindowClickListener(GoogleMap map) {}
```

7. Set an `OnInfoWindowClickListener` to the `GoogleMap`:

```
map.setOnInfoWindowClickListener(
    new GoogleMap.OnInfoWindowClickListener() {
        @Override
        public void onInfoWindowClick(Marker marker) {
        }
    });
});
```

8. In the `onInfoWindowClick()` method, check whether the marker contains the string tag you set in the `onPoiClick()` method:

```
if (marker.getTag() == "poi") {}
```

## Replace the `SupportMapFragment` with a `SupportStreetViewPanoramaFragment`

9. In the case where the marker contains the tag, specify the location for the Street View panorama by using a `StreetViewPanoramaOptions` object. Set the object's `position` property to the position of the passed-in marker:

```
StreetViewPanoramaOptions options =
    new StreetViewPanoramaOptions().position(
        marker.getPosition());
```

10. Create a new instance of `SupportStreetViewPanoramaFragment`, passing in the `options` object you created:

```
SupportStreetViewPanoramaFragment streetViewFragment
    = SupportStreetViewPanoramaFragment
        .newInstance(options);
```

11. Start a fragment transaction. Replace the contents of the fragment container with the new fragment, `streetViewFragment`. Add the transaction to the back stack, so that pressing back will navigate back to the `SupportMapFragment` and not exit the app:

```
getSupportFragmentManager().beginTransaction()
    .replace(R.id.fragment_container,
        streetViewFragment)
    .addToBackStack(null).commit();
```

**Note:** The argument for the `addToBackStack()` method is an optional name. The name is used to manipulate the back stack state. In this case, the name is not used again, so you can pass in `null`.

12. Run the app. Zoom into a city that has [Street View coverage](#), such as Mountain View (home of Google HQ), and find a POI, such as a park. Tap on the POI to place a marker and show the info window. Tap the info window to enter Street View mode for the location of the marker. Press the back button to return to the map fragment.



Google

Report a problem © Brett Dowling

# Coding challenge

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

**Challenge:** If you tap the info window for a POI in a location where there is no Street View coverage, you see a black screen.

- To check whether Street View is available in an area, implement the `OnStreetViewPanoramaReady` callback in combination with the `StreetViewPanorama.OnStreetViewPanoramaChangeListener`.
- If Street View isn't available in a selected area, go back to the map fragment and show an error.

## Solution code

[wander](#) solution code. (Doesn't include the Challenge solution.)

## Summary

- To use the Maps API, you need an API key from the [Google API Console](#).
- In Android Studio, using the Google Maps Activity template generates an `Activity` with a single `SupportMapFragment` in the app's layout. The template also adds the `ACCESS_FINE_PERMISSION` to the app manifest, implements the `OnMapReadyCallback` in your activity, and overrides the required `onMapReady()` method.
- [Google Maps](#) can be in one of the following map types:
  - *Normal* : Typical road map. Shows roads, some features built by humans, and important natural features like rivers. Road and feature labels are also visible.
  - *Hybrid* : Satellite photograph data with road maps added. Road and feature labels are also visible.
  - *Satellite* : Photograph data. Road and feature labels are not visible.
  - *Terrain* : Topographic data. The map includes colors, contour lines and labels, and perspective shading. Some roads and labels are also visible.
  - *None* : No map.
- You can change the map type at runtime by using the `GoogleMap.setMapType()` . method.
- A marker is an indicator for a specific geographic location.
- When tapped, the default behavior for a marker is to display an info window with information about the location.

- By default, points of interest (POIs) appear on the base map along with their corresponding icons. POIs include parks, schools, government buildings, and more.
- In addition, business POIs (shops, restaurants, hotels, and more) appear by default on the map when the map type is `normal`.
- You can capture clicks on POIs using the [OnPoiClickListener](#).
- You can change the visual appearance of almost all elements of a Google Map using the [Styling Wizard](#). The Styling Wizard generates a JSON file that you pass into the Google Map using the `setMapStyle()` method.
- You can customize your markers by changing the default color, or replacing the default marker icon with a custom image.
- Use a [ground overlay](#) to fix an image to a geographic location.
- Use a `GroundOverlayOptions` object to specify the image, the image's size in meters, and the image's position. Pass this object to the `GoogleMap.addGroundOverlay()` method to set the overlay to the map.
- Provided that your app has the `ACCESS_FINE_LOCATION` permission, you can enable location tracking using the `mMap.setMyLocationEnabled(true)` method.
- Google Street View provides panoramic 360-degree views from designated roads throughout its coverage area.
- Use the `StreetViewPanoramaFragment.newInstance()` method to create a new Street View fragment.
- To specify the options for the view, use a `StreetViewPanoramaOptions` object. Pass the object into the `newInstance()` method.

## Related concept

The related concept documentation is in [9.1 C: Google Maps API](#).

## Learn more

Android developer documentation:

- [Getting Started with the Google Maps Android API](#)
- [Adding a Map with a Marker](#)
- [Map Objects](#)
- [Adding a Styled Map](#)
- [Street View](#)
- [Ground Overlays](#)

Reference documentation:

- [GoogleMap](#)
- [SupportMapFragment](#)
- [SupportStreetViewPanoramaFragment](#)

# 10.1A: Creating a custom view from a View subclass

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Customize an EditText view
- Solution code
- Summary
- Related concept
- Learn more

Android offers a large set of `View` subclasses, such as `Button`, `TextView`, `EditText`, `ImageView`, `CheckBox`, or `RadioButton`. You can use these subclasses to construct a UI that enables user interaction and displays information in your app. If the `View` subclasses don't meet your needs, you can create a custom view that does.

After you create a custom view, you can add it to different layouts in the same way you would add a `TextView` or `Button`. This lesson shows you how to create and use custom views based on `View` subclasses.

## What you should already KNOW

You should be able to:

- Create and run apps in Android Studio.
- Use the Layout Editor to create a UI.
- Edit a layout in XML.
- Use touch, text, and click listeners in your code.

## What you will LEARN

You will learn how to:

- Extend the `View` subclass `EditText` to create a custom text-editing view.
- Use listeners to handle user interaction with the custom view.

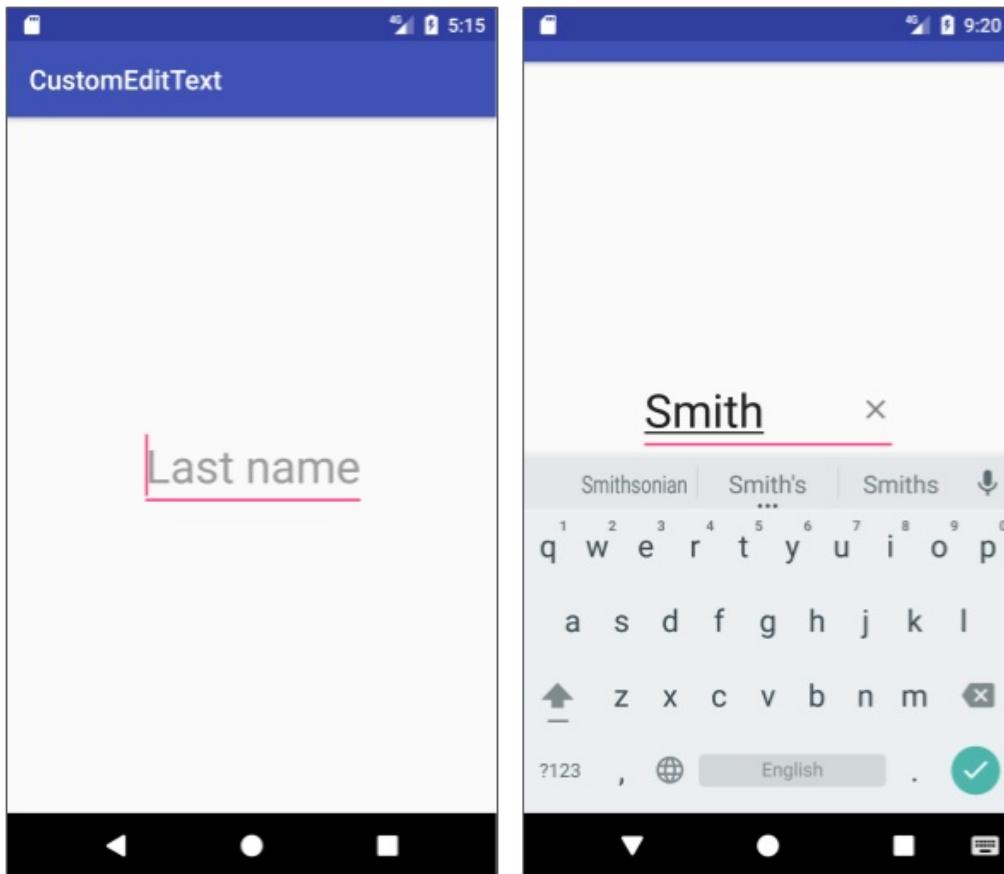
- Use a custom view in a layout.

## What you will DO

- Add a new class that extends the `EditText` class to create a custom view.
- Use listeners to provide the custom view's behavior.
- Add the custom view to a layout.

## App overview

The CustomEditText app demonstrates how to extend `EditText` to make a custom text-editing view. The custom view includes a clear (X) button for clearing text. After the custom view is created, you can use multiple versions of it in layouts, applying different `EditText` attributes as needed.



## Task 1. Customize an `EditText` view

In this task, you create an app with a customized `EditText` view that includes a clear (X) button on the right side of the `EditText`. The user can tap the X to clear the text. Specifically, you will:

- Create an app with an `EditText` view as a placeholder.
- Add layout attributes to position the view, and to support right-to-left (RTL) languages for text input.
- Extend the `EditText` class to create a custom view.
- Initialize the custom view with a `drawable` that appears at the end of the `EditText`.
- Use a text listener to show the `drawable` only when text is entered into the `EditText`.
- Use a touch listener to clear the text if the user taps the `drawable`.
- Replace the placeholder `EditText` with the custom view in the layout.

## 1.1 Create an app with an `EditText` view

In this step, you add two `drawables` for the clear (X) button:

- An opaque version  that appears when the user enters text.
- A black version  that appears while the user is tapping the X.

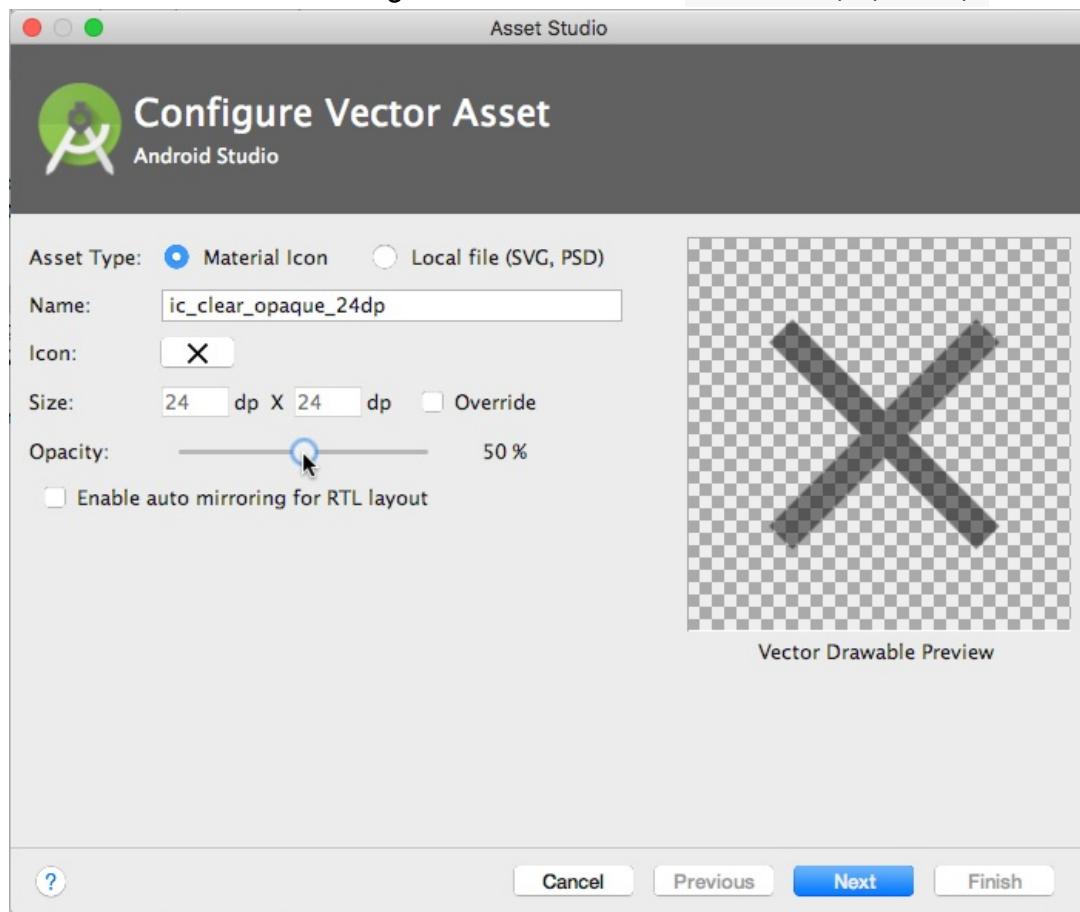
You also change a `TextView` to an `EditText` with attributes for controlling its appearance. If the layout direction is set to a right-to-left (RTL) language, these attributes change the direction in which the user enters text. (For more about supporting RTL languages, see the lesson on localization.)

1. Create an app named `customEditText` using the Empty Activity template. Make sure that **Generate Layout File** is selected so that the `activity_main.xml` layout file is generated.
2. Edit the `build.gradle (Module: app)` file. Change the minimum SDK version to 17, so that you can support RTL languages and place `drawables` in either the left or right position in `EditText` views:

```
minSdkVersion 17
```

3. Right-click the `drawable/` folder and choose **New > Vector Asset**. Click the Android icon and choose the clear (X) icon. Its name changes to `ic_clear_black_24dp`. Click **Next** and **Finish**.

4. Repeat step 3, choosing the clear (X) icon again, but this time drag the **Opacity** slider to 50% as shown below. Change the icon's name to `ic_clear_opaque_24dp`.



5. In `activity_main.xml`, change the "Hello World" `TextView` to an `EditText` with the following attributes:

Attribute	Value
<code>android:id</code>	<code>"@+id/my_edit_text"</code>
<code>android:layout_width</code>	<code>"wrap_content"</code>
<code>android:layout_height</code>	<code>"wrap_content"</code>
<code>android:textAppearance</code>	<code>"@style/Base.TextAppearance.AppCompat.D</code>
<code>android:inputType</code>	<code>"textCapSentences"</code>
<code>android:layout_gravity</code>	<code>"start"</code>
<code>android:textAlignment</code>	<code>"viewStart"</code>
<code>app:layout_constraintBottom_toBottomOf</code>	<code>"parent"</code>
<code>app:layout_constraintLeft_toLeftOf</code>	<code>"parent"</code>
<code>app:layout_constraintRight_toRightOf</code>	<code>"parent"</code>
<code>app:layout_constraintTop_toTopOf</code>	<code>"parent"</code>
<code>android:hint</code>	<code>"Last name"</code>

6. Extract the string resource for "Last name" to `last_name`.

7. Run the app. It displays an `EditText` field for entering text (a last name), and uses the `textCapSentences` attribute to capitalize the first letter.



## 1.2 Add a subclass that extends EditText

1. Create a new Java class called `EditTextWithClear` with the superclass set to `android.support.v7.widget.AppCompatEditText`. `AppCompatEditText` is an `EditText` subclass that supports compatible features on older version of the Android platform.
2. The editor opens `EditTextWithClear.java`. A red bulb appears a few moments after you click the class definition because the class is not complete—it needs constructors.
3. Click the red bulb and select **Create constructor matching super**. Select all three constructors in the popup menu, and click **OK**.

The three constructors are:

- **AppCompatEditText(context:Context)**: Required for creating an instance of a view programmatically.
- **AppCompatEditText(context:Context, attrs:AttributeSet)**: Required to inflate the view from an XML layout and apply XML attributes.
- **AppCompatEditText(context:Context, attrs:AttributeSet, defStyleAttr:int)**: Required to apply a default style to all UI elements without having to specify it in each layout file.

## 1.3 Initialize the custom view

Create a helper method that initializes the view, and call that method from each constructor. That way, you don't have to repeat the same code in each constructor.

1. Define a member variable for the drawable (the X button image).

```
Drawable mClearButtonImage;
```

2. Create a `private` method called `init()`, with no parameters, that initializes the member variable to the `drawable resource ic_clear_opaque_24dp`.

```
private void init() {  
    mClearButtonImage = ResourcesCompat.getDrawable(getResources(),  
        R.drawable.ic_clear_opaque_24dp, null);  
    // TODO: If the clear (X) button is tapped, clear the text.  
    // TODO: If the text changes, show or hide the clear (X) button.  
}
```

The code includes two `TODO` comments for upcoming steps of this task.

3. Add the `init()` method call to each constructor:

```

public EditTextWithClear(Context context) {
    super(context);
    init();
}
public EditTextWithClear(Context context, AttributeSet attrs) {
    super(context, attrs);
    init();
}
public EditTextWithClear(Context context,
                        AttributeSet attrs, int defStyleAttr) {
    super(context, attrs, defStyleAttr);
    init();
}

```

## 1.4 Show or hide the X button

If the user enters text, the `EditTextWithClear` custom view shows the clear (X) button. If there is no text in the field, the `EditTextWithClear` custom view hides the clear (X) button.

To show or hide the button, use the `TextWatcher` interface, whose methods are called if the text changes. Follow these steps:

1. Open `EditTextWithClear` and create two `private` methods with no parameters, `showClearButton()` and `hideClearButton()`. In these methods, use `setCompoundDrawablesRelativeWithIntrinsicBounds()` to show or hide the clear (X) button.

```

/**
 * Shows the clear (X) button.
 */
private void showClearButton() {
    setCompoundDrawablesRelativeWithIntrinsicBounds
        (null,                      // Start of text.
         null,                      // Above text.
         mClearButtonImage,          // End of text.
         null);                     // Below text.
}

/**
 * Hides the clear button.
 */
private void hideClearButton() {
    setCompoundDrawablesRelativeWithIntrinsicBounds
        (null,                      // Start of text.
         null,                      // Above text.
         null,                      // End of text.
         null);                     // Below text.
}

```

In `showClearButton()`, the `setCompoundDrawablesRelativeWithIntrinsicBounds()` method sets the `drawable mClearButtonImage` to the *end* of the text. The method accommodates right-to-left (RTL) languages by using the arguments as "start" and "end" positions rather than "left" and "right" positions. For more about supporting RTL languages, see the lesson on localization.

Use `null` for positions that should *not* show a `drawable`. In `hideClearButton()`, the `setCompoundDrawablesRelativeWithIntrinsicBounds()` method replaces the `drawable` with `null` in the *end* position.

The `setCompoundDrawablesRelativeWithIntrinsicBounds()` method returns the exact size of the `drawable`. This method requires a minimum Android API level 17 or newer. Be sure to edit your `build.gradle (Module: app)` file to use `minSdkVersion 17`.

2. In `EditTextWithClear`, add a `TextWatcher()` inside the `init()` method, replacing the second `TODO` comment (`TODO: If the text changes, show or hide the clear (X) button`). Let Android Studio do the work for you: start by entering **addText**:

```
// If the text changes, show or hide the clear (X) button.  
addText
```

3. After entering **addText**, choose the suggestion that appears for **addTextChangedListener(TextWatcher watcher)**. The code changes to the following, and a red bulb appears as a warning.

```
addTextChangedListener()
```

4. In the code shown above, enter **new T** inside the parentheses:

```
addTextChangedListener(new T)
```

5. Choose the **TextWatcher{...}** suggestion that appears. Android Studio creates the `beforeTextChanged()`, `onTextChanged()`, and `afterTextChanged()` methods inside the `addTextChangedListener()` method, as shown in the code below:

```

addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start,
                                 int count, int after) {
    }

    @Override
    public void onTextChanged(CharSequence s, int start,
                             int before, int count) {
    }

    @Override
    public void afterTextChanged(Editable s) {
    }
});

```

6. In the `onTextChanged()` method, call the `showClearButton()` method for showing the clear (**X**) button. You implement only `onTextChanged()` in this practical, so leave `beforeTextChanged()` and `afterTextChanged()` alone, or just add comments to them.

```

public void onTextChanged(CharSequence s, int start,
                        int before, int count) {
    showClearButton();
}

```

## 1.5 Add touch and text listeners

Other behaviors of the `EditTextWithClear` custom view are to:

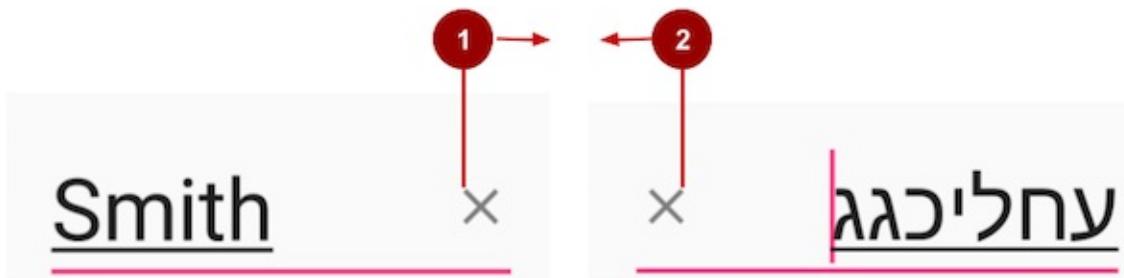
- Clear the text from the field if the user taps the clear (**X**) button.
- Render the clear (**X**) button as opaque before the user taps it, and black while the user is tapping it.

To detect the tap and clear the text, use the `View.OnTouchListener` interface. The interface's `onTouch()` method is called when a touch event occurs with the button.

**Tip:** To learn more about event listeners, see [Input Events](#).

You should design the `EditTextWithClear` class to be useful in both left-to-right (LTR) and right-to-left (RTL) language layouts. However, the button is on the right side in an LTR layout, and on the left side of an RTL layout. The code needs to detect whether the touch occurred on the button itself. It checks to see if the touch occurred *after the start* location of

the button. The start location of the button is different in an RTL layout than it is in an LTR layout, as shown in the figure.



In the figure above:

1. The start location of the button in an LTR layout. Moving to the right, the touch must occur *after* this location on the screen and before the *right* edge.
2. The start location of the button in an RTL layout. Moving to the left, the touch must occur *after* this location on the screen and before the *left* edge.

**Tip:** To learn more about reporting finger movement events, see [MotionEvent](#).

Follow these steps to use the `View.OnTouchListener` interface:

1. In the `init()` method, replace the first `TODO` comment (`TODO: If the clear (X) button is tapped, clear the text`) with the following code. If the clear (X) button is visible, this code sets a touch listener that responds to events inside the bounds of the button.

```
// If the clear (X) button is tapped, clear the text.
setOnTouchListener(new OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        return false;
    }
});
```

2. In the `onTouch()` method, replace the single `return false` statement with the following:

```
if ((getCompoundDrawablesRelative()[2] != null)) {
    float clearButtonStart; // Used for LTR languages
    float clearButtonEnd; // Used for RTL languages
    boolean isClearButtonClicked = false;
    // TODO: Detect the touch in RTL or LTR layout direction.
    // TODO: Check for actions if the button is tapped.
}
return false;
```

In the previous step, you set the location of the clear (X) button using

`setCompoundDrawablesRelativeWithIntrinsicBounds()`:

- Location 0: Start of text (set to `null` ).
- Location 1: Top of text (set to `null` ).
- Location 2: End of text (set to `mClearButtonImage` ).
- Location 3: Bottom of text (set to `null` ).

In the above step, you use the `getCompoundDrawablesRelative()[2]` expression, which uses `getCompoundDrawablesRelative()` to return the `drawable` at the end of the text [2]. If no `drawable` is present, the expression returns `null`. The code executes only if that location is *not* `null` —which means that the clear (X) button is in that location. Otherwise, the code returns `false`.

## 1.6 Recognize the user's tap

To recognize the user's tap on the clear (X) button, you need to get the intrinsic bounds of the button and compare it with the touch event.

- For an LTR language, the clear (X) button starts on the right side of the field. Any touch occurring *after* the start point is a touch on the button itself.
- For an RTL language, the clear (X) button ends on the left side of the field. Any touch occurring *before* the endpoint is a touch on the button itself.

Follow these steps:

1. Use `getLayoutDirection()` to get the current layout direction. Use the `MotionEvent getX()` method to determine whether the touch occurred after the start of the button in an LTR layout, or before the end of the button in an RTL layout. In the `onTouch()` method, replace the first `TODO` comment (`TODO: Detect the touch in RTL or LTR layout direction`):

```
// Detect the touch in RTL or LTR layout direction.  
if (getLayoutDirection() == LAYOUT_DIRECTION_RTL) {  
    // If RTL, get the end of the button on the left side.  
    clearButtonEnd = mClearButtonImage  
        .getIntrinsicWidth() + getPaddingStart();  
    // If the touch occurred before the end of the button,  
    // set isClearButtonClicked to true.  
    if (event.getX() < clearButtonEnd) {  
        isClearButtonClicked = true;  
    }  
} else {  
    // Layout is LTR.  
    // Get the start of the button on the right side.  
    clearButtonStart = (getWidth() - getPaddingEnd()  
        - mClearButtonImage.getIntrinsicWidth());  
    // If the touch occurred after the start of the button,  
    // set isClearButtonClicked to true.  
    if (event.getX() > clearButtonStart) {  
        isClearButtonClicked = true;  
    }  
}
```

2. Check for actions if the clear (**X**) button is tapped. On `ACTION_DOWN`, you want to show the black version of the button as a highlight. On `ACTION_UP`, you want to switch back to the default version of the button, clear the text, and hide the button. In the `onTouch()` method, replace the second `TODO` comment (`TODO: Check for actions if the button is tapped`):

```

// Check for actions if the button is tapped.
if (isClearButtonClicked) {
    // Check for ACTION_DOWN (always occurs before ACTION_UP).
    if (event.getAction() == MotionEvent.ACTION_DOWN) {
        // Switch to the black version of clear button.
        mClearButtonImage =
            ResourcesCompat.getDrawable(getResources(),
                R.drawable.ic_clear_black_24dp, null);
        showClearButton();
    }
    // Check for ACTION_UP.
    if (event.getAction() == MotionEvent.ACTION_UP) {
        // Switch to the opaque version of clear button.
        mClearButtonImage =
            ResourcesCompat.getDrawable(getResources(),
                R.drawable.ic_clear_opaque_24dp, null);
        // Clear the text and hide the clear button.
        getText().clear();
        hideClearButton();
        return true;
    }
} else {
    return false;
}

```

The first touch event is `ACTION_DOWN`. Use it to check if the clear (X) button is touched (`ACTION_DOWN`). If it is, switch the clear button to the black version.

The second touch event, `ACTION_UP` occurs when the gesture is finished. Your code can then clear the text, hide the clear (X) button, and return `true`. Otherwise the code returns `false`.

## 1.7 Change the EditText view to the custom view

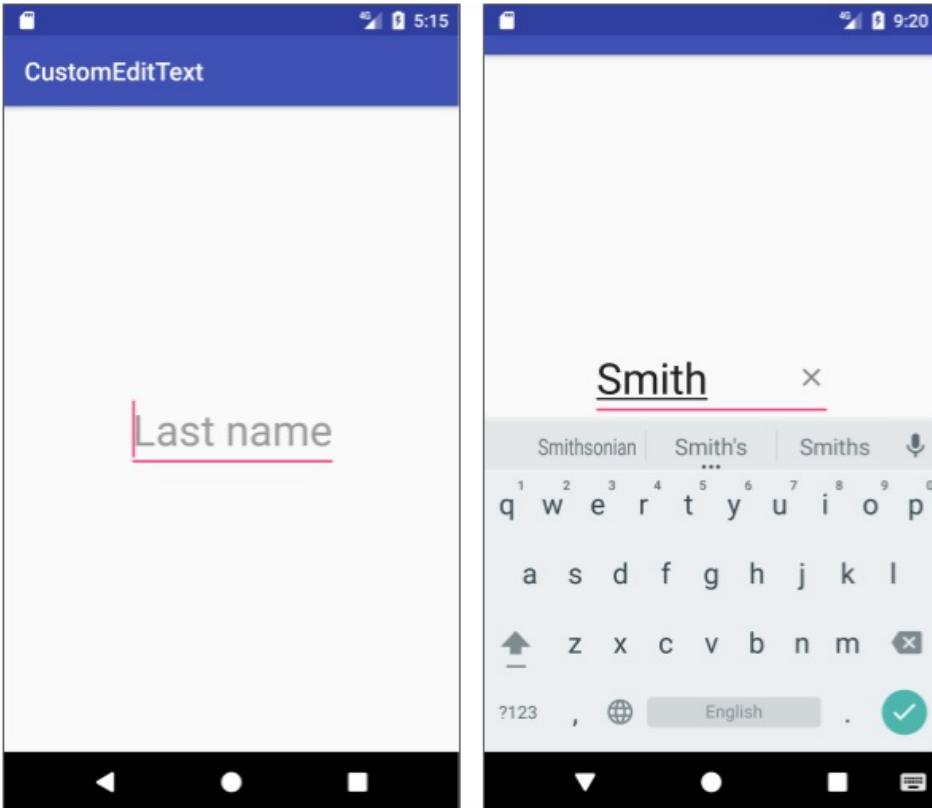
The `EditWithClear` class is now ready to be used in place of the `EditText` view in the layout:

1. In `activity_main.xml`, change the `EditText` tag for the `my_edit_text` element to `com.example.android.customedittext.EditWithClear`.

The `EditWithClear` class inherits the attributes defined for the original `EditText`, so there is no need to change any of them for this step.

If you see the message "classes missing" in the preview, click the link to rebuild the project.

- Run the app. Enter text, and then tap the clear (X) button to clear the text.



## 1.8 Run the app with an RTL language

To test an RTL language, you can add Hebrew in the Translations Editor, switch the device or emulator to Hebrew, and run the app. Follow these steps:

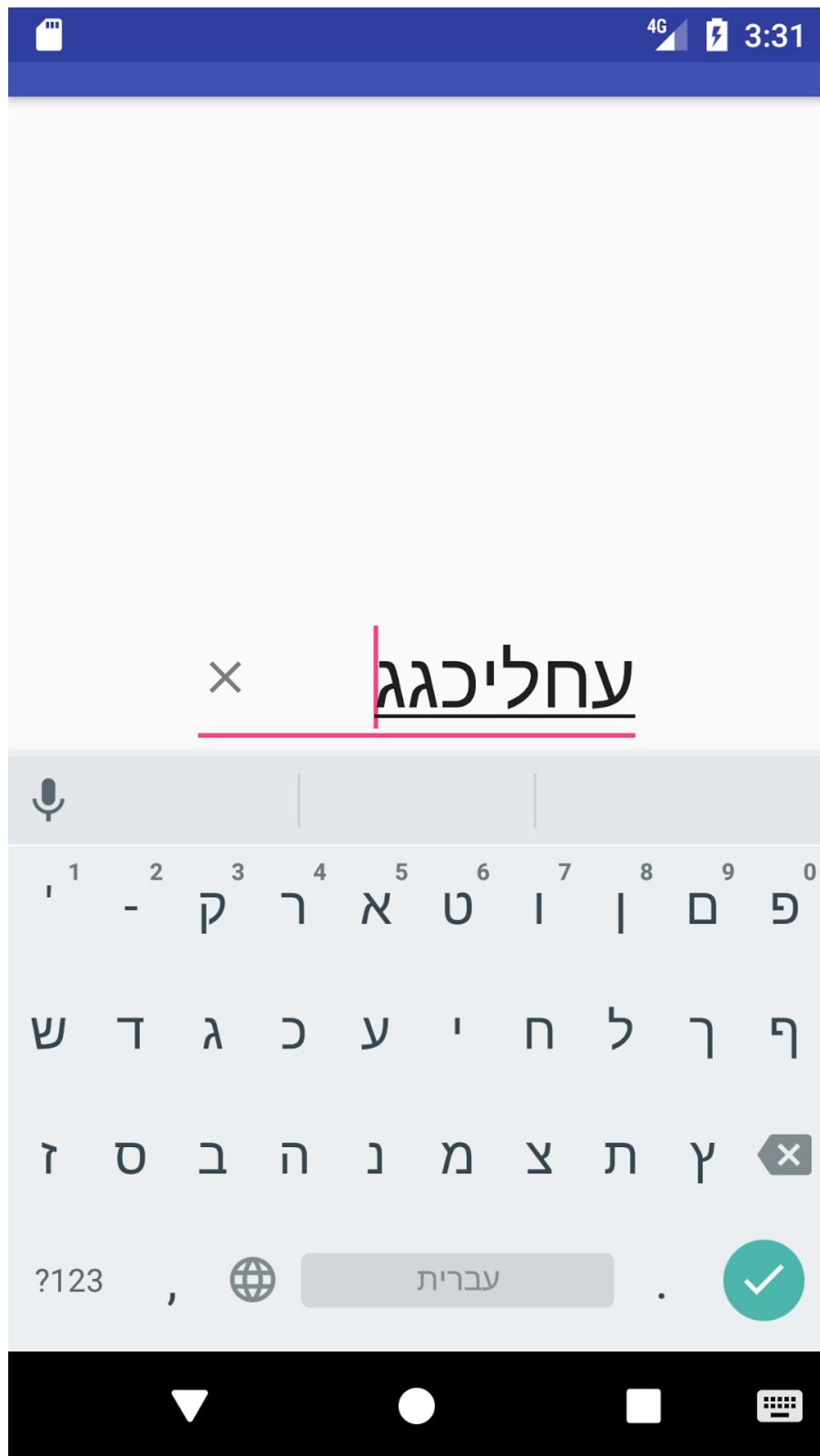
- Open the `strings.xml` file, and click the **Open editor** link in the top right corner to open the Translations Editor.
- Click the globe button in the top left corner of the **Translations Editor** pane, and select **Hebrew (iw) in Israel (IL)** in the dropdown menu.

After you choose a language, a new column with blank entries appears in the **Translations Editor** for that language, and the keys that have not yet been translated appear in red.

- Enter the Hebrew translation of "Last name" for the `last_name` key by selecting the key's cell in the column for the language (Hebrew), and entering the translation in the **Translation** field at the bottom of the pane. (For instructions on using the Translations Editor, see the chapter on localization.) When finished, close the Translations Editor.
- On your device or emulator, find the **Languages & input** settings in the Settings app. For devices using Android Oreo (8) or newer, the **Languages & input** choice is under **System**.

Be sure to remember the globe icon for the **Languages & input** choice, so that you can find it again if you switch to a language you do not understand. 

5. Choose **Languages** (or **Language** on Android 6 or older), which is easy to find because it is the first choice on the **Languages & input** screen.
6. For devices and emulators running Android 6 or older, select **עברית** for Hebrew. For devices and emulators running Android 7 or newer, click **Add a language**, select **עברית**, select ( **עברית (ישראל)** for the locale, and then use the move icon on the right side of the **Language preferences** screen to drag the language to the top of the list.
7. Run the app. The `EditTextWithClear` element should be reversed for an RTL language, with the clear (X) button on the left side, as shown below.
8. Put a finger on the clear (X) button, or if you're using a mouse, click and hold on the clear button. Then drag away from the clear button. The button changes from gray to black, indicating that it is still touched.



9. To change back from Hebrew to English, repeat Steps 4-6 above with the selection **English** for language and **United States** for locale.

## Solution code

Android Studio project: [CustomEditText](#)

## Summary

- To create a custom view that inherits the look and behavior of a `View` subclass such as `EditText`, add a new class that extends the subclass (such as `EditText`), and make adjustments by overriding some of the subclass methods.
- Add listeners such as `View.OnClickListener` to the custom view to define the view's interactive behavior.
- Add the custom view to an XML layout file with attributes to define the view's appearance, as you would with other UI elements.

**Tip:** View the different methods of the `View` subclasses, such as `TextView`, `Button`, and `ImageView`, to see how you can modify a `View` subclass by overriding these methods. For example, you can override the `setCompoundDrawablesRelativeWithIntrinsicBounds()` method of a `TextView` (or an `EditText`, which is a subclass of `TextView`) to set a `drawable` to appear to the start of, above, to the end of, and below the text.

## Related concept

The related concept documentation is [Custom views](#).

## Learn more

Android developer documentation:

- [Creating Custom Views](#)
- [Custom Components](#)
- `View`
- [Input Events](#)
- `onDraw()`
- `Canvas`
- `drawCircle()`

- `drawText()`
- `Paint`

### Video:

- [Quick Intro to Creating a Custom View in Android](#)
- [Android Custom View Tutorial](#)

# 10.1B: Creating a custom view from scratch

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Create a custom view from scratch
- Task 1 solution code
- Coding challenge
- Challenge solution code
- Summary
- Related concept
- Learn more

By extending `View` directly, you can create an interactive UI element of any size and shape by overriding the `onDraw()` method for the `view` to draw it. After you create a custom view, you can add it to different layouts in the same way you would add any other `view`. This lesson shows you how to create a custom view from scratch by extending `View` directly.

## What you should already KNOW

You should be able to:

- Create and run apps in Android Studio.
- Use the Layout Editor to create a UI.
- Edit a layout in XML.
- Use touch, text, and click listeners in your code.

## What you will LEARN

You will learn how to:

- Extend `View` to create a custom view.
- Draw a simple custom view that is circular in shape.
- Use listeners to handle user interaction with the custom view.

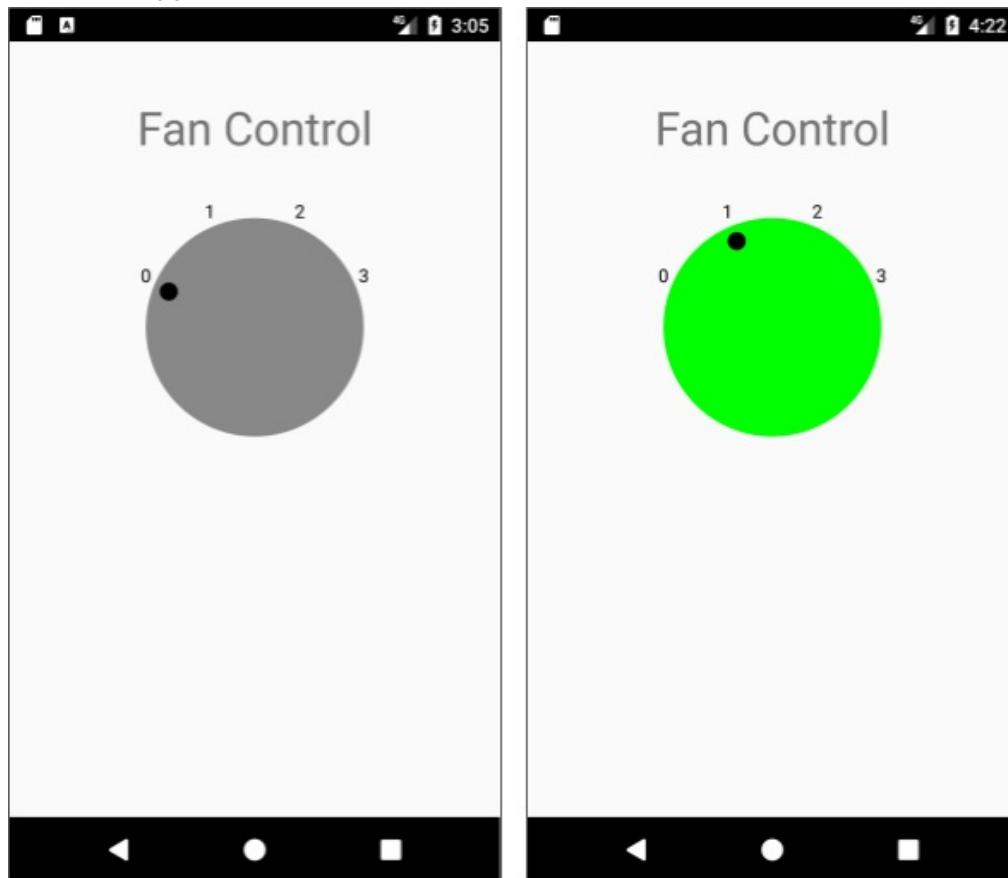
- Use a custom view in a layout.

## What you will DO

- Extend `View` to create a custom view.
- Initialize the custom view with drawing and painting values.
- Override `onDraw()` to draw the view.
- Use listeners to provide the custom view's behavior.
- Add the custom view to a layout.

## App overview

The [CustomFanController](#) app demonstrates how to create a custom view subclass from scratch by extending the `View` class. The app displays a circular UI element that resembles a physical fan control, with settings for off (0), low (1), medium (2), and high (3). You can modify the subclass to change the number of settings, and use standard XML attributes to define its appearance.



## Task 1. Create a custom view from scratch

In this task you will:

- Create an app with an `ImageView` as a placeholder for the custom view.
- Extend `View` to create the custom view.
- Initialize the custom view with drawing and painting values.
- Override `onDraw()` to draw the dial with an indicator and text labels for the settings: 0 (off), 1 (low), 2 (medium), and 3 (high).
- Use the `View.OnClickListener` interface to move the dial indicator to the next selection, and change the dial's color from gray to green for selections 1-3 (indicating that the fan power is on).
- In the layout, replace the `ImageView` placeholder with the custom view.

All the code to draw the custom view is provided in this task. (You learn more about `onDraw()` and drawing on a `Canvas` object with a `Paint` object in another lesson.)

## 1.1 Create an app with an ImageView placeholder

1. Create an app with the title `CustomFanController` using the Empty Activity template, and make sure the **Generate Layout File** option is selected.
2. In `activity_main.xml`, the "Hello World" `TextView` appears centered within a `ConstraintLayout`. Add or change the following attributes, leaving the other layout and constraint attributes (such as `layout_constraintTop_toTopOf`) the same:

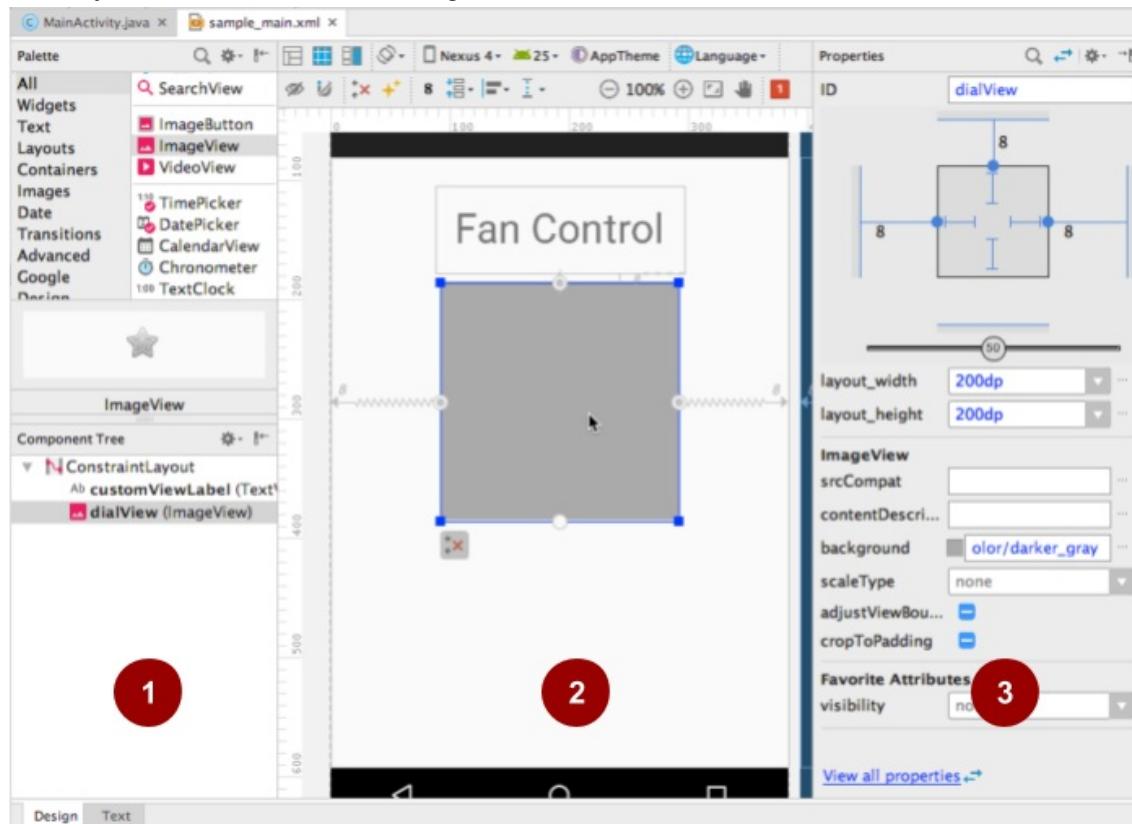
TextView attribute	Value
<code>android:id</code>	" <code>@+id/customViewLabel</code> "
<code>android:textAppearance</code>	" <code>@style/Base.TextAppearance.AppCompat.Display1</code> "
<code>android:padding</code>	" <code>16dp</code> "
<code>android:layout_marginLeft</code>	" <code>8dp</code> "
<code>android:layout_marginStart</code>	" <code>8dp</code> "
<code>android:layout_marginEnd</code>	" <code>8dp</code> "
<code>android:layout_marginRight</code>	" <code>8dp</code> "
<code>android:layout_marginTop</code>	" <code>24dp</code> "
<code>android:text</code>	"Fan Control"

3. Add an `ImageView` as a placeholder, with the following attributes:

ImageView attribute	Value
android:id	"@+id/dialView"
android:layout_width	"200dp"
android:layout_height	"200dp"
android:background	"@android:color/darker_gray"
app:layout_constraintTop_toBottomOf	"@+id/customViewLabel"
app:layout_constraintLeft_toLeftOf	"parent"
app:layout_constraintRight_toRightOf	"parent"
android:layout_marginLeft	"8dp"
android:layout_marginRight	"8dp"
android:layout_marginTop	"8dp"

#### 4. Extract string and dimension resources in both UI elements.

The layout should look like the figure below.



In the above figure:

1. Component Tree with layout elements in activity\_main.xml
2. ImageView to be replaced with a custom view
3. ImageView attributes

## 1.2 Extend View and initialize the view

1. Create a new Java class called `DialView`, whose superclass is `android.view.View`.
2. Click the red bulb for the new `DialView` class, and choose **Create constructor matching super**. Select the first three constructors in the popup menu (the fourth constructor requires API 21 and is not needed for this example).
3. At the top of `DialView` define the member variables you need in order to draw the custom view:

```
private static int SELECTION_COUNT = 4; // Total number of selections.  
private float mWidth; // Custom view width.  
private float mHeight; // Custom view height.  
private Paint mTextPaint; // For text in the view.  
private Paint mDialPaint; // For dial circle in the view.  
private float mRadius; // Radius of the circle.  
private int mActiveSelection; // The active selection.  
// String buffer for dial labels and float for ComputeXY result.  
private final StringBuffer mTempLabel = new StringBuffer(8);  
private final float[] mTempResult = new float[2];
```

The `SELECTION_COUNT` defines the total number of selections for this custom view. The code is designed so that you can change this value to create a control with more or fewer selections.

The `mTempLabel` and `mTempResult` member variables provide temporary storage for the result of calculations, and are used to reduce the memory allocations while drawing.

4. As in the previous app, use a separate method to initialize the view. This `init()` helper initializes the above instance variables:

```
private void init() {  
    mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
    mTextPaint.setColor(Color.BLACK);  
    mTextPaint.setStyle(Paint.Style.FILL_AND_STROKE);  
    mTextPaint.setTextAlign(Paint.Align.CENTER);  
    mTextPaint.setTextSize(40f);  
    mDialPaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
    mDialPaint.setColor(Color.GRAY);  
    // Initialize current selection.  
    mActiveSelection = 0;  
    // TODO: Set up onClick listener for this view.  
}
```

`Paint` styles for rendering the custom view are created in the `init()` method rather than at render-time with `onDraw()`. This is to improve performance, because `onDraw()` is called frequently. (You learn more about `onDraw()` and drawing on a `Canvas` object with a `Paint` object in another lesson.)

5. Call `init()` from each constructor.
6. Since a custom view extends `View`, you can override `View` methods such as `onSizeChanged()` to control its behavior. In this case you want to determine the drawing bounds for the custom view's dial by setting its width and height, and calculating its radius, when the view size changes, which includes the first time it is drawn. Add the following to `DialView`:

```
@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    // Calculate the radius from the width and height.
    mWidth = w;
    mHeight = h;
    mRadius = (float) (Math.min(mWidth, mHeight) / 2 * 0.8);
}
```

The `onSizeChanged()` method is called when the layout is inflated and when the view has changed. Its parameters are the current width and height of the view, and the "old" (previous) width and height.

## 1.3 Draw the custom view

To draw the custom view, your code needs to render an outer grey circle to serve as the dial, and a smaller black circle to serve as the indicator. The position of the indicator is based on the user's selection captured in `mActiveSelection`. Your code must calculate the indicator position before rendering the view. After adding the code to calculate the position, override the `onDraw()` method to render the view.

The code for drawing this view is provided without explanation because the focus of this lesson is creating and using a custom view. The code uses the `Canvas` methods `drawCircle()` and `drawText()`.

1. Add the following `computeXYForPosition()` method to `DialView` to compute the X and Y coordinates for the text label and indicator (0, 1, 2, or 3) of the chosen selection, given the position number and radius:

```
private float[] computeXYForPosition
        (final int pos, final float radius) {
    float[] result = mTempResult;
    Double startAngle = Math.PI * (9 / 8d);    // Angles are in radians.
    Double angle = startAngle + (pos * (Math.PI / 4));
    result[0] = (float) (radius * Math.cos(angle)) + (mWidth / 2);
    result[1] = (float) (radius * Math.sin(angle)) + (mHeight / 2);
    return result;
}
```

The `pos` parameter is a position index (starting at 0). The `radius` parameter is for the outer circle.

You will use the `computeXYForPosition()` method in the `onDraw()` method. It returns a two-element array for the position, in which element 0 is the X coordinate, and element 1 is the Y coordinate.

2. To render the view on the screen, use the following code to override the `onDraw()` method for the view. It uses `drawCircle()` to draw a circle for the dial, and to draw the indicator mark. It uses `drawText()` to place text for labels, using a `StringBuffer` for the label text.

```
@Override  
protected void onDraw(Canvas canvas) {  
    super.onDraw(canvas);  
    // Draw the dial.  
    canvas.drawCircle(mWidth / 2, mHeight / 2, mRadius, mDialPaint);  
    // Draw the text labels.  
    final float labelRadius = mRadius + 20;  
    StringBuffer label = mTempLabel;  
    for (int i = 0; i < SELECTION_COUNT; i++) {  
        float[] xyData = computeXYForPosition(i, labelRadius);  
        float x = xyData[0];  
        float y = xyData[1];  
        label.setLength(0);  
        label.append(i);  
        canvas.drawText(label, 0, label.length(), x, y, mTextPaint);  
    }  
    // Draw the indicator mark.  
    final float markerRadius = mRadius - 35;  
    float[] xyData = computeXYForPosition(mActiveSelection,  
                                         markerRadius);  
    float x = xyData[0];  
    float y = xyData[1];  
    canvas.drawCircle(x, y, 20, mTextPaint);  
}
```

(You learn more about drawing on a `Canvas` object in another lesson.)

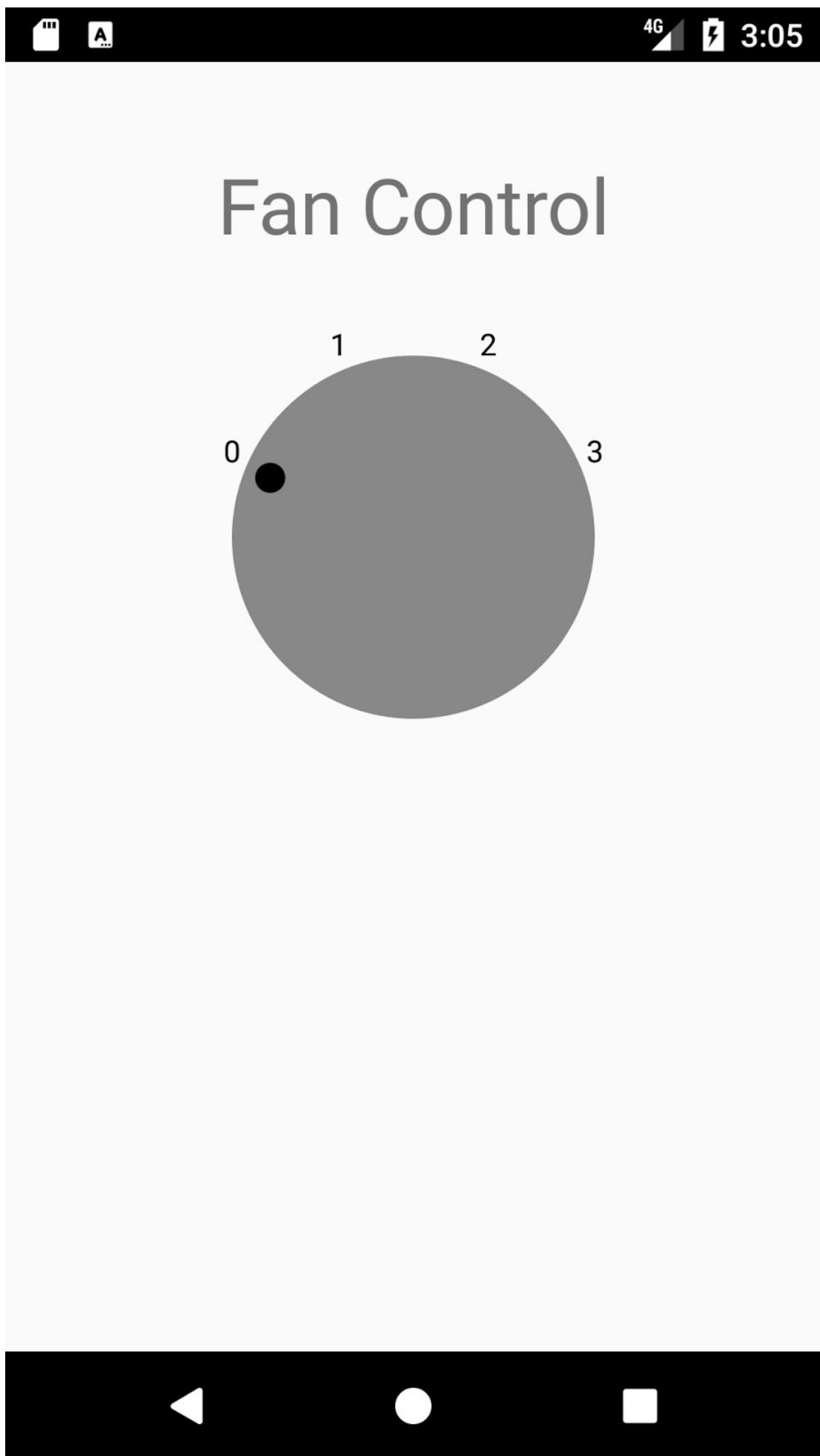
## 1.4 Add the custom view to the layout

You can now replace the `ImageView` with the custom `DialView` class in the layout, in order to see what it looks like:

1. In `activity_main.xml`, change the `ImageView` tag for the `dialView` to `com.example.customfancontroller.DialView`, and delete the `android:background` attribute.

The `DialView` class inherits the attributes defined for the original `ImageView`, so there is no need to change the other attributes.

2. Run the app.



## 1.5 Add a click listener

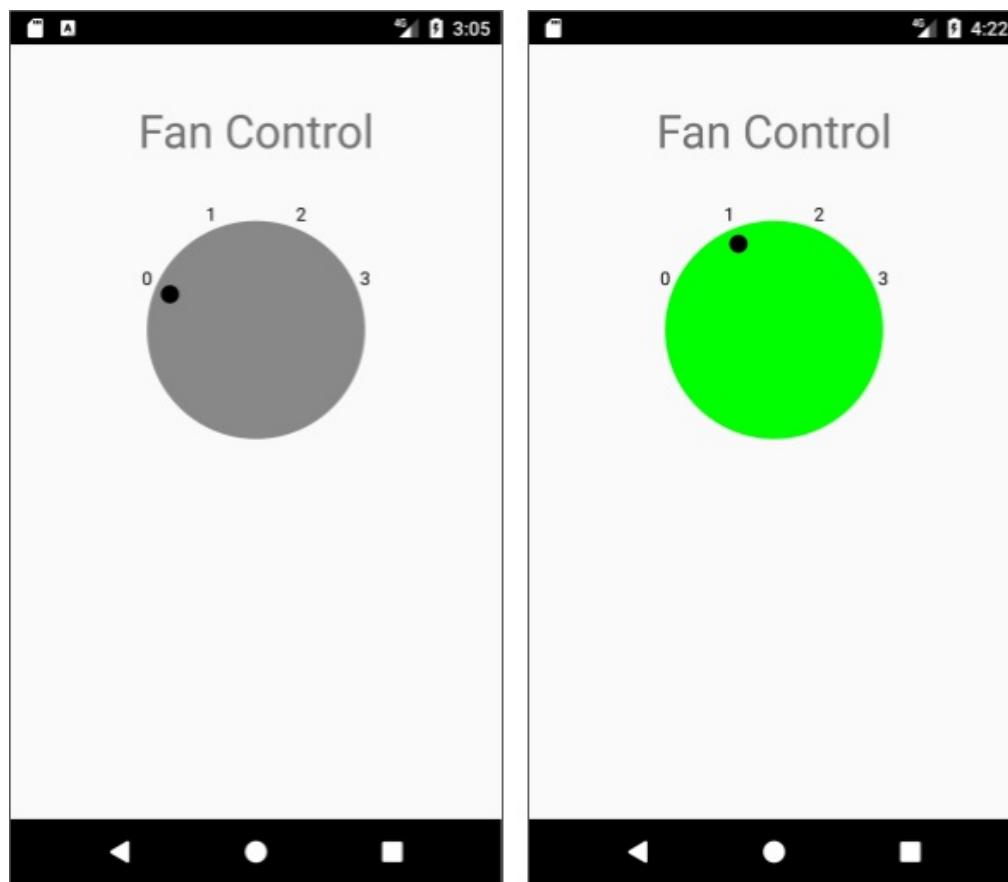
To add behavior to the custom view, add an `OnTouchListener()` to the `DialView init()` method to perform an action when the user taps the view. Each tap should move the selection indicator to the next position: 0-1-2-3 and back to 0. Also, if the selection is 1 or higher, change the background from gray to green (indicating that the fan power is on):

1. Add the following after the `TODO` comment in the `init()` method:

```
setOnTouchListener(new OnTouchListener() {
    @Override
    public void onClick(View v) {
        // Rotate selection to the next valid choice.
        mActiveSelection = (mActiveSelection + 1) % SELECTION_COUNT;
        // Set dial background color to green if selection is >= 1.
        if (mActiveSelection >= 1) {
            mDialPaint.setColor(Color.GREEN);
        } else {
            mDialPaint.setColor(Color.GRAY);
        }
        // Redraw the view.
        invalidate();
    }
});
```

The `invalidate()` method of `view` invalidates the entire view, forcing a call to `onDraw()` to redraw the view. If something in your custom view changes and the change needs to be displayed, you need to call `invalidate()`.

2. Run the app. Tap the `DialView` element to move the indicator from 0 to 1. The dial should turn green. With each tap, the indicator should move to the next position. When the indicator reaches 0, the dial should turn gray.



## Task 1 solution code

Android Studio project: [CustomFanController](#)

## Coding challenge

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

**Challenge:** Define a custom attribute for the `DialView` custom view: a `boolean` named `alternateColor` that flags whether or not to use an alternate set of colors for the dial and its indicator text labels. In the layout, add the `alternateColor` custom attribute for the custom view, and set it to `false`.

**Hint:** Use the following constructor to supply the attributes to the custom view. It uses a typed array for the attributes, and sets the alternate color attribute to false:

```
public DialView(Context context, AttributeSet attrs, int defStyleAttr) {  
    super(context, attrs, defStyleAttr);  
    // Collect the attributes into a typed array.  
    TypedArray typedArray = context.obtainStyledAttributes(attrs,  
        R.styleable.DialView, 0, 0);  
    // Use mAltColor to represent the alternate color attribute.  
    mAltColor = typedArray.getBoolean(R.styleable.DialView_alternateColor,  
        false);  
    // Recycle the array.  
    typedArray.recycle();  
    init();  
}
```

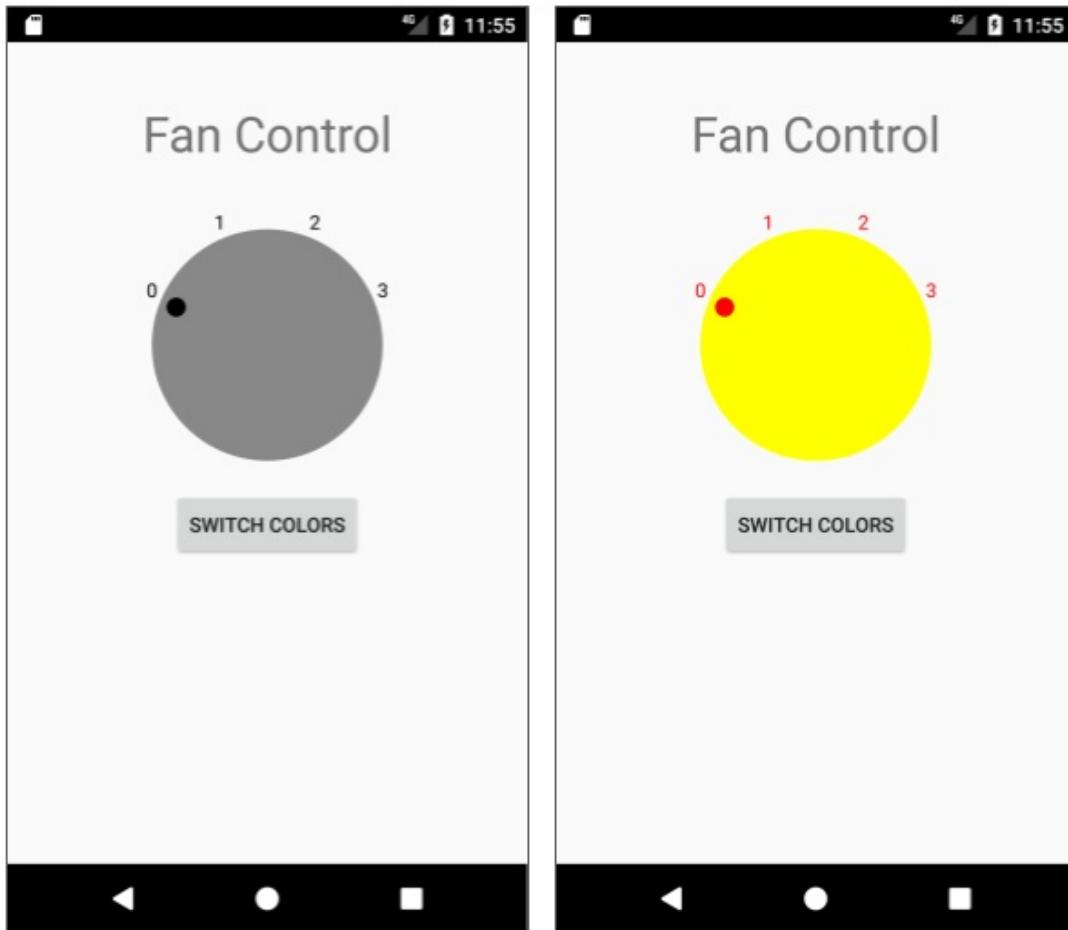
Expose a new getter and setter method in the `DialView` class to get and set the value of the `boolean` flag, and write code to decide which color set to use:

- Use alternate colors if `alternateColor` is `true` (yellow for the dial and red for the text).
- Use normal colors if `alternateColor` is `false` (gray for the dial and black for the text).

Add a **Switch Colors** button to the layout, and call the `DialView` getter and setter methods from `MainActivity`.

When you run the app, the dial is gray and the text is black. Tap **Switch Colors**, and the dial should change to yellow and the text to red. Tap **Switch Colors** again, and the dial should change back to the normal colors of gray and black. The dial should still change to green (as

before) when the user taps the controller to turn the fan on.



## Challenge solution code

Android Studio project: [CustomFanControllerChallenge](#)

## Summary

To create a custom view of any size and shape:

- Add a new class that extends `View`.
- Override `View` methods such as `onDraw()` to define the view's shape and basic appearance.
- Use `invalidate()` to force a draw or redraw of the view.

To optimize performance, assign any required values for drawing and painting *before* using them in `onDraw()`, such as in the constructor or the `init()` helper method.

Add listeners such as `View.OnClickListener` to the custom view to define the view's interactive behavior.

Add the custom view to an XML layout file with attributes to define its appearance, as you would with other UI elements.

## Related concept

The related concept documentation is [Custom views](#).

## Learn more

Android developer documentation:

- [Creating Custom Views](#)
- [Custom Components](#)
- [View](#)
- [Input Events](#)
- [onDraw\(\)](#)
- [Canvas](#)
- [drawCircle\(\)](#)
- [drawText\(\)](#)
- [Paint](#)

Video:

- [Quick Intro to Creating a Custom View in Android](#)
- [Android Custom View Tutorial](#)

# 11.1A: Creating a simple Canvas object

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Create a canvas and draw on it](#)
- [Solution code](#)
- [Summary](#)
- [Related concept](#)
- [Learn more](#)

When you want to create your own custom 2D drawings for Android, you can do so in the following ways.

1. Draw your graphics or animations on a `View` object in your layout. By using this option, the system's rendering pipeline handles your graphics—it's your responsibility to define the graphics inside the view.
2. Draw your graphics in a `Canvas` object. To use this option, you pass your `Canvas` to the appropriate class' `onDraw(Canvas)` method. You can also use the drawing methods in `Canvas`. This option also puts you in control of any animation.

Drawing to a view is a good choice when you want to draw simple graphics that don't need to change dynamically, and when your graphics aren't part of a performance-intensive app such as a game. For example, you should draw your graphics into a view when you want to display a static graphic or predefined animation, within an otherwise static app. For more information, read [Drawables](#).

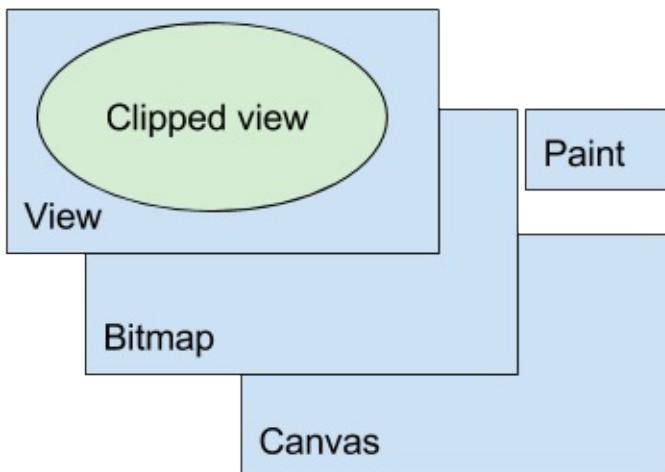
Drawing to a canvas is better when your app needs to regularly redraw itself. Apps, such as video games, should draw to the canvas on their own. This practical shows you how to create a canvas, associate it with a bitmap, and associate the bitmap with an `ImageView` for display.

When you want to draw shapes or text into a view on Android, you need:

- A `Canvas` object. Very simplified, a `Canvas` is a logical 2D drawing surface that provides methods for drawing onto a bitmap.
- An instance of the `Bitmap` class which represents the physical drawing surface and gets pushed to the display by the GPU.

- A `View` instance associated with the bitmap.
- A `Paint` object that holds the style and color information about how to draw geometries, text, and on bitmap.
- The `Canvas` class also provides methods for clipping views. *Clipping* is the action of defining geometrically what portion of the canvas the user sees in the view. This visible portion is called the *viewport* in graphics terminology.

The figure below shows all the pieces required to draw to a canvas.



You do not need a custom view to draw, as you learn in this practical. Typically you draw by overriding the `onDraw()` method of a `View`, as shown in the next practicals.

See the [Graphics Architecture](#) series of articles for an in-depth explanation of how the Android framework draws to the screen.

## What you should already KNOW

You should be able to:

- Create apps with Android Studio and run them on a physical or virtual mobile device.
- Add a click event handler to a `View`.
- Create and display a custom `View`.

## What you will LEARN

You will learn how to:

- Create a `Canvas` object, associate it with a `Bitmap` object, and display the bitmap in an `ImageView`.
- Style drawing properties with a `Paint` object.
- Draw on a canvas in response to a click event.

# What you will DO

- Create an app that draws on the screen in response to touch events.

## App overview

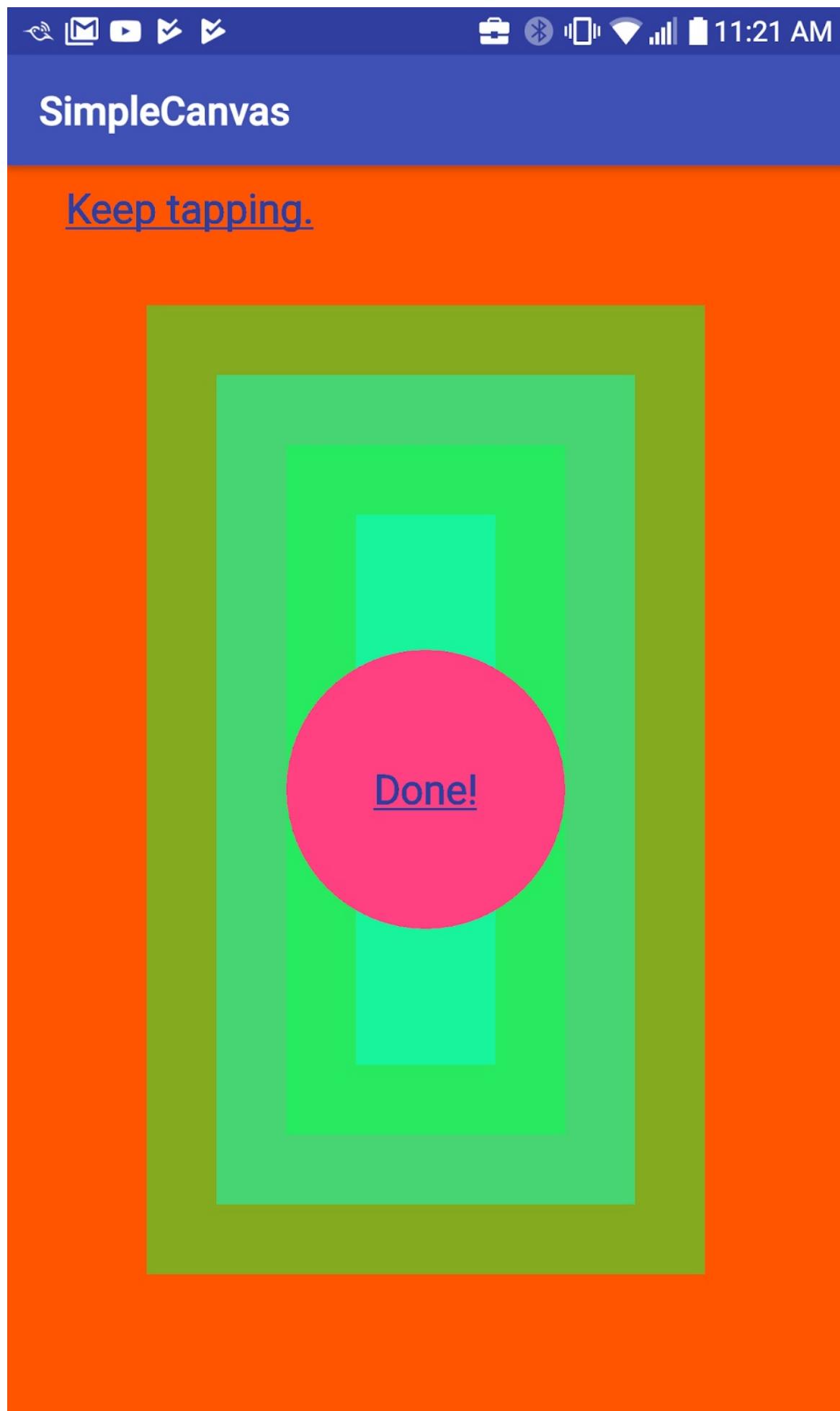
As you build the [SimpleCanvas](#) app, you learn how to create a canvas, associate it with a bitmap, and associate the bitmap with an `ImageView` for display.

When the user clicks in the app, a rectangle appears. As the user continues to click, the app draws increasingly smaller rectangles onto the canvas.

When you start the app, you see a white surface, the default background for the `ImageView`.

Tap the screen, and it fills with orange color, and the underlined text "Keep tapping" is drawn. For the next four taps, four differently colored inset rectangles are drawn. On the final tap, a circle with centered text tells you that you are "Done!", as shown in the screenshot below.

If the device is rotated, the drawing is reset, because the app does not save state. In this case, this behavior is "by design," to give you a quick way of clearing the canvas.



# Task 1. Create a canvas and draw on it

You can associate a `Canvas` with an `ImageView` and draw on it in response to user actions. This basic implementation of drawing does not require a custom `View`. You create an app with a layout that includes an `ImageView` that has a click handler. You implement the click handler in `MainActivity` to draw on and display the `Canvas`.

**Note:** The benefit of doing an example without a custom view is that you can focus on drawing on the canvas. For a real-world application, you are likely to need a custom view.

## 1.1 Create the SimpleCanvas project and layout

1. Create the SimpleCanvas project with the **Empty Activity** template.
2. In `activity_main.xml`, replace the `TextView` with an `ImageView` that fills the parent.
3. Add an `onClick` property to the `ImageView` and create a stub for the click handler called `drawSomething()`. Your XML code should look similar to this.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    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"
    tools:context="com.example.simplecanvas.MainActivity">

    <ImageView
        android:id="@+id/myimageview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:onClick="drawSomething"/>

</android.support.constraint.ConstraintLayout>
```

4. Add the following color resources to the `colors.xml` file.

```
<color name="colorRectangle">#455A64</color>
<color name="colorBackground">#FFFFD600</color>
```

5. Add the following string resources to the `strings.xml` file.

```
<string name="keep_tapping">Keep tapping.</string>
<string name="done">Done!</string>
```

## 1.2 Create the SimpleCanvas member variables and constants

In `MainActivity.java` :

1. Create a `Canvas` member variable `mCanvas`.

The `Canvas` object stores information on *what* to draw onto its associated bitmap. For example, lines, circles, text, and custom paths.

```
private Canvas mCanvas;
```

2. Create a `Paint` member variable `mPaint` and initialize it with default values.

The `Paint` objects store *how* to draw. For example, what color, style, line thickness, or text size. `Paint` offers a rich set of coloring, drawing, and styling options. You customize them below.

```
private Paint mPaint = new Paint();
```

3. Create a `Paint` object for underlined text. `Paint` offers a full complement of typographical styling methods. You can supply these styling flags when you initialize the object or set them later.

```
private Paint mPaintText = new Paint(Paint.UNDERLINE_TEXT_FLAG);
```

4. Create a `Bitmap` member variable `mBitmap`.

The `Bitmap` represents the pixels that are shown on the display.

```
private Bitmap mBitmap;
```

5. Create a member variable for the `ImageView`, `mImageView`.

A view, in this example an `ImageView`, is the container for the bitmap. Layout on the screen and all user interaction is through the view.

```
ImageView mImageView;
```

6. Create two `Rect` variables, `mRect` and `mBounds` and initialize them to rectangles.

```
private Rect mRect = new Rect();
private Rect mBounds = new Rect();
```

7. Create a constant `OFFSET` initialized to 120, and initialize a member variable `mOffset` with the constant. This offset is the distance of a rectangle you draw from the edge of the canvas.

```
private static final int OFFSET = 120;
private int mOffset = OFFSET;
```

8. Create a `MULTIPLIER` constant initialized to 100. You will need this constant later, for generating random colors.

```
private static final int MULTIPLIER = 100;
```

9. Add the following private member variables for colors.

```
private int mColorBackground;
private int mColorRectangle;
private int mColorAccent;
```

## 1.3 Fix the `onCreate` method and customize the `mPaint` member variable

In `MainActivity.java`:

1. Verify that `onCreate()` looks like the code below.

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

2. In `onCreate()`, get color resources and assign them to the color member variables.

```
mColorBackground = ResourcesCompat.getColor(getResources(),
    R.color.colorBackground, null);
mColorRectangle = ResourcesCompat.getColor(getResources(),
    R.color.colorRectangle, null);
mColorAccent = ResourcesCompat.getColor(getResources(),
    R.color.colorAccent, null);
```

3. In `onCreate()`, set the color of `mPaint` to `mColorBackground`.

```
mPaint.setColor(mColorBackground);
```

4. In `onCreate()`, set the color for `mPaintText` to the theme color `colorPrimaryDark`, and set the text size to 70. Depending on the screen size of your device, you may need to

adjust the text size.

```
mPaintText.setColor(  
    ResourcesCompat.getColor(getResources(),  
        R.color.colorPrimaryDark, null)  
);  
mPaintText.setTextSize(70);
```

5. Get a reference to the image view.

```
mImageView = (ImageView) findViewById(R.id.myimageview);
```

**Important:** You cannot create the `Canvas` in `onCreate()`, because the views have not been laid out, so their final size is not available. When you create a custom view in a later lesson, you learn different ways to initialize your drawing surface.

## 1.4 Implement the `drawSomething()` click handler method

The `drawSomething()` method is where all the interaction with the user and drawing on the canvas are implemented.

The `drawSomething()` click handler responds to user taps by drawing an increasingly smaller rectangle until it runs out of room. Then it draws a circle with the text "Done!" to demonstrate basics of drawing on canvas.

You always need to do at least the following:

1. Create `Bitmap`.
2. Associate `Bitmap` with `View`.
3. Create `Canvas` with `Bitmap`.
4. Draw on `Canvas`.
5. Call `invalidate()` on the `View` to force redraw.

Inside the `drawSomething()` method, add code as follows.

1. Create or verify the signature for the `drawSomething()` method.

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

2. Get the width and height of the view and create convenience variables for half the width and height. You must do this step every time the method is called, because the size of the view can change (for example, when the device is rotated).

```

int vWidth = view.getWidth();
int vHeight = view.getHeight();
int halfWidth = vWidth / 2;
int halfHeight = vHeight / 2;

```

3. Add an if-else statement for `(mOffset == OFFSET)`.

When `drawSomething()` is called, the app is in one of three states:

4. `mOffset == OFFSET`. The app is only in this state the first time the user taps. Create the `Bitmap`, associate it with the `View`, create the `Canvas`, fill the background, and draw some text. Increase the offset.
5. `mOffset != OFFSET` and the offset is smaller than half the screen width and height. Draw a rectangle with a computed color and increase the offset.
6. `mOffset != OFFSET` and the offset is equal to or larger than half the screen width and height. Draw a circle with the text "Done!".

```

if (mOffset == OFFSET) {
} else {
    if (mOffset < halfWidth && mOffset < halfHeight) {
    } else {
    }
}

```

7. Inside the outer `if` statement `(mOffset == OFFSET)`, create a `Bitmap`.
8. Supply the width and height for the bitmap, which are going to be the same as the width and height of the view.
9. Pass in a `Bitmap.Config` configuration object. A bitmap configuration describes how pixels are stored. How pixels are stored affects the quality (color depth) as well as the ability to display transparent/translucent colors. The `ARGB_8888` format supports `Alpha`, Red, Green, and Blue channels for each pixel. Each color is encoded in 8 bits, for a total of 4 bytes per pixel.

```
mBitmap = Bitmap.createBitmap(vWidth, vHeight, Bitmap.Config.ARGB_8888);
```

10. Associate the bitmap with the `ImageView`.

```
mImageView.setImageBitmap(mBitmap);
```

11. Create a `Canvas` and associate it with `mBitmap`, so that drawing on the canvas draws on the bitmap.

```
mCanvas = new Canvas(mBitmap);
```

12. Fill the entire canvas with the background color.

```
mCanvas.drawColor(mColorBackground);
```

13. Draw the "Keep tapping" text onto the canvas. You need to supply a string, x and y positions, and a `Paint` object for styling.

```
mCanvas.drawText(getString(R.string.keep_tapping), 100, 100, mPaintText);
```

14. Increase the offset.

```
mOffset += OFFSET;
```

15. At the end of the `drawSomething()` method, `invalidate()` the view so that the system redraws the view every time `drawSomething()` is executed.

When a view is invalidated, the system does not draw the view with the values it already has. Instead, the system recalculates the view with the new values that you supply. The screen refreshes 60 times a second, so the view is drawn 60 times per second. To save work and time, the system can reuse the existing view until it is told that the view has changed, the existing view is invalid, and the system thus has to recalculate an updated version of the view.

```
view.invalidate();
```

**Note:** If you run the app at this point, it should start with a blank screen, and when you tap, the screen fills and the text appears.

16. In the `else` block, inside the `if` statement
17. Set the color of `mPaint`. This code generates the next color by subtracting the current offset times a multiplier from the original color. A color is represented by a single number, so you can manipulate it in this way for some fun effects.
18. Change the size of the `mRect` rectangle to the width of the view, minus the current offset.
19. Draw the rectangle with `mPaint` styling.
20. Increase the offset.

Below is the complete if portion of the code.

```

if (mOffset < halfWidth && mOffset < halfHeight) {
    // Change the color by subtracting an integer.
    mPaint.setColor(mColorRectangle - MULTIPLIER*mOffset);
    mRect.set(
        mOffset, mOffset, vWidth - mOffset, vHeight - mOffset);
    mCanvas.drawRect(mRect, mPaint);
    // Increase the indent.
    mOffset += OFFSET;
}

```

1. In the `else` statement, when the offset is too large to draw another rectangle:
2. Set the color of `mPaint`.
3. Draw a circle with the paint.
4. Get the "Done" string and calculate its bounding box, then calculate x and y to draw the text at the center of the circle. The bounding box defines a rectangle that encloses the string. You cannot make calculations on a string, but you can use the dimensions of the bounding box to calculate its center.

```

else {
    mPaint.setColor(mColorAccent);
    mCanvas.drawCircle(halfWidth, halfHeight, halfWidth / 3, mPaint);
    String text = getString(R.string.done);
    // Get bounding box for text to calculate where to draw it.
    mPaintText.getTextBounds(text, 0, text.length(), mBounds);
    // Calculate x and y for text so it's centered.
    int x = halfWidth - mBounds.centerX();
    int y = halfHeight - mBounds.centerY();
    mCanvas.drawText(text, x, y, mPaintText);
}

```

5. Run your app and tap multiple times to draw. Rotate the screen to reset the app.

## Solution code

Android Studio project: [SimpleCanvas](#).

## Summary

- To draw on the display of a mobile device with Android you need a `View`, a `Canvas`, a `Paint`, and a `Bitmap` object.
- The `Bitmap` is the physical drawing surface. The `Canvas` provides an API to draw on the bitmap, the `Paint` is for styling what you draw, and the `View` displays the `Bitmap`.

- You create a `Bitmap`, associate it with a `View`, create a `Canvas` with a `Paint` object for the `Bitmap`, and then you can draw.
- You must `invalidate()` the view when you are done drawing, so that the Android System redraws the display.
- All drawing happens on the UI thread, so performance matters.

## Related concepts

The related concept documentation is in [The Canvas class](#).

## Learn more

Android developer documentation:

- [Canvas class](#)
- [Bitmap class](#)
- [View class](#)
- [Paint class](#)
- [Bitmap.config configurations](#)
- [Canvas and Drawables](#)
- [Graphics Architecture series of articles \(advanced\)](#)

# 11.1B: Drawing on a Canvas object

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Create a canvas and respond to user events
- Solution code
- Coding challenge
- Summary
- Related concept
- Learn more

In a previous practical, you learned the fundamentals of 2D custom drawing in Android by drawing on a `Canvas` in response to user input.

A more common pattern for using the `Canvas` class is to subclass one of the `View` classes, override its `onDraw()` and `onSizeChanged()` methods to draw, and override the `onTouchEvent()` method to handle user touches.

In this practical, you write an app that uses that pattern.

## What you should already KNOW

You should be able to:

- Create apps with Android Studio and run them on a physical or virtual mobile device.
- Add event handlers to views.
- Create a custom `View`.
- Create a bitmap and associate it with a view. Create a canvas for a bitmap. Create and customize a `Paint` object for styling. Draw on the `canvas` and refresh the display.

## What you will LEARN

You will learn how to:

- Create a custom `View`, capture the user's motion event, and interpret it to draw lines onto the canvas.

## What you will DO

- Create an app that draws lines on the screen in response to motion events.

## App overview

The **CanvasExample** uses a custom view to display a line in response to user touches, as shown in the screenshot below.



# Task 1. Create a canvas and respond to user events

## 1.1 Create the CanvasExample project

1. Create a **CanvasExample** project with the **Empty Activity** template. Do not add a layout file as you won't need it.
2. Add the following two colors to the colors.xml file.

```
<color name="opaque_orange">#FFFF5500</color>
<color name="opaque_yellow">#FFFEB3B</color>
```

3. In `styles.xml`, set the parent of the default style to `NoActionBar` to remove the action bar, so that you can draw fullscreen.

```
<style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
```

## 1.2 Create the MyCanvasView class

1. In a separate file, create a new class called `MyCanvasView`.
2. Make the `MyCanvasView` class extend the `View` class.
3. Add member variables for `Canvas`, `Bitmap`, `Paint`, and `Path` objects. Import `android.graphics.Path` for the `Path`. The `path` holds the path that you are currently drawing while the user moves their finger across the screen. Add an `int` variable `mDrawColor`.

```
private Canvas mCanvas;
private Bitmap mBitmap;
private Paint mPaint;
private Path mPath;
private int mDrawColor;
```

4. Add constructors to initialize the `mPath`, `mPaint`, and `mDrawColor` variables. (You only need these two constructors of all that are available.)

- o `Paint.Style` specifies if the primitive being drawn is filled, stroked, or both (in the same color).
- o `Paint.Join` specifies how lines and curve segments join on a stroked path.
- o `Paint.Cap` specifies how the beginning and ending of stroked lines and paths.
- o See [Paint documentation](#) for a list of attributes that can be set.

Here is the code:

```
MyCanvasView(Context context) {  
    this(context, null);  
}  
  
public MyCanvasView(Context context, AttributeSet attributeSet) {  
    super(context);  
  
    int backgroundColor;  
    mDrawColor = ResourcesCompat.getColor(getResources(),  
        R.color.opaque_orange, null);  
    backgroundColor = ResourcesCompat.getColor(getResources(),  
        R.color.opaque_yellow, null);  
    // Holds the path we are currently drawing.  
    mPath = new Path();  
    // Set up the paint with which to draw.  
    mPaint = new Paint();  
    mPaint.setColor(backgroundColor);  
    // Smoothes out edges of what is drawn without affecting shape.  
    mPaint.setAntiAlias(true);  
    // Dithering affects how colors with higher-precision than the device  
    // are down-sampled.  
    mPaint.setDither(true);  
    mPaint.setStyle(Paint.Style.STROKE); // default: FILL  
    mPaint.setStrokeJoin(Paint.Join.ROUND); // default: MITER  
    mPaint.setStrokeCap(Paint.Cap.ROUND); // default: BUTT  
    mPaint.setStrokeWidth(12); // default: Hairline-width (really thin)  
}
```

5. In the `onCreate()` method of `MainActivity` :

- Create a variable `myCanvasView` of type `MyCanvasView` .
- Create an instance of `MyCanvasView` and assign it to `myCanvasView` .
- Set the `SYSTEM_UI_FLAG_FULLSCREEN` flag on `myCanvasView` so that the app fills the screen.
- Set the `myCanvasView` view as the content view. You cannot get the size of the view in the `onCreate()` method.

Here is the code:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    MyCanvasView myCanvasView;
    // No XML file; just one custom view created programmatically.
    myCanvasView = new MyCanvasView(this);
    // Request the full available screen for layout.
    myCanvasView.setSystemUiVisibility(SYSTEM_UI_FLAG_FULLSCREEN);
    setContentView(myCanvasView);
}

```

6. In `MyCanvasView`, override the `onSizeChanged()` method.

The `onSizeChanged()` method is called whenever a view changes size. Because the view starts out with no size, the `onSizeChanged()` method is also called after the activity first inflates the view. This method is thus the ideal place to create and set up the canvas.

Create a `Bitmap`, create a `Canvas` with the `Bitmap`, and fill the `Canvas` with color.

```

@Override
protected void onSizeChanged(int width, int height,
                            int oldWidth, int oldHeight) {
    super.onSizeChanged(width, height, oldWidth, oldHeight);
    // Create bitmap, create canvas with bitmap, fill canvas with color.
    mBitmap = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
    mCanvas = new Canvas(mBitmap);
    mCanvas.drawColor(mDrawColor);
}

```

7. In `MyCanvasView`, override the `onDraw()` method.

All the drawing work for `MyCanvasView` happens in the `onDraw()` method.

- Draw the colored bitmap.
- Then draw the path on top of it. You will set `mPath` in response to user motion in the next series of steps.

Here is the code:

```
@Override  
protected void onDraw(Canvas canvas) {  
    super.onDraw(canvas);  
    // First, draw the bitmap as created.  
    canvas.drawBitmap(mBitmap, 0, 0, mPaint);  
    // Then draw the path on top, styled by mPaint.  
    canvas.drawPath(mPath, mPaint);  
}
```

8. Run your app. The whole screen should be filled with orange color.

## 1.3 Respond to motion on the display

The `onTouchEvent()` method of the view is called whenever the user touches the display.

In the `MyCanvasView` class:

1. Override the `onTouchEvent()` method.
  - Get the x and y coordinates of the event.
  - Use a `switch` statement to handle the events you are interested in. There are many more touch events available. See the `MotionEvent` class documentation for a full list.
  - Call a utility method for each type of event. You implement those methods next.
  - You must call `invalidate()` to redraw the view after it changes. You call it inside the `case` statement because you do not want to call `invalidate()` when the event is not one of interest.

Here is the code:

```

@Override
public boolean onTouchEvent(MotionEvent event) {
    float x = event.getX();
    float y = event.getY();

    // Invalidate() is inside the case statements because there are many
    // other types of motion events passed into this listener,
    // and we don't want to invalidate the view for those.
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            touchStart(x, y);
            // No need to invalidate because we are not drawing anything.
            break;
        case MotionEvent.ACTION_MOVE:
            touchMove(x, y);
            invalidate();
            break;
        case MotionEvent.ACTION_UP:
            touchUp();
            invalidate();
            break;
        default:
            // do nothing
    }
    return true;
}

```

2. Add member variables to hold the latest x and y values, which are the starting point for the next path.

```
private float mX, mY;
```

3. Add a `TOUCH_TOLERANCE` `float` constant and set it to 4. This tolerance serves two functions:

- If the finger has barely moved, there is no need to draw.
- Using the path, it is not necessary to draw every pixel and request a refresh of the display. Instead, you can interpolate for much better performance.

Here is the code:

```
private static final float TOUCH_TOLERANCE = 4;
```

4. Implement the `touchStart()` method.

- When the user starts to draw a new line, set the beginning of the contour (line) to x, y and save the beginning coordinates.

Here is the code:

```
private void touchStart(float x, float y) {
    mPath.moveTo(x, y);
    mX = x;
    mY = y;
}
```

5. Add the `touchMove()` method.

- Calculate the distance that has been moved (`dx`, `dy`).
- If the movement was further than the touch tolerance, add a segment to the path.
- Set the starting point for the next segment to the endpoint of this segment.
- Using `quadTo()` instead of `lineTo()` creates a smoothly drawn line without corners. See [Bezier Curves](#).

Here is the code:

```
private void touchMove(float x, float y) {
    float dx = Math.abs(x - mX);
    float dy = Math.abs(y - mY);
    if (dx >= TOUCH_TOLERANCE || dy >= TOUCH_TOLERANCE) {
        // QuadTo() adds a quadratic bezier from the last point,
        // approaching control point (x1,y1), and ending at (x2,y2).
        mPath.quadTo(mX, mY, (x + mX)/2, (y + mY)/2);
        mX = x;
        mY = y;
    }
}
```

6. Finally, add the `touchUp()` method.

- Finish up by drawing a line to the last coordinate.
- Draw the path.
- Reset the path so it doesn't get drawn again when you draw more.

Here is the code:

```
private void touchUp() {
    mPath.lineTo(mX, mY);
    mCanvas.drawPath(mPath, mPaint);
    // Reset so it doesn't get drawn again.
    mPath.reset();
}
```

7. Run your app. When the app opens, use your finger to draw. (Rotate the device to clear the screen.)

## Solution code

Android Studio project: [CanvasExample](#)

## Coding challenge

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

- Create an app that lets the user draw overlapping rectangles. First, implement it so that tapping the screen creates the rectangles.
- Add functionality where the rectangle starts at a very small size. If the user drags their finger, let the rectangle increase in size until the user lifts the finger off the screen.

## Summary

- A common pattern for working with a canvas is to create a custom view and override the `onDraw()` and `onSizeChanged()` methods.
- Override the `onTouchEvent()` method to capture user touches and respond to them by drawing things.

## Related concepts

The related concept documentation is in [The Canvas class](#).

## Learn more

Android developer documentation:

- [Canvas class](#)
- [Bitmap class](#)
- [View class](#)
- [Paint class](#)
- [Bitmap.config configurations](#)
- [Path class](#)
- [Bezier curves Wikipedia page](#)
- [Canvas and Drawables](#)
- [Graphics Architecture series of articles \(advanced\)](#)



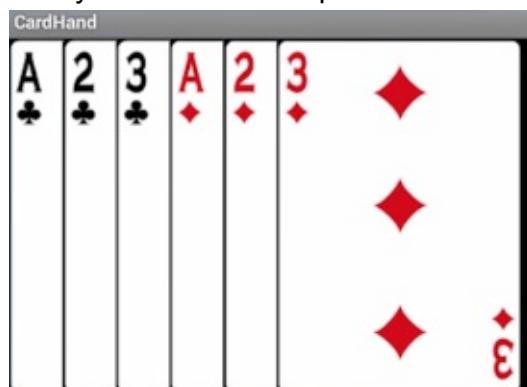
# 11.1C: Applying clipping to a Canvas object

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Create an app that demonstrates clipping regions
- Solution code
- Summary
- Related concept
- Learn more

For the purpose of this practical, [clipping](#) is a method for defining regions of an image, canvas, or bitmap that are selectively drawn or not drawn onto the screen. One purpose of clipping is to reduce [overdraw](#). You can also use clipping to create interesting effects in user interface design and animation.

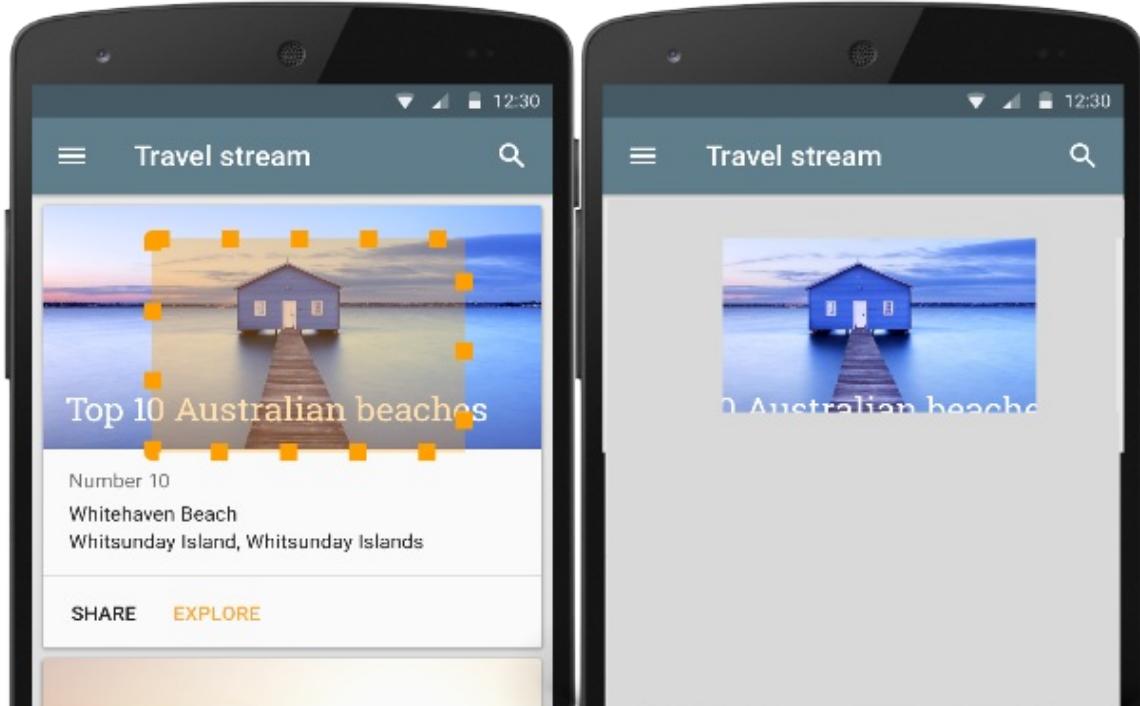
For example, when you draw a stack of overlapping cards as shown below, instead of fully drawing each card, it is usually more efficient to only draw the visible portions. "Usually",



because clipping operations also have a cost.

You do this by specifying a *clipping region* for each card. For example in the diagram below, when a *clipping rectangle* is applied to an image, only the portion inside that rectangle is displayed. The clipping region is commonly a rectangle, but it can be any shape or combination of shapes. You can also specify whether you want the region inside the clipping

region included or excluded. The screenshot below shows an example. When a clipping rectangle is applied to an image, only the portion inside that rectangle is displayed.



## What you should already KNOW

You should be able to:

- Create apps with Android Studio and run them on a physical or virtual mobile device.
- Add event handlers to views.
- Create a custom `View`.
- Create and draw on a `Canvas`.
- Create a `Bitmap` and associate it with a `View`; create a `Canvas` for a `Bitmap`; create and customize a `Paint` object for styling; draw on the canvas and refresh the display.
- Create a custom `View`, override `onDraw()` and `onSizeChanged()`.

## What you will LEARN

You will learn how to:

- Apply different kinds of clipping to a canvas.
- How to save and restore drawing states of a canvas.

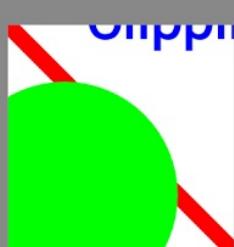
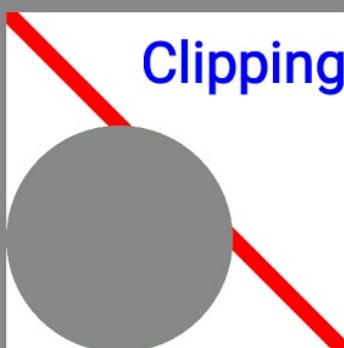
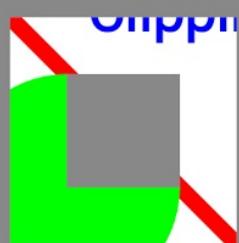
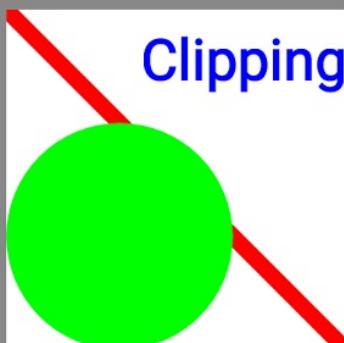
## What you will DO

- Create an app that draws clipped shapes on the screen.

## App overview

The **ClippingExample** app demonstrates how you can use and combine shapes to specify which portions of a canvas are displayed in a view.

## Clipping Example



Skewed and Translated Text

# Task 1. Create an app that demonstrates clipping regions

## 1.1 Create the ClippingExample project

1. Create the ClippingExample app with the **Empty Activity** template. Uncheck **Generate layout file** as you don't need it.
2. In the `MainActivity` class, in the `onCreate()` method, set the content view to a new instance of `ClippedView`.

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(new ClippedView(this));  
}
```

3. Create a new class for a custom view called `ClippedView` which extends `View`. The rest of the work will all be inside `ClippedView`.

```
private static class ClippedView extends View {...}
```

## 1.2 Add convenience variables for the ClippedView class

1. Define member variables `mPaint` and `mPath` in the `ClippedView` class.

```
private Paint mPaint;  
private Path mPath;
```

2. For the app to look correct on smaller screens, define dimensions for the smaller screen in the default `dimens.xml` file.

```
<dimen name="clipRectRight">90dp</dimen>
<dimen name="clipRectBottom">90dp</dimen>
<dimen name="clipRectTop">0dp</dimen>
<dimen name="clipRectLeft">0dp</dimen>

<dimen name="rectInset">8dp</dimen>
<dimen name="smallRectOffset">40dp</dimen>

<dimen name="circleRadius">30dp</dimen>
<dimen name="textOffset">20dp</dimen>
<dimen name="strokeWidth">4dp</dimen>

<dimen name="textSize">18sp</dimen>
```

3. Create a `values-sw480dp` folder and define values for the larger screens in `dimens.xml` in the `values-sw480dp` folder. (Note: If the empty folder does not show up in Android Studio, manually add a resource file to the `ClippingExample/app/src/main/res/values-sw480dp` directory. This makes the folder show in your **Project** pane.)

```
<dimen name="clipRectRight">120dp</dimen>
<dimen name="clipRectBottom">120dp</dimen>

<dimen name="rectInset">10dp</dimen>
<dimen name="smallRectOffset">50dp</dimen>

<dimen name="circleRadius">40dp</dimen>
<dimen name="textOffset">25dp</dimen>
<dimen name="strokeWidth">6dp</dimen>
```

4. In `clippedView`, add convenience member variables for dimensions, so that you only have to fetch the resources once.

```

private int mClipRectRight =
    (int) getResources().getDimension(R.dimen.clipRectRight);
private int mClipRectBottom =
    (int) getResources().getDimension(R.dimen.clipRectBottom);
private int mClipRectTop =
    (int) getResources().getDimension(R.dimen.clipRectTop);
private int mClipRectLeft =
    (int) getResources().getDimension(R.dimen.clipRectLeft);
private int mRectInset =
    (int) getResources().getDimension(R.dimen.rectInset);
private int mSmallRectOffset =
    (int) getResources().getDimension(R.dimen.smallRectOffset);

private int mCircleRadius =
    (int) getResources().getDimension(R.dimen.circleRadius);

private int mTextOffset =
    (int) getResources().getDimension(R.dimen.textOffset);
private int mTextSize =
    (int) getResources().getDimension(R.dimen.textSize);

```

5. In `clippedView`, add convenience member variables for row and column coordinates so that you only have to calculate them once.

```

private int mColumnOne = mRectInset;
private int mColumnTwo = mColumnOne + mRectInset + mClipRectRight;

private int mRowOne = mRectInset;
private int mRowTwo = mRowOne + mRectInset + mClipRectBottom;
private int mRowThree = mRowTwo + mRectInset + mClipRectBottom;
private int mRowFour = mRowThree + mRectInset + mClipRectBottom;
private int mTextRow = mRowFour + (int)(1.5 * mClipRectBottom);

```

6. In `clippedView`, add a private final member variable for a rectangle of type `RectF`:

```
private final RectF mRectF;
```

## 1.3 Add constructors for the ClippedView class

- Add a constructor that initializes the `Paint` and `Path` objects for the canvas.

Note that the `Paint.Align` property specifies which side of the text to align to the origin (not which side of the origin the text goes, or where in the region it is aligned!). Aligning the right side of the text to the origin places it on the left of the origin.

```
public ClippedView(Context context) {
    this(context, null);
}

public ClippedView(Context context, AttributeSet attributeSet) {
    super(context, attributeSet);
    setFocusable(true);
    mPaint = new Paint();
    // Smooth out edges of what is drawn without affecting shape.
    mPaint.setAntiAlias(true);
    mPaint.setStrokeWidth(
        (int) getResources().getDimension(R.dimen.strokeWidth));
    mPaint.setTextSize((int) getResources().getDimension(R.dimen.textSize));
    mPath = new Path();

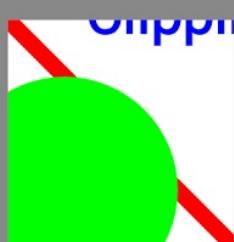
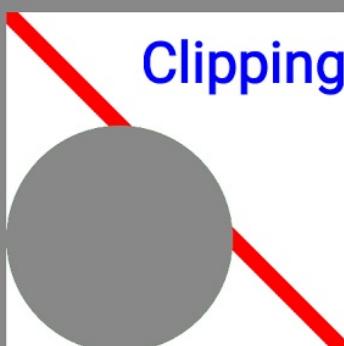
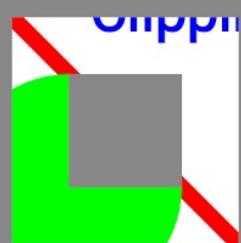
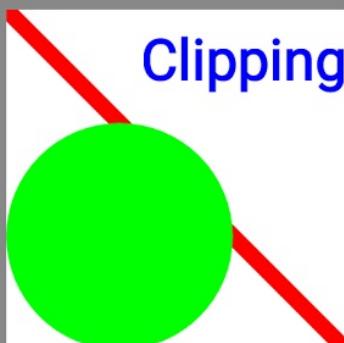
    mRectF = new RectF(new Rect(mRectInset, mRectInset,
        mClipRectRight-mRectInset, mClipRectBottom-mRectInset));
}
```

2. Run your app to make sure the code is correct. You should see the name of the app and a white screen.

## 1.4 Understand the drawing algorithm

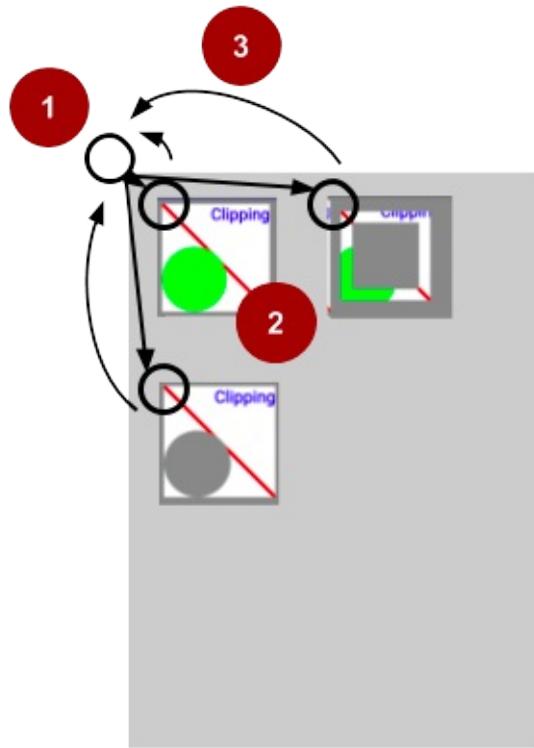
In `onDraw()`, you define seven different clipped rectangles as shown in the app screenshot below. The rectangles are all drawn the same way; the only difference is their defined clipping regions.

## Clipping Example



Skewed and Translated Text

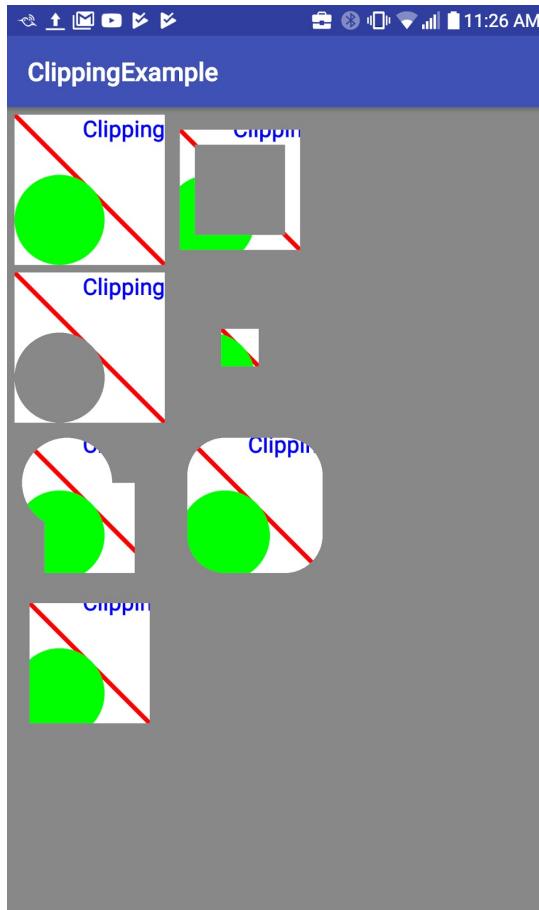
The algorithm used to draw the rectangles works as shown in the screenshot and explanation below. In summary, drawing a series of rectangles by moving the origin of the `Canvas`. (1) Translate `Canvas`. (2) Draw rectangle. (3) Restore `Canvas` and `Origin`.



1. Fill the `Canvas` with the gray background color.
2. Save the current state of the `Canvas` so you can reset to that initial state.
3. Translate the `origin` of the canvas to the location where you want to draw the next rectangle. That is, instead of calculating where the next rectangle and all the other shapes need to be drawn, you move the `Canvas origin`, that is, its coordinate system, and then draw the shapes at the same location in the translated coordinate system. This is simpler and slightly more efficient.
4. Apply clipping shapes and paths.
5. Draw the rectangle.
6. Restore the state of the `Canvas`.
7. GOTO Step 2 and repeat until all rectangles are drawn.

## 1.5 Add a helper method to draw clipped rectangles

The app draws the rectangle below seven times, first with no clipping, then six time with



various clipping paths applied.

The `drawClippedRectangle()` method factors out the code for drawing one rectangle.

1. Create a `drawClippedRectangle()` method that takes a `Canvas` argument.

```
private void drawClippedRectangle(Canvas canvas) {...}
```

2. Apply a clipping rectangle that constraints to drawing only the square to the `canvas`.

```
canvas.clipRect(mClipRectLeft, mClipRectTop,
    mClipRectRight, mClipRectBottom);
```

The `Canvas.clipRect(left, top, right, bottom)` method reduces the region of the screen that future draw operations can write to. It sets the clipping boundaries (`clipBounds`) to be the spatial intersection of the current clipping rectangle and the rectangle specified. There are lot of variants of the `clipRect()` method that accept different forms for regions and allow different operations on the clipping rectangle.

3. Fill the `canvas` with white color. Because of the clipping rectangle, only the region defined by the clipping rectangle is filled, creating a white rectangle.

```
canvas.drawColor(Color.WHITE);
```

4. Draw the red line, green circle, and text, as shown in the completed method below.
5. After you paste the code, create a string resource "clipping" to get rid of the error for `R.string.clipping` in the last line.

```
private void drawClippedRectangle(Canvas canvas) {  
    // Set the boundaries of the clipping rectangle for whole picture.  
    canvas.clipRect(mClipRectLeft, mClipRectTop,  
                    mClipRectRight, mClipRectBottom);  
  
    // Fill the canvas with white.  
    // With the clipped rectangle, this only draws  
    // inside the clipping rectangle.  
    // The rest of the surface remains gray.  
    canvas.drawColor(Color.WHITE);  
  
    // Change the color to red and  
    // draw a line inside the clipping rectangle.  
    mPaint.setColor(Color.RED);  
    canvas.drawLine(mClipRectLeft, mClipRectTop,  
                   mClipRectRight, mClipRectBottom, mPaint);  
  
    // Set the color to green and  
    // draw a circle inside the clipping rectangle.  
    mPaint.setColor(Color.GREEN);  
    canvas.drawCircle(mCircleRadius, mClipRectBottom - mCircleRadius,  
                      mCircleRadius, mPaint);  
  
    // Set the color to blue and draw text aligned with the right edge  
    // of the clipping rectangle.  
    mPaint.setColor(Color.BLUE);  
    // Align the RIGHT side of the text with the origin.  
    mPaint.setTextAlign(Paint.Align.RIGHT);  
    canvas.drawText(getContext().getString(R.string.clipping),  
                   mClipRectRight, mTextOffset, mPaint);  
}
```

6. If you run your app, you still only see the white screen, because you have not overridden `onDraw()` and thus are not drawing anything yet.

## 1.6 Override the `onDraw()` method

In the `onDraw()` method you apply various combinations of clipping regions to achieve graphical effects and learn how you can combine clipping regions to create any shape you need.

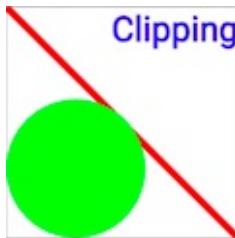
When you use `View` classes provided by the Android system, the system clips views for you to minimize overdraw. When you use custom `View` classes and override the `onDraw()` method, clipping what you draw becomes your responsibility.

**WARNING:**For Android O some `clipPath()` methods have been replaced with `clipOutPath()` and `clipOutRect()` methods, and some `Region.Op` operators have been deprecated. Depending on the version of Android you are using, you may need to adjust the methods and operators you use to create the exact effects shown in the app screenshot. See the [Canvas](#) and documentation for details.

1. Create the `onDraw()` method, if it is not already present as a code stub.

```
@Override protected void onDraw(Canvas canvas) { ... }
```

Next, add code to draw the first rectangle, which has no additional clipping.



2. In `onDraw()`, fill the `canvas` with gray color.

```
canvas.drawColor(Color.GRAY);
```

3. Save the drawing state of the `canvas`.

Context maintains a stack of drawing states. Each state includes the currently applied transformations and clipping regions. Undoing a transformation by reversing it is error-prone, as well as chaining too many transformations relative to each other. Translation is straightforward to reverse, but if you also stretch, rotate, or custom deform, it gets complex quickly. Instead, you save the state of the canvas, apply your transformations, draw, and then restore the previous state.

```
canvas.save();
```

4. Translate the origin of the canvas to the top-left corner of the first rectangle.

```
canvas.translate(mColumnOne, mRowOne);
```

5. Call the `drawClippedRectangle()` method to draw the first rectangle.

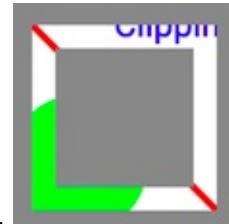
```
drawClippedRectangle(canvas);
```

6. Restore the previous state of the canvas.

```
canvas.restore();
```

7. Run your app. You should now see the first rectangle drawn on a gray background.

Next, add code to draw the second rectangle, which uses the difference between two



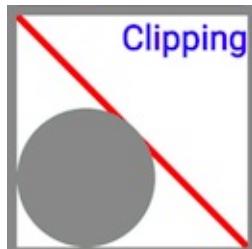
clipping rectangles to create a picture frame effect.

Use the code below which does the following:

1. Save the canvas.
2. Translate the origin of the canvas into open space to the right of the first rectangle.
3. Apply two clipping rectangles. The `DIFFERENCE` operator subtracts the second rectangle from the first one.
4. Call the `drawClippedRectangle()` method to draw the modified canvas.
5. Restore the canvas state.
6. Run your app.

```
// Draw a rectangle that uses the difference between two
// clipping rectangles to create a picture frame effect.
canvas.save();
// Move the origin to the right for the next rectangle.
canvas.translate(mColumnTwo, mRowOne);
// Use the subtraction of two clipping rectangles to create a frame.
canvas.clipRect(2 * mRectInset, 2 * mRectInset,
    mClipRectRight-2 * mRectInset, mClipRectBottom-2 * mRectInset);
canvas.clipRect(4 * mRectInset, 4 * mRectInset,
    mClipRectRight-4 * mRectInset, mClipRectBottom-4 * mRectInset,
    Region.Op.DIFFERENCE);
drawClippedRectangle(canvas);
canvas.restore();
```

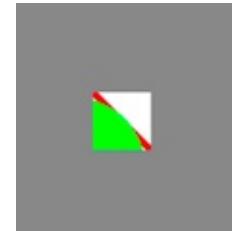
Next, add code to draw the third rectangle, which uses a circular clipping region created



from a circular path.

Here is the code:

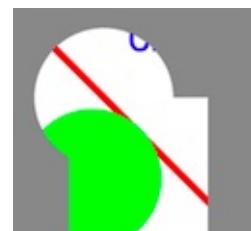
```
// Draw a rectangle that uses a circular clipping region
// created from a circular path.
canvas.save();
canvas.translate(mColumnOne, mRowTwo);
// Clears any lines and curves from the path but unlike reset(),
// keeps the internal data structure for faster reuse.
mPath.rewind();
mPath.addCircle(mCircleRadius, mClipRectBottom-mCircleRadius,
    mCircleRadius, Path.Direction.CCW);
canvas.clipPath(mPath, Region.Op.DIFFERENCE);
drawClippedRectangle(canvas);
canvas.restore();
```



Next, add code to draw the intersection of two clipping rectangles.

Here is the code:

```
// Use the intersection of two rectangles as the clipping region.
canvas.save();
canvas.translate(mColumnnTwo, mRowTwo);
canvas.clipRect(mClipRectLeft, mClipRectTop,
    mClipRectRight-mSmallRectOffset,
    mClipRectBottom-mSmallRectOffset);
canvas.clipRect(mClipRectLeft+mSmallRectOffset,
    mClipRectTop+mSmallRectOffset,
    mClipRectRight, mClipRectBottom, Region.Op.INTERSECT);
drawClippedRectangle(canvas);
canvas.restore();
```



Next, combine shapes and draw any path to define a clipping region.

Here is the code:

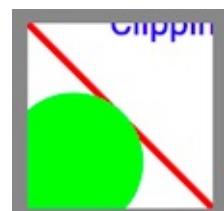
```
// You can combine shapes and draw any path to define a clipping region.  
canvas.save();  
canvas.translate(mColumnOne, mRowThree);  
mPath.rewind();  
mPath.addCircle(mClipRectLeft+mRectInset+mCircleRadius,  
    mClipRectTop+mCircleRadius+mRectInset,  
    mCircleRadius, Path.Direction.CCW);  
mPath.addRect(mClipRectRight/2-mCircleRadius,  
    mClipRectTop+mCircleRadius+mRectInset,  
    mClipRectRight/2+mCircleRadius,  
    mClipRectBottom-mRectInset, Path.Direction.CCW);  
canvas.clipPath(mPath);  
drawClippedRectangle(canvas);  
canvas.restore();
```

Next, add a rounded rectangle which is a commonly used clipping shape:



Here is the code:

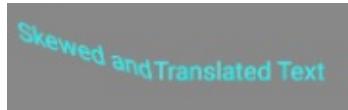
```
// Use a rounded rectangle. Use mClipRectRight/4 to draw a circle.  
canvas.save();  
canvas.translate(mColumnTwo, mRowThree);  
mPath.rewind();  
mPath.addRoundRect(mRectF, (float)mClipRectRight/4,  
    (float)mClipRectRight/4, Path.Direction.CCW);  
canvas.clipPath(mPath);  
drawClippedRectangle(canvas);  
canvas.restore();
```



Next, clip the outside around the rectangle.

Here is the code:

```
// Clip the outside around the rectangle.  
canvas.save();  
// Move the origin to the right for the next rectangle.  
canvas.translate(mColumnOne, mRowFour);  
canvas.clipRect(2 * mRectInset, 2 * mRectInset,  
    mClipRectRight-2*mRectInset,  
    mClipRectBottom-2*mRectInset);  
drawClippedRectangle(canvas);  
canvas.restore();
```



Finally, draw and transform text.

In the previous steps you used the translate transform to move the origin of the canvas. You can apply transformations to any shape, including text, before you draw it, as shown in the following example.

```
// Draw text with a translate transformation applied.  
canvas.save();  
mPaint.setColor(Color.CYAN);  
// Align the RIGHT side of the text with the origin.  
mPaint.setTextAlign(Paint.Align.LEFT);  
// Apply transformation to canvas.  
canvas.translate(mColumnnTwo, mTextRow);  
// Draw text.  
canvas.drawText(  
    getContext().getString(R.string.translated), 0, 0, mPaint);  
canvas.restore();  
  
// Draw text with a translate and skew transformations applied.  
canvas.save();  
mPaint.setTextSize(mTextSize);  
mPaint.setTextAlign(Paint.Align.RIGHT);  
// Position text.  
canvas.translate(mColumnnTwo, mTextRow);  
// Apply skew transformation.  
canvas.skew(0.2f, 0.3f);  
canvas.drawText(  
    getContext().getString(R.string.skewed), 0, 0, mPaint);  
canvas.restore();  
} // End of onDraw()
```

## Solution code

Android Studio project: [ClippingExample](#)

# Summary

- The `Context` of an activity maintains a state that preserves transformations and clipping regions for the `Canvas`.
- Use `canvas.save()` and `canvas.restore()` to draw and return to the original state of your canvas.
- To draw multiple shapes on a canvas, you can either calculate their location, or you can move (translate) the origin of your drawing surface. The latter can make it easier to create utility methods for repeated draw sequences.
- Clipping regions can be any shape, combination of shapes or path.
- You can add, subtract, and intersect clipping regions to get exactly the region you need.
- You can apply transformations to text.

## Related concepts

The related concept documentation is in [The Canvas class](#).

## Learn more

Android developer documentation:

- [Canvas class](#)
- [Bitmap class](#)
- [View class](#)
- [Paint class](#)
- [Bitmap.Config configurations](#)
- [Region.Op operators](#)
- [Path class](#)

# 11.2: Creating a SurfaceView object

## Contents:

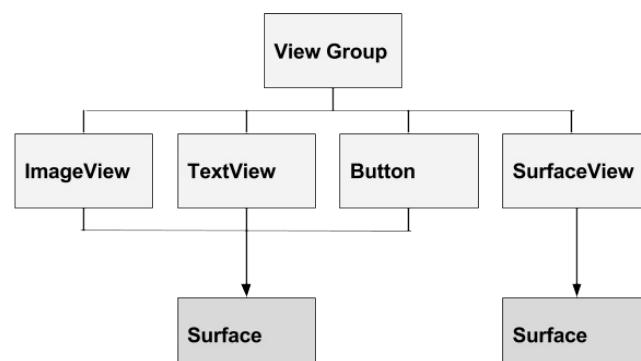
- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Create the SurfaceView example app
- Solution code
- Summary
- Related concept
- Learn more

When you create a custom view and override its `onDraw()` method, all drawing happens on the UI thread. Drawing on the UI thread puts an upper limit on how long or complex your drawing operations can be, because your app has to complete all its work for every screen refresh.

One option is to move some of the drawing work to a different thread using a `SurfaceView`.

- All the views in your view hierarchy are rendered onto one `Surface` in the UI thread.
- In the context of the Android framework, `Surface` refers to a lower-level drawing surface whose contents are eventually displayed on the user's screen.
- A `SurfaceView` is a view in your view hierarchy that has its own separate `Surface`, as shown in the diagram below. You can draw to it in a separate thread.
- To draw, start a thread, lock the `SurfaceView`'s canvas, do your drawing, and post it to the `Surface`.

The following diagram shows a View Hierarchy with a `Surface` for the views and another



separate `Surface` for the `SurfaceView`.

# What you should already KNOW

You should be able to:

- Create a custom `View`.
- Draw on and clip a `Canvas`.
- Add event handlers to views.
- Understand basic threading.

# What you will LEARN

You will learn how to:

- How to use a `SurfaceView` to draw to the screen from a different thread.
- A basic app architecture for simple games.

# What you will DO

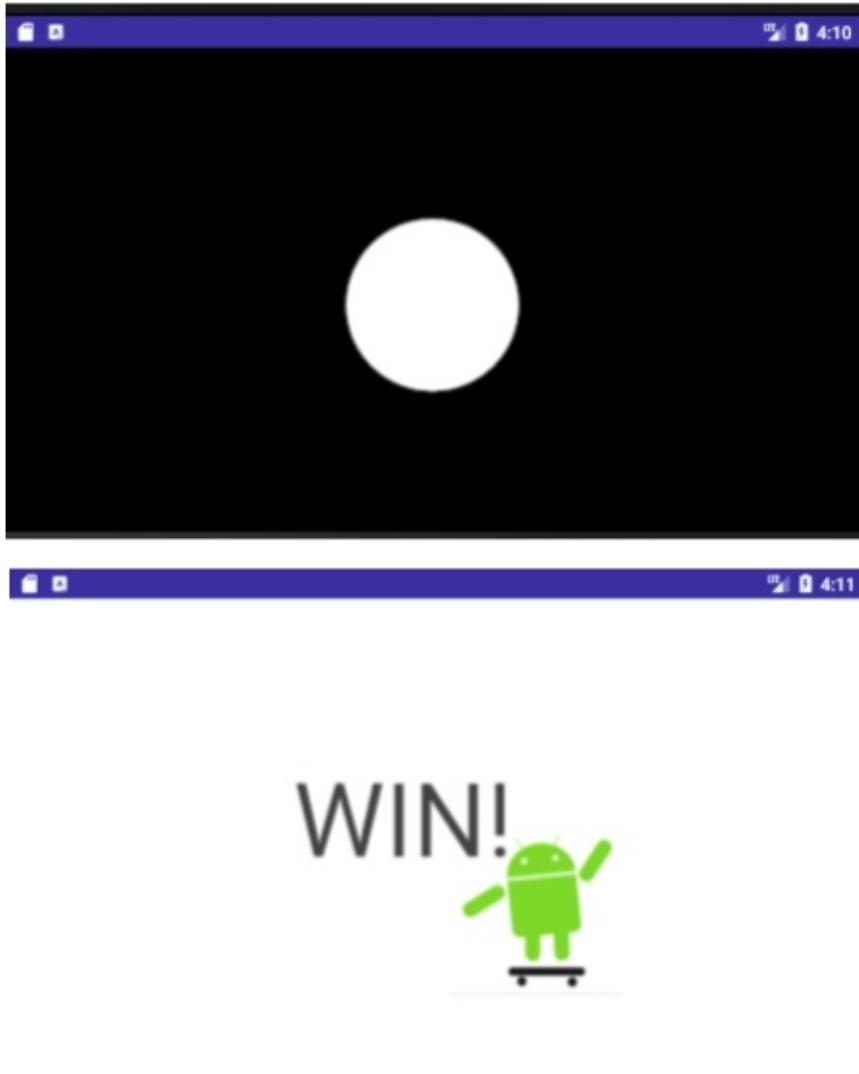
- Create an app that uses a `SurfaceView` to implement a simple game.

## App overview

The [SurfaceViewExample](#) app lets you search for an Android image on a dark phone screen using a "flashlight."

1. At app startup, the user sees a black screen with a white circle, the "flashlight."
2. While the user drags their finger, the white circle follows the touch.
3. When the white circle intersects with the hidden Android image, the screen lights up to reveal the complete image and a "win" message.
4. When the user lifts their finger and touches the screen again, the screen turns black and the Android image is hidden in a new random location.

The following is a screenshot of the SurfaceViewExample app at startup, and after the user has found the Android image by moving around the flashlight.



Additional features:

- Size of the flashlight is a ratio of the smallest screen dimension of the device.
- Flashlight is not centered under the finger, so that the user can see what's inside the circle.

## Task 1. Create the SurfaceViewExample app

You are going to build the SurfaceViewExample app from scratch. The app consists of the following three classes:

- **MainActivity**—Locks screen orientation, gets the display size, creates the `GameView`, and sets the `GameView` as its content view. Overrides `onPause()` and `onResume()` to pause and resume the game thread along with the `MainActivity`.

- **FlashlightCone**—Represents the cone of a flashlight with a radius that's proportional to the smaller screen dimension of the device. Has get methods for the location and size of the cone and a set method for the cone's location.
- **GameView**—A custom `SurfaceView` where game play takes place. Responds to motion events on the screen. Draws the game screen in a separate thread, with the flashlight cone at the current position of the user's finger. Shows the "win" message when winning conditions are met.

## 1.1 Create an app with an empty activity

1. Create an app using the **Empty Activity** template. Call the app `SurfaceViewExample`.
2. Uncheck **Generate Layout File**. You do not need a layout file.

## 1.2 Create the FlashlightCone class

1. Create a Java class called `FlashlightCone`.

```
public class FlashlightCone {}
```

2. Add member variables for x, y, and the radius.

```
private int mX;
private int mY;
private int mRadius;
```

3. Add methods to get values for x, y, and the radius. You do not need any methods to set them.

```
public int getX() {
    return mX;
}

public int getY() {
    return mY;
}

public int getRadius() {
    return mRadius;
}
```

4. Add a constructor with integer parameters `viewWidth` and `viewHeight`.
5. In the constructor, set `mX` and `mY` to position the circle at the center of the screen.
6. Calculate the radius for the flashlight circle to be one third of the smaller screen dimension.

```

public FlashlightCone(int viewWidth, int viewHeight) {
    mX = viewWidth / 2;
    mY = viewHeight / 2;
    // Adjust the radius for the narrowest view dimension.
    mRadius = ((viewWidth <= viewHeight) ? mX / 3 : mY / 3);
}

```

7. Add a `public void update()` method. The method takes integer parameters `newX` and `newY`, and it sets `mX` to `newX` and `mY` to `newY`.

```

public void update(int newX, int newY) {
    mX = newX;
    mY = newY;
}

```

## 1.3 Create a new SurfaceView class

1. Create a new Java class and call it `GameView`.
2. Let it extend `SurfaceView` and implement `Runnable`. `Runnable` adds a `run()` method to your class to run its operations on a separate thread.

```
public class GameView extends SurfaceView implements Runnable {}
```

3. Implement methods to add a stub for the only required method, `run()`.

```

@Override
public void run(){}

```

4. Add the stubs for the constructors and have each constructor call `init()`.

```

public GameView(Context context) {
    super(context);
    init(context);
}

public GameView(Context context, AttributeSet attrs) {
    super(context, attrs);
    init(context);
}

public GameView(Context context, AttributeSet attrs, int defStyleAttr) {
    super(context, attrs, defStyleAttr);
    init(context);
}

```

5. Add private `init()` method and set the `mContext` member variable to `context`.

```
private void init(Context context) {
    mContext = context;
}
```

6. In the `GameView` class, add stubs for the `pause()` and `resume()` methods. Later, you will manage your thread from these two methods.

## 1.4 Finish the MainActivity

1. In `MainActivity`, create a member variable for the `GameView` class.

```
private GameView mGameView;
```

In the `onCreate()` method:

2. Lock the screen orientation into landscape. Games often lock the screen orientation.

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
```

3. Create an instance of `GameView`.
4. Set `mGameview` to completely fill the screen.
5. Set `mGameView` as the content view for `MainActivity`.

```
mGameView = new GameView(this);
// Android 4.1 and higher simple way to request fullscreen.
mGameView.setSystemUiVisibility(View.SYSTEM_UI_FLAG_FULLSCREEN);
setContentView(mGameView);
```

6. Still in `MainActivity`, override the `onPause()` method to also pause the `mGameView` object. This `onPause()` method shows an error, because you have not implemented the `pause()` method in the `GameView` class.

```
@Override
protected void onPause() {
    super.onPause();
    mGameView.pause();
}
```

7. Override `onResume()` to resume the `mGameView`. The `onResume()` method shows an error, because you have not implemented the `resume()` method in the `GameView`.

```

@Override
protected void onResume() {
    super.onResume();
    mGameView.resume();
}

```

## 1.5 Finish the init() method for the GameView class

In the constructor for the `GameView` class:

1. Assign the `context` to `mContext`.
2. Get a persistent reference to the `SurfaceHolder`. Surfaces are created and destroyed by the system while the holder persists.
3. Create a `Paint` object and initialize it.
4. Create a `Path` to hold drawing instructions. If prompted, import



`android.graphics.Path`.

Here is the code for the `init()` method.

```

private void init(Context context) {
    mContext = context;
    mSurfaceHolder = getHolder();
    mPaint = new Paint();
    mPaint.setColor(Color.DKGRAY);
    mPath = new Path();
}

```

5. After copy/pasting the code, define the missing member variables.

## 1.6 Add the setUpBitmap() method to the GameView class

The `setUpBitmap()` method calculates a random location on the screen for the Android image that the user has to find. You also need a way to calculate whether the user has found the bitmap.

1. Set `mBitmapX` and `mBitmapY` to random x and y positions that fall inside the screen.
2. Define a rectangular bounding box that contains the Android image.
3. Define the missing member variables.

```

private void setUpBitmap() {
    mBitmapX = (int) Math.floor(
        Math.random() * (mViewWidth - mBitmap.getWidth()));
    mBitmapY = (int) Math.floor(
        Math.random() * (mViewHeight - mBitmap.getHeight()));
    mWinnerRect = new RectF(mBitmapX, mBitmapY,
        mBitmapX + mBitmap.getWidth(),
        mBitmapY + mBitmap.getHeight());
}

```

## 1.7 Implement the methods to pause and resume the GameView class

The `pause()` and `resume()` methods on the `GameView` are called from the `MainActivity` when it is paused or resumed. When the `MainActivity` pauses, you need to stop the `GameView` thread. When the `MainActivity` resumes, you need to create a new `GameView` thread.

1. Add the `pause()` and `resume()` methods using the code below. The `mRunning` member variable tracks the thread status, so that you do not try to draw when the activity is not running anymore.

```

public void pause() {
    mRunning = false;
    try {
        // Stop the thread (rejoin the main thread)
        mGameThread.join();
    } catch (InterruptedException e) {
    }
}

public void resume() {
    mRunning = true;
    mGameThread = new Thread(this);
    mGameThread.start();
}

```

2. As before, add the missing member variables.

Thread management can become a lot more complex after you have multiple threads in your game. See [Sending Operations to Multiple Threads](#) for lessons in thread management.

## 1.8 Implement the `onSizeChanged()` method

There are several ways in which to set up the view after the system has fully initialized the view. The `onSizeChangedMethod()` is called every time the view changes. The view starts out with 0 dimensions. When the view is first inflated, its size changes and `onSizeChangedMethod()` is called. Unlike in `onCreate()`, the view's correct dimensions are available.

1. Get the image of Android on a skateboard from [github](#) and add it to your drawable folder, or use a small image of your own choice.
2. In `GameView`, override the `onSizeChanged()` method. Both the new and the old view dimensions are passed as parameters as shown below.

```
@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    super.onSizeChanged(w, h, oldw, oldh);
}
```

Inside the `onSizeChanged()` method:

3. Store the width and height in member variables `mViewWidth` and `mViewHeight`.

```
mViewWidth = w;
mViewHeight = h;
```

4. Create a `FlashlightCone` and pass in `mViewWidth` and `mViewHeight`.

```
mFlashlightCone = new FlashlightCone(mViewWidth, mViewHeight);
```

5. Set the font size proportional to the view height.

```
mPaint.setTextSize(mViewHeight / 5);
```

6. Create a `Bitmap` and call `setupBitmap()`.

```
mBitmap = BitmapFactory.decodeResource(
    mContext.getResources(), R.drawable.android);
setUpBitmap();
```

## 1.9 Implement the `run()` method in the `GameView` class

The interesting stuff, such as drawing and screen refresh synchronization, happens in the `run()` method. Inside your `run()` method stub, do the following:

1. Declare a `Canvas` `canvas` variable at the top of the `run()` method:

```
Canvas canvas;
```

2. Create a loop that only runs while `mRunning` is true. All the following code must be inside that loop.

```
while (mRunning) {  
}
```

3. Check whether there is a valid `surface` available for drawing. If not, do nothing.

```
if (mSurfaceHolder.getSurface().isValid()) {
```

All code that follows must be inside this `if` statement.

4. Because you will use the flashlight cone coordinates and radius multiple times, create local helper variables inside the `if` statement.

```
int x = mFlashlightCone.getX();  
int y = mFlashlightCone.getY();  
int radius = mFlashlightCone.getRadius();
```

5. Lock the canvas.

In an app, with more threads, you must enclose this with a `try/catch` block to make sure only one thread is trying to write to the `Surface`.

```
canvas = mSurfaceHolder.lockCanvas();
```

1. Save the current canvas state.

```
canvas.save();
```

2. Fill the canvas with white color.

```
canvas.drawColor(Color.WHITE);
```

3. Draw the Skateboarding Android bitmap on the canvas.

```
canvas.drawBitmap(mBitmap, mBitmapX, mBitmapY, mPaint);
```

4. Add a circle that is the size of the flashlight cone to `mPath`.

```
mPath.addCircle(x, y, radius, Path.Direction.CCW);
```

- Set the circle as the clipping path using the `DIFFERENCE` operator, so that's what's inside the circle is clipped (not drawn).

```
canvas.clipPath(mPath, Region.Op.DIFFERENCE);
```

- Fill everything outside of the circle with black.

```
canvas.drawColor(Color.BLACK);
```

- Check whether the center of the flashlight circle is inside the winning rectangle. If so, color the canvas white, redraw the Android image, and draw the winning message.

```
if (x > mWinnerRect.left && x < mWinnerRect.right  
    && y > mWinnerRect.top && y < mWinnerRect.bottom) {  
    canvas.drawColor(Color.WHITE);  
    canvas.drawBitmap(mBitmap, mBitmapX, mBitmapY, mPaint);  
    canvas.drawText(  
        "WIN!", mViewWidth / 3, mViewHeight / 2, mPaint);  
}
```

- Drawing is finished, so you need to rewind the path, restore the canvas, and release the lock on the canvas.

```
mPath.rewind();  
canvas.restore();  
mSurfaceHolder.unlockCanvasAndPost(canvas);
```

Run your app. It should display a black screen with a white circle at the center of the screen.

## 1.10 Respond to motion events

For the game to work, your app needs to detect and respond to the user's motions on the screen.

- In `GameView`, override the `onTouchEvent()` method and update the flashlight position on the `ACTION_DOWN` and `ACTION_MOVE` events.

```

@Override
public boolean onTouchEvent(MotionEvent event) {
    float x = event.getX();
    float y = event.getY();

    // Invalidate() is inside the case statements because there are
    // many other motion events, and we don't want to invalidate
    // the view for those.
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            setUpBitmap();
            updateFrame((int) x, (int) y);
            invalidate();
            break;
        case MotionEvent.ACTION_MOVE:
            updateFrame((int) x, (int) y);
            invalidate();
            break;
        default:
            // Do nothing.
    }
    return true;
}

```

2. Implement the `updateFrame()` method called in `onTouchEvent()` to set the new coordinates of the `FlashlightCone`.

```

private void updateFrame(int newX, int newY) {
    mFlashlightCone.update(newX, newY);
}

```

3. Run your app and GAME ON!
4. After you win, tap the screen to play again.

## Solution code

Android Studio project: [SurfaceViewExample](#)

## Summary

- To offload drawing to a different thread, create a custom view that extends `SurfaceView` and implements `Runnable`. The `SurfaceView` is part of your view hierarchy but has a drawing `Surface` that is separate from the rest of the view hierarchy.
- Create an instance of your custom view and set it as the content view of your activity.

- Add `pause()` and `resume()` methods to the `SurfaceView` that stop and start a thread.
- Override `onPause()` and `onResume()` in the activity to call the `pause()` and `resume()` methods of the `SurfaceView`.
- If appropriate, handle touch events, for example, by overriding `onTouchEvent()`.
- Add code to update your data.
- In the `SurfaceView`, implement the `run()` method to:
  - Check whether a `surface` is available.
  - Lock the canvas.
  - Draw.
  - Unlock the canvas and post to the `surface`.

## Related concept

The related concept documentation is in [The SurfaceView class](#).

## Learn more

Android developer documentation:

- [SurfaceView class](#)
- [SurfaceHolder interface](#)
- [Runnable interface](#)
- [Sending Operations to Multiple Threads](#)
- [Paint class](#)
- [Path class](#)
- [ClippingBasic code sample](#)
- [Graphics architecture](#) for a comprehensive introduction

# 12.1: Creating property animations

## Contents:

- What you should already KNOW
- What you will LEARN
- What you will DO
- App overview
- Task 1. Creating the PropertyAnimation app
- Solution code
- Coding challenge
- Summary
- Related concept
- Learn more

The [Property Animation](#) system allows you to animate almost any property of an object. You can define an animation to change any object property (a field in an object) over time, regardless of whether the change draws to the screen or not. A property animation changes a property's value over a specified length of time. For example, you can animate a circle to grow bigger by increasing its radius.

With the property animation system, you assign animators to the properties that you want to animate, such as color, position, or size. You also define aspects of the animation such as interpolation. For example, you would create an animator for the radius of a circle whose size you want to change.

The property animation system lets you define the following characteristics of an animation:

- *Duration*: You can specify the duration of an animation. The default length is 300 milliseconds.
- *Time interpolation*: You can specify how the values for the property are calculated as a function of the animation's current elapsed time. You can choose from provided interpolators or create your own.
- *Repeat count and behavior*: You can specify whether or not to have an animation repeat when it reaches the end of a duration, and how many times to repeat the animation. You can also specify whether you want the animation to play back in reverse. Setting it to reverse plays the animation forwards then backwards repeatedly, until the number of repeats is reached.
- *Animator sets*: You can group animations into logical sets that play together or sequentially or after specified delays.

- *Frame-refresh delay:* You can specify how often to refresh frames of your animation. The default is set to refresh every 10 ms, but the speed in which your application can refresh frames ultimately depends on how busy the system is overall and how fast the system can service the underlying timer.

## What you should already KNOW

You should be able to:

- Create and run apps in Android Studio.
- Create a custom `View` object.
- Draw to the screen using a `Canvas` object.

## What you will LEARN

You will learn how to:

- Create a custom `View` that includes a `radius` property.
- Create simple radius-based animations using `ObjectAnimator` objects.
- Combine and sequence animations using an `AnimatorSet` object.

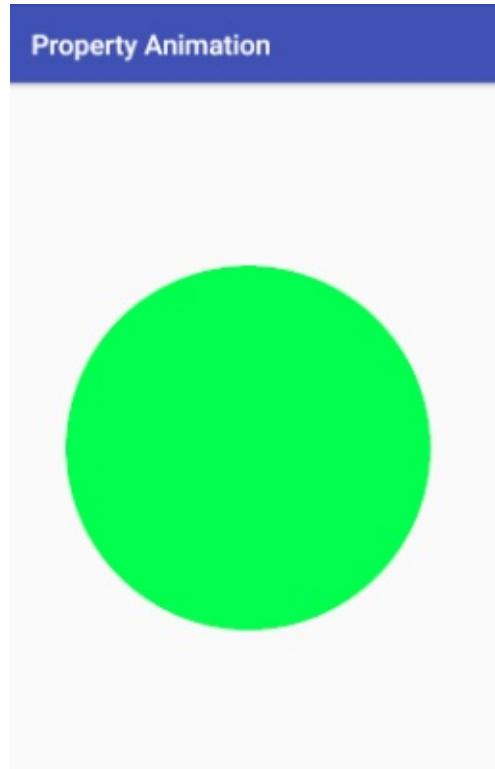
## What you will DO

- Create an app with a single activity and a `PulseAnimationView` custom view.
- Add an `mRadius` property to the `PulseAnimationView` class and a `setRadius()` "setter" method for that custom property. The `Animator` instance will call the `setRadius()` method to change the size of `mRadius` during the animation.
- Override the `onDraw()` method to draw a circle of size `mRadius`. The circle is created at the place where the user taps.
- Create three `Animator` instances for the radius property.
- Use an `AnimatorSet` to play the animations in sequence after the user taps the screen.

## App overview

The PropertyAnimation app opens with a white screen. When the user taps the screen, an animation plays. The animation draws an expanding circle, then pauses. Then the animation draws a shrinking circle that changes color. Finally, the animation draws the same circle again, reversing without pausing. The screenshot below shows a snapshot of the

The PropertyAnimation app opens with a white screen. When the user taps the screen, an animation plays. The animation draws an expanding circle, then pauses. Then the animation draws a shrinking circle that changes color. Finally, the animation draws the same circle again, reversing without pausing. The screenshot below shows a snapshot of the



PropertyAnimation app during animation.

## Task 1. Creating the PropertyAnimation app

Note that in this advanced practical, you are expected to create member variables, import classes, and extract values as needed.

### 1.1 Create an app with one activity

Create an app that uses the **Empty Activity template**. Make sure that **Backwards Compatibility** and **Generate Layout File** are enabled.

### 1.2 Create the custom view to animate

1. Create a new custom view class called `PulseAnimationView` that extends `View`.

```
public class PulseAnimationView extends View {}
```

```

public PulseAnimationView(Context context) {this(context, null);}

public PulseAnimationView(Context context, AttributeSet attrs) {
    super(context, attrs);
}

```

3. In `activity_main.xml`, remove the `TextView` and add a `PulseAnimationView` that matches the size of the parent.

```

<com.example.android.propertyanimation.PulseAnimationView
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>

```

4. Run your app. It shows a white screen and the name of the app.

## 1.3 Implement the method to set the radius

This app uses a property animator, which changes a property's value over a specified length of time. You will change the `radius` property of the `PulseAnimationView` to animate a circle by changing its size.

The property animator needs to be able to change the property that will be animated. It does this through a "setter" method for the property. In order for the animator to find and use the setter, the following conditions need to be met:

- If the class whose property is being animated does not provide a setter property, you have to create one. For the `PulseAnimationView`, you will create a member variable for the `radius` and a `setRadius()` method to set the variable's value.
- The property setter's name needs to be of the form `set` *PropertyName* `()`. The *PropertyName* can be any valid string. When you create the animator, you will pass the *PropertyName* as string to the constructor.
- To cause the object to be redrawn after the property changes, you must call the `invalidate()` method. Calling `invalidate()` tells the system that something about the view has changed, and the system measures, lays out, and redraw the views.
- Create a member variable for the radius in `PulseAnimationView`.

```

private float mRadius;

```

- Create a `public void setRadius()` method that takes a `float radius` as its argument. Set `mRadius` to the passed-in radius and call the `invalidate()` method.

- Create a `public void setRadius()` method that takes a `float radius` as its argument. Set `mRadius` to the passed-in radius and call the `invalidate()` method.

```
public void setRadius(float radius) {  
    mRadius = radius;  
    invalidate();  
}
```

To make the animation more interesting, use the radius to affect other aspects of the animation. For example, you could play a sound as the circle grows, and the sound could change as a function of the radius. For the PropertyAnimation app, in addition to changing the size, you are going to change the color of the circle as a function of the radius.

- Create and initialize a `mPaint` member variable.

```
private final Paint mPaint = new Paint();
```

- Create a constant `COLOR_ADJUSTER` and set it to 5. You will use this constant in the next step.

```
private static final int COLOR_ADJUSTER = 5;
```

- Inside the `setRadius()` method, after setting `mRadius` and before calling the `invalidate()` method, change the color of the `mPaint` variable. Colors are integers and can thus be used in integer operations. You can change the value of the `COLOR_ADJUSTER` constant to see how it affects the color of the circle. You can also use a more sophisticated color function, if you want to.

```
mPaint.setColor(Color.GREEN + (int) radius / COLOR_ADJUSTER);
```

## 1.4 Add code to respond to touch events

In the PropertyAnimation app, animation is initiated by a user touch, and the animation originates at the location of the touch event.

1. In the custom view class, create `private float` member variables `mX` and `mY` to store the event coordinates.

```
private float mX;  
private float mY;
```

2. Override the `onTouchEvent()` method to get the event coordinates and store them in the

```

@Override
public boolean onTouchEvent(MotionEvent event) {
    if (event.getActionMasked() == MotionEvent.ACTION_DOWN) {
        mX = event.getX();
        mY = event.getY();
    }
    return super.onTouchEvent(event);
}

```

## 1.5 Add the animation code

The animation is performed by an `Animator` object that, once started, changes the value of a property from a starting value towards an end value over a given duration.

1. Create class constants for the animation duration and for a delay before the animation starts. You can change the values of these constants later and explore how that affects the appearance of the animation. The time is in milliseconds.

```

private static final int ANIMATION_DURATION = 4000;
private static final long ANIMATION_DELAY = 1000;

```

2. Override the `onSizeChanged()` method.

```

@Override
public void onSizeChanged(int w, int h, int oldw, int oldh) {}

```

Inside the `onSizeChanged()` method, you will create three `ObjectAnimator` objects and one `AnimatorSet` object. You cannot create them in the `onCreate()` method, because the views have not been inflated, and so the call to `getWidth()` used below would not return a valid value.

3. In the `onSizeChanged()` method, create an `ObjectAnimator` called `growAnimator`. You need to pass in a reference to the object that is being animated (`this`), the name of the property that is to be animated (`"radius"`), the starting value (`0`), and the ending value (`getWidth()`). In this case, the ending value is the width of the

`PulseAnimationView`.

```

ObjectAnimator growAnimator = ObjectAnimator.ofFloat(this,
    "radius", 0, getWidth());

```

4. Set the duration of the animation in milliseconds.

```

growAnimator.setDuration(ANIMATION_DURATION);

```

5. Choose an interpolator for the animation. The interpolator affects the rate of change; that is, the interpolator affects how the animated property changes from its starting value to its ending value.
6. With a `LinearInterpolator`, the rate of change is constant; that is, the value changes by the same amount for every animation step. For example, if the starting `radius` value were 0 and your ending value were 10, every step of the animation might increase the radius by 2. For example,  $0 > 2 > 4 > 6 > 8 > 10$ .
7. With an `AccelerateDecelerateInterpolator`, the rate of change starts and ends slowly but accelerates through the middle. The animation starts with increasingly larger steps and then ends with decreasingly smaller steps. For example  $0 > 2 > 5 > 10 > 20 > 30 > 35 > 38 > 40$ .
8. You can create custom interpolators, too. See the `BaseInterpolator` class for a list of many of the available interpolators, and try some of them with this app.

```
growAnimator.setInterpolator(new LinearInterpolator());
```

9. Create a second `ObjectAnimator` object named `shrinkAnimator`. Use the same parameters as for `growAnimator` but swap the start and end values.

```
ObjectAnimator shrinkAnimator = ObjectAnimator.ofFloat(this,
    "radius", getWidth(), 0);
```

10. Set the duration to `ANIMATION_DURATION` and use a `LinearOutSlowInInterpolator`. This is a more complex interpolator and you can check the documentation for details.

```
shrinkAnimator.setDuration(ANIMATION_DURATION);
shrinkAnimator.setInterpolator(new LinearOutSlowInInterpolator());
```

11. Add a starting delay. When the animation is started, it will wait for the specified delay before it runs.

```
shrinkAnimator.setStartDelay(ANIMATION_DELAY);
```

12. Still in `onSizeChanged()`, create a third `ObjectAnimator` instance called `repeatAnimator`.

```
ObjectAnimator repeatAnimator = ObjectAnimator.ofFloat(this,
    "radius", 0, getWidth());
repeatAnimator.setStartDelay(ANIMATION_DELAY);
repeatAnimator.setDuration(ANIMATION_DURATION);
```

13. Add a repeat count to `repeatAnimator`. A repeat count of `0` is the default. With a

repeat count of `0`, the animation plays once and does not repeat. With a repeat count of `1`, the animation plays twice.

```
repeatAnimator.setRepeatCount(1);
```

14. Set the repeat mode to `REVERSE`. In this mode, every time the animation plays, it reverses the beginning and end values. (The other possible value, which is the default, is `RESTART`.)

```
repeatAnimator.setRepeatMode(ValueAnimator.REVERSE);
```

15. Create a `private AnimatorSet` member variable called `mPulseAnimatorSet` and initialize the variable with an `AnimatorSet`.

```
private AnimatorSet mPulseAnimatorSet = new AnimatorSet();
```

An `AnimatorSet` allows you to combine several animations and to control in what order they are played. You can have several animations play at the same time or in a specified sequence. `AnimatorSet` objects can contain other `AnimatorSet` objects. See the [Property Animation](#) guide for all the cool things you can do with animator sets.

16. The following `AnimatorSet` is very simple and specifies that the `growAnimator` should play before the `shrinkAnimator`, followed by the `repeatAnimator`. Add it after you have created the animators.

```
mPulseAnimatorSet.play(growAnimator).before(shrinkAnimator);
mPulseAnimatorSet.play(repeatAnimator).after(shrinkAnimator);
```

17. If you run your app now, you still only see the white screen.

## 1.6 Add code to draw the circle

The `Animator` that you implemented does not draw anything. The actual drawing of the circle must be done in the `onDraw()` method, which is executed after the view has been invalidated, which happens in the `setRadius()` method.

1. Override the `onDraw()` method to draw a circle at the `mX`, `mY` coordinates.
2. Give the circle a radius of `mRadius` and set the color to `mPaint`.

```
@Override  
protected void onDraw(Canvas canvas) {  
    super.onDraw(canvas);  
    canvas.drawCircle(mX, mY, mRadius, mPaint);  
}
```

## 1.7 Play the animation

1. In the `onTouchEvent()` method, add code to play the animation if there is an `ACTION_DOWN` event. If an animation is running, cancel it. This resets the `mPulseAnimatorSet` and its animations to the starting values.

```
if(mPulseAnimatorSet != null && mPulseAnimatorSet.isRunning()) {  
    mPulseAnimatorSet.cancel();  
}  
mPulseAnimatorSet.start();
```

2. Run your app and tap the white screen to see the animations play. Have some fun and experiment with the animation!

This example app creates and runs animations from the Java code because it uses data that is not available until the view has been drawn. You can also define animators and animator sets in XML. See the [Property Animation](#) guide and the [Animations](#) concept for instructions and code examples.

## Solution code

Android Studio project: [PropertyAnimation](#).

## Coding challenge

Write an app that demonstrates the use of the [physics-based animation](#) from the support library.

- You need to add the library dependency to your `build.gradle` file. Use the latest version of the physics-based library as listed in the [official documentation](#).

```
dependencies {  
    ...  
    compile "com.android.support:support-dynamic-animation:26.0.0"  
}
```

- Here is a code snippet for a simple vertical spring animation. See the [Spring Animation](#) documentation and `SpringAnimation` class for more information.

```
final SpringAnimation anim = new SpringAnimation(
    this, DynamicAnimation.Y, 10)
    .setStartVelocity(10000);
anim.getSpring().setStiffness(STIFFNESS_LOW);
anim.start();
```

- Here is a code snippet for a simple rotation fling animation. See the [Fling Animation](#) documentation and `FlingAnimation` class for more information.

```
FlingAnimation fling = new FlingAnimation(this, DynamicAnimation.ROTATION_X);
fling.setStartVelocity(150)
    .setMinValue(0)
    .setMaxValue(1000)
    .setFriction(0.1f)
    .start();
```

- The [PhysicsAnimation](#) app shows a possible solution.

## Summary

- With property animation, you can use almost any property of an object to create an animation.
- One way to create a property animation is to:
  - Create a view or custom view with the property.
  - If the view does not have a setter for the property, create one and name it `setPropertyName`. The setter is called by the `Animator` object to change the property value during animation. You must call `invalidate()` in the setter.
  - Override `onDraw()` to perform any drawing.
  - Decide how the animation is to be triggered. For example, the animation could be triggered when the user taps the screen.
  - In the method that responds to the trigger event, for example, in the `onTouchEvent()` method, create the objects.
  - To create an `Animator`, set the object and property to be animated. For the property, set a start value, an end value, and a duration.
  - Use interpolators to specify how the animated property changes over time. You can use one of the many supplied interpolators or create your own.
  - You can combine several animations to run in sequence or at the same time using

`AnimatorSet` objects.

- See the [PropertyAnimation](#) guide for a complete description of all the cool things you can do with property animations.

## Related concept

The related concept documentation is in [Animations](#).

## Learn more

Android developer documentation:

- [View Animation](#)
- [Property Animation](#)
- [Drawable Animation](#)
- [Physics based animation](#)
- See the [Graphics Architecture](#) series of articles for an in-depth explanation of how the Android framework draws to the screen.

# Appendix: Setup

## Contents:

- [What you should already KNOW](#)
- [What you will LEARN](#)
- [What you will DO](#)
- [App overview](#)
- [Task 1. Set up your environment](#)
- [Learn more](#)

The practicals in this book assume that you are using the latest version of Android Studio, and some of the practicals require at least Android Studio 3.0. For example, the Performance chapters teach you how to use the [Android Profiler](#) tools.

This page summarizes how to set up the latest version of Android Studio.

If you need more help, see [Install Android Studio and Run Hello World](#) in the Android Developer Fundamentals course and the resources at the end of this page.

## What you should already KNOW

You should be able to:

- Create and run an Android app using Android Studio.

## What you will LEARN

- How to set up your environment for the Advanced Android Development practicals.

## What you will DO

- Install Android Studio 3 or later.
- Update Android Studio, if necessary.
- Update the SDK for building and running the apps you create in the Advanced Android Development course.
- If necessary, set up your mobile device to run and debug Android apps from your computer.

# App overview

You can use the finished [WordListSQLInteractive](#) app from the Android Developer Fundamentals course to test your setup, or you can use any other app that's not simplistic.

## Task 1. Set up your environment

### 1.1 Get Android Studio 3 or later

What follows are the summary instructions for installing Android Studio 3, for each platform that supports Android Studio.

Note that you can keep two independent versions of Android Studio on your development machine, if you want to.

#### Windows:

1. [Download Android Studio](#) for Windows.
2. Unpack the zip file.
3. Rename the resulting folder to something unique like "Android Studio 3."
4. Move the folder to a permanent location, such as next to your existing Android Studio install in `C:\Program Files\Android\`.
5. Inside `C:\Program Files\Android\Android Studio 3\bin\`, launch `studio64.exe` (or if you're on a 32-bit machine, launch `studio.exe`).
6. To make the version available in your Start menu, right-click `studio64.exe` and click **Pin to Start Menu**.

#### Mac:

1. [Download Android Studio](#) for Mac.
2. Unpack the zip file.

Note: If you download version 2.3 or lower, the app name does not include the version number. Rename the new version before moving it into your apps directory. Otherwise, you might override your existing version of Android Studio.

3. Drag the app file into your Applications folder.
4. Launch the app.

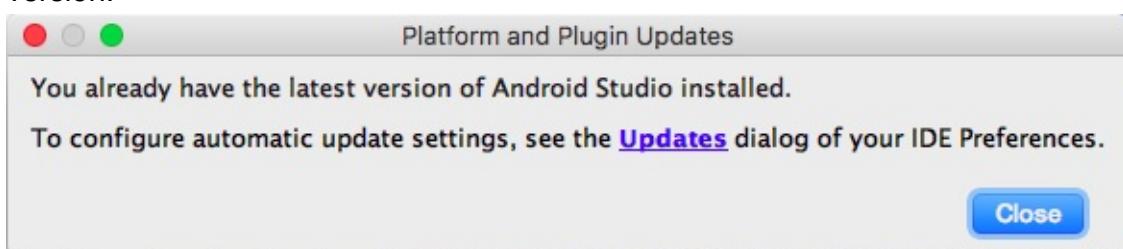
#### Linux:

1. [Download Android Studio](#) for Linux.
2. Unpack the zip file.

3. Rename the resulting folder to something unique like "android-studio-3".
4. Move the folder to wherever you have your stable version, such as within `/usr/local/` for your user profile, or within `/opt/` for shared users.
5. Open a terminal, navigate into `android-studio-3/bin/`, and execute `studio.sh`.
6. To make the new version available in your list of apps, select **Tools > Create Desktop Entry** from the Android Studio menu bar.

## 1.2 Update Android Studio

1. Open Android Studio.
2. If you are not prompted to update, select **Android Studio > Check for updates....**
3. Continue to check for updates, until you see a dialog saying that you have the latest version.



## 1.3 Build an app to verify your Android Studio installation

You probably need to install additional SDK downloads before your app builds.

1. Open Android Studio, then open an existing app of some complexity, such as [WordListSQLInteractive](#).

When you build an app that you built with a previous version of Android Studio, you may get errors about components and libraries that are missing.

2. Click the links as prompted by the error messages, and install the needed components.
3. Update Gradle, if you're prompted to do so.
4. Follow the prompts until your app finishes building.

## 1.4 Run the app on a mobile device

1. On your mobile device, [enable developer options](#), if they are not already enabled. To find these settings on the device, open **Settings > Developer options**.

On Android 4.2 and higher, the **Developer options** screen is hidden by default. To make the screen visible, go to **Settings > About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options** at the bottom.

2. In **Developer options**, enable **USB Debugging**. This is necessary for your development machine to see and access your mobile device.
3. Connect the mobile device to your development computer with a USB data cable.
4. In Android Studio, click **Run**.
5. You may be prompted to install HAXM and emulator images. If you have been using several emulator images, this installation can take a while. Your largest update may be over 1 GB and will take some time to download even on a fast connection. You can postpone the system image downloads and run them later from the AVD manager.
6. After installation, choose your device and run the app.

## 1.5 Run the app on an emulator

1. If the app is running on your device, stop it.
2. Run the app again, choosing an emulator. Your emulator may update before it runs your app.

If you don't have an emulator, click **Create Virtual Device** in the **Select Deployment Target** dialog. Choose a phone and an existing system image, if you have one, because additional system images are large to download.

3. Make sure that the app runs correctly.

## 1.6 Create and run Hello World

To make sure that you're ready to work, create and run a Hello World app:

1. Create a new project using the Basic Activity.
2. Accept all the defaults.
3. Run the app on any device or emulator.

## Learn more

- [Android Studio Release Notes](#)
- [Android Plugin for Gradle Release Notes](#)
- [Install Android Studio and Run Hello World](#) practical from [Android Developer Fundamentals](#)
- [Run Apps on a Hardware Device](#)
- [Android Studio Preview documentation](#), for information on how to get and install preview versions of Android Studio



# Appendix: Homework

## Contents:

- 1.1 Creating a Fragment with a UI
- 1.2: Communicating with a Fragment
- 2.1: Building app widgets
- 3.1: Working with sensor data
- 3.2: Working with sensor-based orientation
- 4.0 Optimizing for performance
- 4.1 Analyzing rendering and layout
- 4.2 Using the Memory Profiler
- 4.3 Optimizing network, battery, and image use
- 5.1 Using resources for languages
- 5.2 Using the locale
- 6.1: Exploring accessibility in Android
- 6.2: Creating accessible apps
- 7.1: Using the device location
- 8.1: Using the Places API
- 9.1 Adding a Google Map to your app
- 10.1 Creating a custom view
- 11.1 Applying clipping to a Canvas object
- 11.2 Creating a SurfaceView object
- 12.1 Creating animations

This appendix lists possible homework assignments that students can complete at the end of each practical. It is the instructor's responsibility to do the following:

- Assign homework, if homework is required.
- Communicate to students how to submit homework assignments.
- Grade homework assignments.

Instructors can use these suggested assignments as little or as much as they want. Instructors should feel free to assign any other homework they feel is appropriate.

## 1.1: Creating a Fragment with a UI

### Build and run an app

Build an app that uses the same fragment (`SimpleFragment`) with more than one activity.

1. In Android Studio, open the `FragmentExample2` app.
2. Add another Empty Activity called `SecondActivity`. For `SecondActivity`, use the same layout as `MainActivity`.
3. In the `SecondActivity` layout, replace the `article1` string resource with `article2`. Replace the `title1` string resource with `title2`. Both string resources are provided in the starter app and included in `FragmentExample2`.

Add the following functionality to the app, using `SimpleFragment` with `SecondActivity`:

1. Add a **Next** button under the **Open** button to navigate from the first activity to the second activity.
2. Add a **Previous** button in the second activity to navigate back to the first activity.

## Answer these questions

### Question 1

Which subclass of `Fragment` displays a vertical list of items that are managed by an adapter?

- `RowsFragment()`
- `PreferenceFragment()`
- `DialogFragment()`
- `ListFragment()`

### Question 2

Which of the following is the best sequence for adding a fragment to an activity that is already running?

- Declare the fragment inside the activity's layout file using a `<fragment>` view.
- Declare a location for the fragment inside the activity's layout file using the `<FrameLayout>` view group.
- Declare the location for the fragment inside the activity's layout file using the `<FrameLayout>` view group, get an instance of the fragment and `FragmentManager`, and use the `add()` transaction.
- Declare the location for the fragment inside the activity's layout file using the `<FrameLayout>` view group. Then get an instance of the fragment and `FragmentManager`, begin a transaction, use the `add()` transaction, and commit the transaction.

## Question 3

Which statement gets a reference to a fragment using the fragment's layout resource?

- `fragment = new SimpleFragment();`
- `SimpleFragment fragment = (SimpleFragment) fragmentManager.findViewById(R.id.fragment_container);`
- `SimpleFragment fragment = (SimpleFragment) fragmentManager.findFragmentById(R.id.fragment_container);`
- `FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();`

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

- The app displays a second activity when the user taps the **Next** button.
- The app's second activity includes an **Open** button. The **Open** button opens the same fragment (`SimpleFragment`) that appears when the user taps **Open** in the first activity.
- The fragment looks the same, with the **X** button.

## 1.2: Communicating with a Fragment

### Build and run an app

In the [FragmentCommunicate](#) app, when the user makes a choice in the fragment, the app shows the user's choice in a Toast message. The Toast shows "Choice is 0" for "Yes" or "Choice is 1" for "No."

Change the FragmentCommunicate app so that the Toast shows the "Yes" or "No" message rather than "0" or "1."

### Answer these questions

## Question 1

Which fragment-lifecycle callback draws the fragment's UI for the first time?

- `onAttach()`
- `onActivityCreated()`

- `onCreate()`
- `onCreateView()`

## Question 2

How do you send data from an activity to a fragment?

- Set a `Bundle` and use the `Fragment . setArguments(Bundle)` method to supply the construction arguments for the fragment.
- Use the Fragment method `getArguments()` to get the arguments.
- Define an interface in the `Fragment` class, and implement the interface in the activity.
- Call `addToBackStack()` during a transaction that removes the fragment.

## Submit your app for grading

### Guidance for graders

Check that the app has the following feature:

- After the user taps **Open** and makes a choice, the Toast message displays "Choice is Yes" or "Choice is No".

## 2.1: Building app widgets

### Build and run an app

Create an app that has an `EditText` view and a button. When the user taps the button, save the string that's displayed in the `EditText` view to the shared preferences. Ensure that the app loads this string from the preferences in `onCreate()`.

Add a 1x3 app widget that displays the current value of the string. Add a click handler to the entire app widget such that the app opens when the user taps the app widget.

### Answer these questions

## Question 1

Which of these app-widget components are *required* ? (Choose all that apply)

- Provider-info file
- Widget-provider class

- Configuration activity
- Layout file

## Question 2

Which of these layout and view classes can be used in an app widget?

- `Button`
- `ConstraintLayout`
- `LinearLayout`
- `ImageButton`
- `CardView`
- `CalendarView`

## Question 3

In which method in your widget-provider class do you initialize the layout (remote views) for the app widget?

- `onCreate()`
- `onReceive()`
- `onEnabled()`
- `onUpdate()`

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

- The main activity of the app has an `EditText` view and a button. If you change the string in the `EditText` and tap the button, the string is saved. Quitting and restarting the app should load the current string from the shared preferences.
- The app widget displays the current text in the app's `EditText`.
- Tapping the app widget opens the app.

## 3.1: Working with sensor data

### Build and run an app

Create an app to print the current value of a device's humidity sensor (`TYPE_RELATIVE_HUMIDITY`). If the sensor is not available in the device, print "no sensor" instead of the value.

## Answer these questions

### Question 1

Which of the following features are provided by the `sensorManager` class? (Choose all that apply)

- Methods to register and unregister sensor listeners.
- Methods to determine device orientation.
- Constants representing sensor types.
- Constants representing sensor accuracy.
- Methods to indicate whether a sensor is a wake-up sensor.

### Question 2

In which `Activity` lifecycle method should you register your sensor listeners?

- `onResume()`
- `onCreate()`
- `onStart()`
- `onRestart()`

### Question 3

What are best practices for using sensors in your app? (Choose all that apply)

- Register listeners for only for the sensors you're interested in.
- Test to make sure that a sensor is available on the device before you use the sensor.
- Check permissions for the sensor before you use it.
- Register a sensor listener for the slowest possible data rate.
- Don't block `onSensorChanged()` to filter or transform incoming data.

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

- In `onCreate()`, the app should retrieve `Sensor.TYPE_RELATIVE_HUMIDITY` from the sensor manager.
- In `onStart()`, the app should register a listener for the humidity sensor. In `onStop()`, the app should unregister all listeners.
- The app should implement `onSensorChanged()` and test that the event's sensor type is `TYPE_HUMIDITY`.
- When you run the app in the emulator and change the value of the humidity sensor, the app's display should reflect the current value.

## 3.2: Working with sensor-based orientation

### Build and run an app

Create an app based on the [TiltSpot](#) app that simulates a bubble level along the long edge of the device. (A *bubble or spirit level* is a tool that uses colored liquid and an air bubble to show whether a surface is level.) Use a single spot as the "bubble." When the device is level (flat on the table), the bubble should appear in the center of the screen. Move the spot left or right on the screen when one long edge of the device is tilted.

### Answer these questions

#### Question 1

Which sensors report values with respect to Earth's coordinate system, instead of reporting values with respect to the device-coordinate system?

- Geomagnetic field sensor
- Accelerometer
- Gyroscope
- Orientation sensor

#### Question 2

Which sensors can you use to get the orientation of the device?

- Gyroscope
- Accelerometer and gravity sensor
- Orientation sensor
- Geomagnetic field sensor and accelerometer

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

- When you place the device flat on a table, a spot should appear in the middle of the screen.
- When you lift the top or bottom of the device, the spot should move to the upper or lower edge of the screen. The position of the spot should be further away from the center as you tilt the device at a greater angle.

## 4.0: Optimizing for performance

### Build and run an app

Build an app that shows a scrollable list of items for sale, with thumbnail images. When the user taps an image, the app displays a details-view popup with a larger version of the image. You will use this app to complete other performance homework assignments.

- Images must be in WebP format and appropriately sized.
- Supply separate images for thumbnails and detail images.
- To show the product image when the user taps a thumbnail image, you can use a [dialog](#).

### Answer these questions

#### Question 1

Select all of the following that are good basic performance tests you can perform.

- Install your app on the lowest-end device that your target audience might have.
- Observe your friends using the app and make note of their comments.
- Run a small usability test.
- Publish your app and look at the ratings and feedback you receive.

#### Question 2

Select all that are good ways to approach performance problems.

- Guess at what might be the problem, make a change, and see whether it helps.

- Use a systematic, iterative approach, so that you can measure improvements resulting from your changes to the app.
- Use tools to inspect your app and acquire performance data measurements.
- Run your code and have someone else run your code to evaluate it.

## Question 3

Select all of the following that are performance tools available on your mobile device.

- Profile GPU Rendering
- USB debugging
- Show GPU view updates
- Memory Profiler
- Debug GPU Overdraw
- Show CPU usage

## Question 4

What is the Performance Improvement Lifecycle technique for improving app performance?

- A systematic approach to testing and improving app performance.
- A process with four phases: *gather information* , *gain insight* , *take action* , and *verify* .
- A set of tools you can use to measure and improve app performance.
- An iterative approach that allows you to evaluate how changes to your code affect app performance.

## Submit your app for grading

App should show a list or grid of items with thumbnail images. When you tap an item, a larger image of the item is displayed.

## Guidance for graders

Check that the app has the following features:

- App shows a list of items.
- Tapping an item shows a larger image of the item.
- Images load without delay.
- Scrolling is smooth.

## 4.1: Analyzing rendering and layout

## Build and run an app

In the app you created for the 4.0 homework assignment ([4.0: Optimizing for performance](#)), make the following changes:

1. Use the same larger images for the thumbnails as for the details view . Run Profile GPU Rendering with the app, take a screenshot of the bars.
2. Change the app to use small thumbnail images or text to list the items. Run Profile GPU Rendering with the app again and take a screenshot of the bars.
3. Do the two screenshots look different? Comment on the difference or lack of difference.
4. Turn on Debug GPU Overdraw. Take a screenshot of your app. If there is a lot of red color, use Layout Inspector to fix your app, then take another screenshot.

## Answer these questions

### Question 1

How much time does your app have available to calculate and display one frame of content?

- 16 milliseconds
- Less than 16 milliseconds, because the Android framework is doing work at the same time as your app does work.
- It depends on the device.

### Question 2

What are some techniques you can use to make rendering faster?

- Reduce overdraw.
- Move work away from the UI thread.
- Use `AsyncTask` .
- Use smaller images.
- Compress your data.
- Flatten your view hierarchy.
- Use as few views as possible.
- Use loaders to load data in the background.
- Use efficient views such as `RecyclerView` and `ConstraintLayout` .

### Question 3

Which answer best describes the measure-and-layout stage of the rendering pipeline?

- The GPU measures device dimensions and appropriately sizes views.
- The system traverses the view hierarchy and calculates the size and position of each view inside the view's parent, relative to other views.
- The system optimizes layout times for your views.

## Submit your findings for grading

No app to submit for this homework assignment.

Submit the two screenshots you took with Profile GPU Rendering turned on, and the final screenshot you took with Debug GPU Overdraw turned on. Also submit your reflections on why the two screenshots that show Profile GPU Rendering output might be (almost) the same, or why they are different.

## Guidance for graders

Check that:

- Student submitted two screenshots of the app with Profile GPU Rendering turned on
- Screenshots are different.
- Student submitted comments explaining the difference.
- If there is no difference, student should reflect on why not.
- If there is a difference, students should reflect on what may have caused it.
- Student submitted one screenshot of the app with Debug GPU Overdraw turned on, and this screenshot shows only a little bit of red coloring.

## 4.2: Using the Memory Profiler

### Build and run an app

1. Use the app you created for the 4.0 homework assignment ([4.0: Optimizing for performance](#)).
2. Run Memory Profiler on the app.
3. Take a screenshot of the Memory graph to submit with your observations.
4. On the screenshot, identify portions of the graph that are relevant to your app and reflect on them.
5. Change your app in a way that noticeably changes the graph. Take a screenshot and submit it with an explanation of the changes.

### Answer these questions

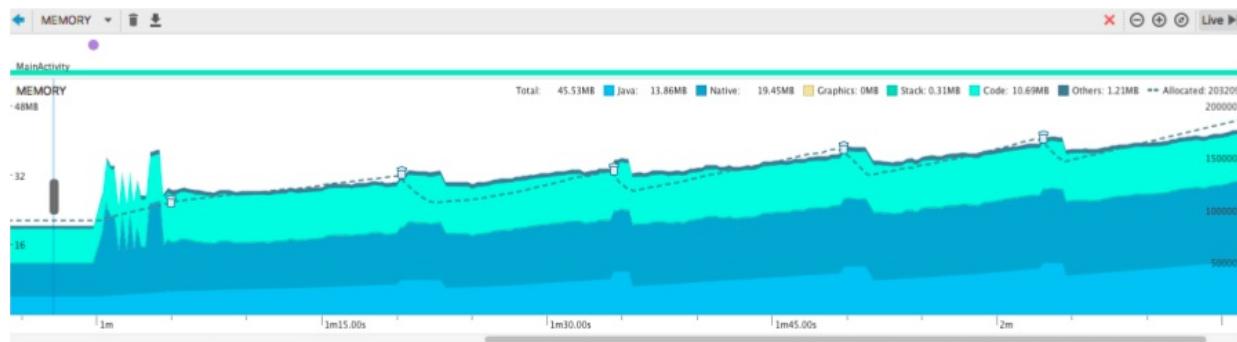
## Question 1

What tools are available in Android Studio for measuring memory performance?

- Systrace
- Heap dumps
- Debug GPU Overdraw
- Memory Profiler

## Question 2

Looking at this Memory Monitor graph, which of the following statements is factually true?



- The app is allocating an increasing amount of memory, and you should investigate for possible memory leaks.
- The app will run out of memory and crash.
- The app will slow down over time because less memory is available.
- More garbage collection events will happen, resulting in a slower app.

## Question 3

Select all answers that describe features and benefits of a heap dump?

- A heap dump is a snapshot of the allocated memory at a specific time.
- You can look at a static snapshot of allocated memory.
- You can dump the Java heap to see which objects are using up memory at any given time.
- Heap dumps show how memory is allocated over time.
- Doing several heap dumps over an extended period can help you identify memory leaks.

## Question 4

What are the benefits of recording memory allocations? Select up to four.

- You can track allocations and deallocations over time.
- You can track how much space files take up on each partition.
- You can inspect the call stack and find out where in your code an allocation was made.
- You can track down which objects might be responsible for memory leaks.
- You can record memory allocations during normal and extreme user interactions. This recording lets you identify where your code is allocating too many objects in a short time, or allocating objects that leak memory.

## Submit your findings for grading

No app to submit for this homework assignment.

Submit the annotated screenshot that you took of the Memory graph, along with your observations.

## Guidance for graders

Check that:

- Student submitted annotated screenshot of the app with Memory Profiler turned on
- Student submitted comments relating the graph to their app.
- Student submitted a second, different screenshot, along with explanations on what they did to cause this change, as well as reflections on the change.

## 4.3: Optimizing network, battery, and image use

### Answer these questions

#### Question 1

On mobile devices, what uses up battery power?

- Mobile radio
- Wi-Fi
- Keeping the Display open
- Running your device in airplane mode
- Any hardware on the device that is actively in use

#### Question 2

Your app can affect the amount of power that some device components use. Which of the following device components does this include? Select up to three.

- Power and volume buttons
- Display (managing wake lock)
- Mobile radio (batching requests, size of requests)
- GPU (complex visual content, media content)

## Question 3

Which of the following are best practices you should *always* incorporate in your app?

- Defer all network requests until the user's device is on Wi-Fi and plugged in.
- Use the most efficient image format for the type of images your app uses.
- Compress all your data.
- Respond to user actions immediately.
- Always prefetch as much data as possible.
- If you must poll the server, use an exponential back-off pattern.
- To make your app work offline, cache data locally.

## Question 4

What are best practices for working with images?

- Use the smallest image possible. Resize images used for thumbnails.
- Don't use images. They take up too much space.
- Always use a quality setting that does not diminish the user experience but results in a smaller image.
- If you store images on a remote server, make multiple sizes available for each image. That way, your app can request the appropriate size for the device it's running on.
- WebP, PNG, and JPG image formats can be used interchangeably.
- If possible, use the WebP image format, because it usually results in smaller, higher-quality images.

## Submit your findings for grading

No app to submit for this homework assignment.

## 5.1: Using resources for languages

### Build and run an app

In the [LocaleText1](#) app, make the following changes:

1. Add Spanish for Mexico and Arabic for any region.
2. Add translations for both languages.
3. Test the app with both language choices.

## Answer these questions

### Question 1

Which of the following attributes should you add to the `android:layout_marginLeft` attribute to support an RTL language?

- `android:layout_marginEnd`
- `android:layout_marginStart`
- `app:layout_constraintLeft_toRightOf`
- `app:layout_constraintStart_toEndOf`

### Question 2

Which of the following attributes should you add to the `app:layout_constraintLeft_toRightOf` attribute to support an RTL language?

- `app:layout_constraintRight_toLeftOf`
- `app:layout_constraintStart_toEndOf`
- `app:layout_constraintRight_toRightOf`
- `app:layout_constraintEnd_toEndOf`

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

- If the user chooses Español (Mexico), the app displays Spanish.
- If the user chooses Arabic and any country, the app displays Arabic in an RTL layout.

## 5.2: Using the locale

### Build and run an app

The [LocaleText3](#) app calculates and shows a total after the user enters a quantity and taps the **Done** button. Modify the LocaleText3 app as follows:

- Modify the `onEditorAction()` method of the `OnEditorActionListener` anonymous inner class. Add code that calculates the total from the quantity and price, then shows the total.
- The code you add should make the calculation *price* times *quantity*.
- If the user chooses **Français (France)**, the code should show the total in euros. If the user chooses **Hebrew (Israel)**, the code should show the total in new shekels. Otherwise, the code should show the total in U.S. dollars.
- This code is similar to the code you added in the practical to show the price in euros or new shekels.

## Answer these questions

### Question 1

Which of the following statements returns a general-purpose number format for the user-selected language and locale?

- `NumberFormat numberFormat = NumberFormat.getInstance();`
- `String myFormattedQuantity = numberFormat.format(myInputQuantity);`
- `String deviceLocale = Locale.getDefault().getCountry();`

### Question 2

Which of the following statements converts a locale-formatted input quantity string to a number?

- `NumberFormat numberFormat = NumberFormat.getInstance();`
- `String myFormattedQuantity = numberFormat.format(myInputQuantity);`
- `myInputQuantity = numberFormat.parse(v.getText().toString()).intValue();`

### Question 3

Which of the following statements retrieves the country code for the user-chosen locale?

- `String deviceLocale = Locale.getDefault().getCountry();`
- `String deviceLocale = Locale.getDisplayCountry();`
- `String deviceLocale = Locale.getDisplayName();`

## Submit your app for grading

## Guidance for graders

Check that the app has the following features:

- If the user chooses **Français (France)**, the app displays the total in euros.
- If the user chooses **Hebrew (Israel)**, the app displays the total in new shekels.
- If the user chooses any other locale, the app displays the total in U.S. dollars.

## 6.1: Exploring accessibility in Android

### Answer these questions

#### Question 1

Why should you consider making your app accessible?

- Accessibility enables your app to be used by people with disabilities.
- Accessibility allows your users—those users with disabilities and those users without—to customize their experiences of your app.
- Accessible apps run faster.
- Testing your app with accessibility in mind can reveal user-experience issues or limitations you might not have recognized.

#### Question 2

Which of the following are accessibility features available in Android?

- TalkBack
- Switch Access
- Text-to-speech
- Closed captions
- Magnification
- Ability to change display or font size

#### Question 3

What is TalkBack?

- An app that converts audio into readable captions on the screen.
- An app that enables screen content to be presented on a refreshable braille display.
- An app that reads screen content aloud.
- An app that magnifies the computer screen.

## Question 4

How should you test your app for accessibility?

- Turn on TalkBack and manually try to use your app.
- Add unit tests and Espresso tests.
- Use tools such as Accessibility Scanner and Android Studio's lint tool to reveal potential accessibility problems.
- Use the Android Testing Support Library to enable automated accessibility testing.

## Submit your findings for grading

No app to submit for this homework assignment.

## 6.2: Creating accessible apps

### Build and run an app

In the [SimpleAccessibility](#) app, add three new images:

1. Add a description for the first image that describes the image. Ensure that the image is focusable.
2. Add a decorative image with no description. Ensure that the image is *not* focusable.
3. For the third image, add a text label that describes the image.

### Answer these questions

#### Question 1

Which of the following attributes should you add to `ImageView` and `ImageButton` elements to enable screen readers to describe the image?

- `android:text`
- `android:contentDescription`
- `android:hint`
- `android:labelFor`

#### Question 2

When should you add a content description to an `ImageView` or `ImageButton` ?

- When the image's role on the screen is solely decorative and does not provide any

function or meaning.

- When an image's dimensions are very small and the image is difficult to see.
- When the image is meaningful to the user in their use of the app.
- When the image is also a button.

## Question 3

When do you NOT need to add a content description to a view element?

- When the view is a `TextView` or `EditText`.
- When the view is an `ImageView` that serves only a decorative purpose.
- When the view is a checkbox or radio button.
- When the view has an associated text view that includes with the `android:labelFor` attribute.
- All of the above.

## Submit your findings for grading

### Guidance for graders

Check that the app has the following features:

- The app should have three new images, of any kind. The third image should have a text view that serves as a label.
- The layout for the first new image should include the following:
  - An `android:contentDescription` attribute with a description of the image.
  - An `android:focusable="true"` attribute.
- The layout for the second new image should include the following:
  - An `android:contentDescription` attribute with a null string ( `""` ).
  - An `android:focusable="false"` attribute.
- The layout for the third new image should include the following:
  - An `android:focusable="true"` attribute.
  - A `TextView` element with the `android:labelFor` attribute. The `android:labelFor` attribute should indicate the ID of the image.

## 7.1: Using the device location

### Build and run an app

In the [WalkMyAndroid](#) app, add a second `TextView` to the app that shows the following:

1. Accuracy in meters.
2. Speed in meters per second.

**Hint:** See the `getSpeed()` and `getAccuracy()` documentation.

## Answer these questions

### Question 1

Which API do you use to request the last known location on the device?

- `getLastKnownLocation()` method in the `FusedLocationProviderApi` class
- `getLastKnownLocation()` method in the `LocationServices` class
- `getLastLocation()` method in the `FusedLocationProviderClient` class
- `getLastLocation()` method in the `LocationServices` class

### Question 2

Which class do you use for handling geocoding and reverse geocoding?

- `GeoDecoder`
- `Geocoder`
- `ReverseGeocoder`
- `GeocoderDecoder`

### Question 3

Which method do you use for periodic location updates ?

- `requestPerodicUpdates()` method in the `FusedLocationClient` class
- `requestLocationUpdates()` method in the `FusedLocationProviderClient` class
- `requestUpdates()` method in the `FusedLocationProviderClient` class
- `requestLocationUpdates()` method in the `FusedLocationProvider` class

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

- The app displays a second `TextView`, in addition to the `TextView` that shows the address.
- The second `TextView` shows speed and accuracy.

## 8.1 Using the Places API

### Build and run an app

Modify the [WalkMyAndroidPlaces](#) app to display a new Android image for shopping malls:

1. Create a new Android image for shopping malls. (**Hint:** Use the [Androdify](#) tool.)
2. If the place type is `TYPE_SHOPPING_MALL`, replace the Android image with your image.
3. To test your app, pick a shopping mall from the `PlacePicker` dialog. Make sure that your new Android image is animated, and that `TextView` object is updated with the shopping mall's name and address.

### Answer these questions

#### Question 1

Which class displays a dialog that allows a user to pick a [place](#) using an interactive map ?

- `PlaceDetectionApi`
- `GeoDataApi`
- `PlacePicker`
- `PlaceAutocomplete`

#### Question 2

What are the two ways to add the autocomplete widget to your app?

- Embed a `PlaceAutocompleteFragment` fragment.
- Autocomplete is added along with the `PlacePicker` widget.
- Use an intent to launch the autocomplete activity.
- Add a `PlaceAutocompleteActivity` object to your app.

#### Question 3

If your app uses the `PlacePicker` UI or the `PlaceDetectionApi` interface, what permission does your app require?

- `ACCESS_COARSE_LOCATION`
- `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION`
- `ACCESS_LOCATION_EXTRA_COMMANDS`
- `ACCESS_FINE_LOCATION`

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

1. When you tap the **Pick a Place** button, the place picker UI opens.
2. When you pick a shopping mall on the map, the new Android image is displayed.
3. The image is animated.
4. The `TextView` updates to show the name and address of the shopping mall.

## 9.1 Adding a Google Map to your app

### Build and run an app

1. Create a new app that uses the Google Maps Activity template, which loads Google Maps when the app launches.
2. When the Google Map is loaded, move the camera to your school location, your home location, or some other location that has meaning for you.
3. Add two markers to the map, one at your school location and one at your home or some other meaningful location.
4. Customize the marker icons by changing the default color or replacing the default marker icon with a custom image.

**Hint:** See the `onMapReady (GoogleMap googleMap)` documentation.

### Answer these questions

#### Question 1

Which method is called when the map is loaded and ready to be used in the app ?

- `onMapReady ( GoogleMap googleMap )`
- `onMapLoaded ( GoogleMap googleMap )`
- `onMapCreate ( GoogleMap googleMap )`
- `onMapInitialize ( GoogleMap googleMap )`

#### Question 2

Which Android components can you use to include Google Maps in your app ?

- `MapView` and `MapFragment`
- `MapFragment` and `MapActivity`
- `MapView` and `MapActivity`
- Only `MapFragment`

## Question 3

What types of maps does the Google Maps Android API offer?

- Normal, hybrid, terrain, satellite, and roadmap
- Normal, hybrid, terrain, satellite, and "none"
- Hybrid, terrain, satellite, roadmap, and "none"
- Normal, terrain, satellite, imagemap, and "none"

## Question 4

What interface do you implement to add on-click functionality to a point of interest (POI)?

- `GoogleMap.OnPoiListener`
- `GoogleMap.OnPoiClickListener`
- `GoogleMap.OnPoiClick`
- `GoogleMap.OnPoiclicked`

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

- When the app is launched, the Google Map is displayed correctly, indicating that an API key was generated properly.
- After the Google Map loads, the camera moves to the student's home or school location. In the code, this step should happen in the `onMapReady (GoogleMap googleMap)` callback method.
- Markers are displayed at the student's school location and another location, such as the student's home.
- The two markers are customized. For example, the markers use a color other than the default red color, or they use a custom icon.

## 10.1: Creating a custom view

## Build and run an app

In the [CustomEditText](#) app, add a custom view that enables phone-number entry:

1. In the layout, add a second version of the `EditTextWithClear` custom view underneath the first version (the **Last name** field).
2. Use XML attributes to define the second version of the custom view as a phone number field that accepts only numeric phone numbers as input.

## Answer these questions

### Question 1

Which constructor do you need to inflate the layout for a custom view? Choose one:

- `public MyCustomView(Context context)`
- `public MyCustomView(Context context, AttributeSet attrs)`
- `public static SimpleView newInstance() { return new SimpleView(); }`
- `protected void onDraw(Canvas canvas) { super.onDraw(canvas) }`

### Question 2

To define how your custom view fits into an overall layout, which method do you override?

- `onMeasure()`
- `onSizeChanged()`
- `invalidate()`
- `onDraw()`

### Question 3

To calculate the positions, dimensions, and any other values when the custom view is first assigned a size, which method do you override?

- `onMeasure()`
- `onSizeChanged()`
- `invalidate()`
- `onDraw()`

### Question 4

To indicate that you'd like your view to be redrawn with `onDraw()`, which method do you call from the UI thread, after an attribute value has changed?

- `onMeasure()`
- `onSizeChanged()`
- `invalidate()`
- `getVisibility()`

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

- The app displays a **Phone number** field with a clear (X) button on the right side of the field, just like the **Last name** field.
- The second version of the `EditTextWithClear` custom field (**Phone number**) should use the `android:inputType` attribute so users can enter values with a numeric keypad.

## 11.1: Applying clipping to a Canvas object

### Build and run an app

Create a MemoryGame app that hides and reveals "cards" as the user taps on the screen. Use clipping to implement the hide/reveal effect.

- You can use simple colored squares or shapes for the "cards."
- If the user reveals two matching cards, show a congratulatory toast. If the user reveals two cards that don't match, show an encouraging message telling them to tap to continue.
- Click handling: On the first tap, show the first card. On the second tap, show the second card and display a message. On the next tap, restart.

### Answer these questions

#### Question 1

To display something to the screen, which one of the following draw and animation classes is always required?

- `View`
- `Drawable`
- `Canvas`
- `Bitmap`

## Question 2

What are some properties of drawables?

- Drawables are drawn into a view and the system handles drawing.
- Drawables are best for simple graphics that do not change dynamically.
- Drawables offer the best performance for game animations.
- You can use drawables for frame-by-frame animations.

## Question 3

Which of the following statements are true?

- You use a `Canvas` object when elements in your app are redrawn regularly.
- To draw on a `Canvas`, you must override the `onDraw()` method of a custom view.
- Every view has a `Canvas` that you can access.
- A `Paint` object holds style and color information about how to draw geometries, text, and bitmaps.

## Question 4

What is clipping?

- A technique for defining regions on a `Canvas` that will not be drawn to the screen.
- A technique for making the `Canvas` smaller so the `Canvas` uses less memory.
- A way of telling the system which portions of a `Canvas` do not need to be redrawn.
- A technique to consider when you're trying to speed up drawing.
- A way to create interesting graphical effects.

## Submit your app for grading

## Guidance for graders

Check that the app has the following features:

- When the user taps, the app reveals a "card."
- When the user taps again, the app reveals a second "card" and shows a toast congratulating or encouraging the user.
- On the third tap, the game restarts.
- The code uses a `Canvas` object and clipping methods to achieve the hide/reveal effects of playing the game.

## 11.2: Creating a SurfaceView object

### Build and run an app

Implement the same MemoryGame app that you created in the 11.1 homework, but use a `SurfaceView` object.

The MemoryGame app hides and reveals "cards" as the user taps on the screen. Use clipping to implement the hide/reveal effect.

- You can use simple colored squares or shapes for the "cards."
- If the user reveals two matching cards, show a congratulatory toast. If the user reveals two cards that don't match, show an encouraging message telling them to tap to continue.
- Click handling: On the first tap, show the first card. On the second tap, show the second card and display a message. On the next tap, restart.

### Answer these questions

#### Question 1

What is a `SurfaceView` ?

- A view in your app's view hierarchy that has its own separate surface.
- A view that directly accesses a lower-level drawing surface.
- A view that is not part of the view hierarchy.
- A view that can be drawn to from a separate thread.

#### Question 2

What is the most distinguishing benefit of using a `SurfaceView` ?

- A `SurfaceView` can make an app more responsive to user input.
- You can move drawing operations away from the UI thread.
- Your animations may run more smoothly.

#### Question 3

When should you consider using a `SurfaceView` ? Select up to three.

- When your app does a lot of drawing, or does complex drawing.
- When your app combines complex graphics with user interaction.

- When your app uses a lot of images as backgrounds.
- When your app stutters, and moving drawing off the UI thread could improve performance.

## Submit your app for grading

### Guidance for graders

Check that the app has the following features:

- When the user taps, the app reveals a "card."
- When the user taps again, the app reveals a second "card" and shows a toast congratulating or encouraging the user.
- On the third tap, the game restarts.
- The code uses a `SurfaceView` object, and a separate thread for drawing.

## 12.1: Creating animations

### Build and run an app

Create an app that uses non-trivial property animation. For example, animate multiple properties, animate multiple objects, or create complex animations by using animator sets.

Since this is your last homework for the course, get creative! Below are some ideas.

- Create an animation where text spins and recedes while getting smaller. Or text appears from nowhere and spins to fill the screen. Combine these two animations.
- Create an animation that simulates a ball that grows until it bursts into multiple smaller balls.
- Create a simple card game, where touching a card flips the card around.

### Answer these questions

#### Question 1

What types of animations are available with Android?

- View animation
- Property animation
- Canvas animation
- Drawable animation

- Physics-based animation

## Question 2

Which of the following statements about property animation are true?

- Property animation lets you define an animation to change any object property over time.
- Property animation lets you create objects with custom properties that you can animate.
- A property animation tracks time and adapts its velocity to the time.
- Property animation lets you animate multiple properties with animator sets.
- The duration of a property animation is fixed.

## Question 3

What are the advantages of using physics-based animation libraries? Select up to three answers.

- Physics-based animations are more realistic than other types of animations, because physics-based animations appear more natural.
- It is easier to use the physics-based support library than to implement adaptive animations yourself.
- Physics-based animations keep momentum when their target changes and end with a smoother motion than other types of animations.
- Physics-based animations are easier to combine with audio.

## Submit your app for grading

## Guidance for graders

Check that the app has the following features:

- App uses property animation to implement a non-trivial animation, animates multiple objects, and/or uses animator sets.