Android Developer Fundamentals (Version 2) course

Last updated Tue Sep 11 2018

This course was created by the Google Developer Training team.

- For details about the course, including links to all the concept chapters, apps, and slides, see Android Developer Fundamentals (Version 2).

  developer.android.com/courses/adf-v2

Note: This course uses the terms "codelab" and "practical" interchangeably.

We advise you to use the online version of this course rather than this static PDF to ensure you are using the latest content.

See developer.android.com/courses/adf-v2.

Unit 2: User Experience

This PDF contains a one-time snapshot of the lessons in **Unit 2: User Experience.**

Lessons in this unit

Lesson 4: User interaction

4.1 : Clickable images

4.2 : Input controls

4.3 : Menus and pickers

4.4 : User navigation

4.5 : RecyclerView

Lesson 5: Delightful user experience

5.1 : Drawables, styles, and themes

5.2 : Cards and colors

5.3 : Adaptive layouts

Lesson 6: Testing your UI

6.1 : Espresso for UI testing

Lesson 4.1: Clickable images

Introduction

The user interface (UI) that appears on a screen of an Android-powered device consists of a hierarchy of objects called *views*. Every element of the screen is a `View`.

The `View` class represents the basic building block for all UI components. `View` is the base class for classes that provide interactive UI components, such as `Button` elements. A `Button` is a UI element the user can tap or click to perform an action.

You can turn any `View`, such as an `ImageView`, into a UI element that can be tapped or clicked. You must store the image for the `ImageView` in the `drawables` folder of your project.

In this practical, you learn how to use images as elements that the user can tap or click.

What you should already know

You should be able to:

- Create an Android Studio project from a template and generate the main layout.
- Run apps on the emulator or a connected device.
- Create and edit UI elements using the layout editor and XML code.
- Access UI elements from your code using `findViewById()`.
- Handle a `Button` click.
- Display a `Toast` message.
- Add images to a project's `drawable` folder.

What you'll learn

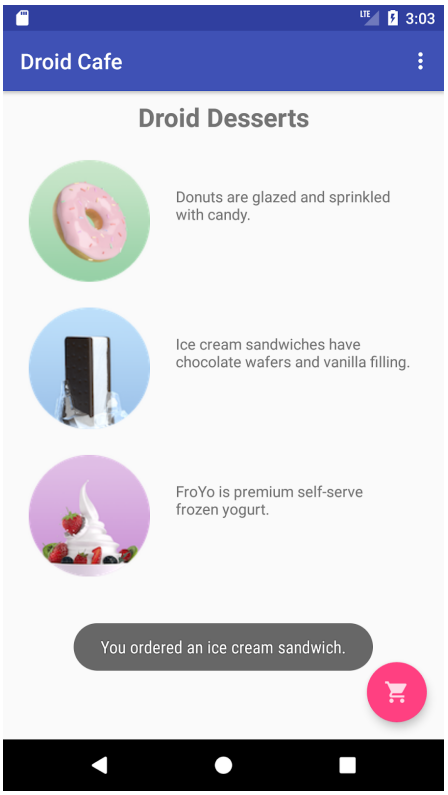
- How to use an image as an interactive element to perform an action.
- How to set attributes for `ImageView` elements in the layout editor.
- How to add an `onClick()` method to display a `Toast` message.

What you'll do

- Create a new Android Studio project for a mock dessert-ordering app that uses images as interactive elements.
- Set `onClick()` handlers for the images to display different `Toast` messages.
- Change the floating action button supplied by the template so that it shows a different icon and launches another `Activity`.

App overview

In this practical, you create and build a new app starting with the Basic Activity template that imitates a dessert-ordering app. The user can tap an image to perform an action—in this case display a `Toast` message—as shown in the figure below. The user can also tap a shopping-cart button to proceed to the next `Activity`.



Task 1: Add images to the layout

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

In this task you create a prototype of an app for ordering desserts from a café. After starting a new project based on the Basic Activity template, you modify the "Hello World" `TextView` with appropriate text, and add images that the user can tap.

1.1 Start the new project

1. Start a new Android Studio project with the app name **Droid Cafe**.
2. Choose the **Basic Activity** template, and accept the default `Activity` name (`MainActivity`). Make sure the **Generate Layout file** and **Backwards Compatibility (AppCompat)** options are selected.
3. Click **Finish**.

   The project opens with two layouts in the **res > layout** folder: `activity_main.xml` for the app bar and floating action button (which you don't change in this task), and `content_main.xml` for everything else in the layout.

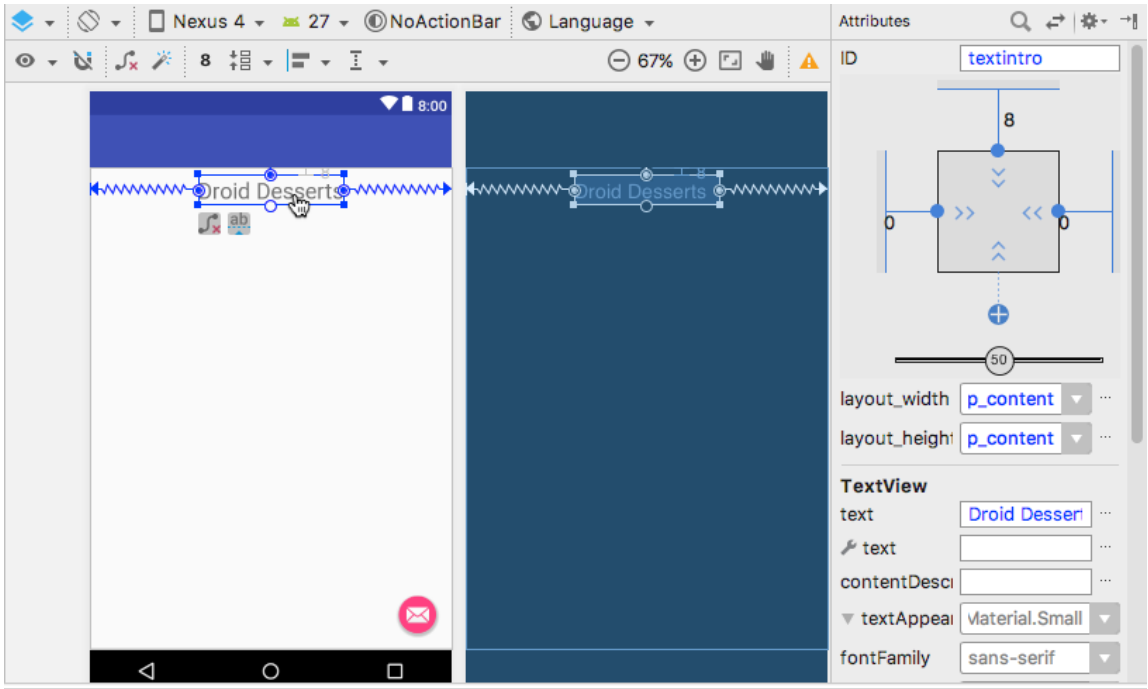
4. Open **content_main.xml** and click the **Design** tab (if it is not already selected) to show the layout editor.
5. Select the "Hello World" `TextView` in the layout and open the **Attributes** pane.
6. Change the `textintro` attributes as follows:

Attribute field	**Enter the following:**

`ID`	`textintro`
`text`	Change `Hello World` to Droid Desserts
`textStyle`	`B` (bold)
`textSize`	`24sp`

This adds the `android:id` attribute to the `TextView` with the `id` set to `textintro`, changes the text, makes the text bold, and sets a larger text size of `24sp`.

7. Delete the constraint that stretches from the bottom of the `textintro` `TextView` to the bottom of the layout, so that the `TextView` snaps to the top of the layout, and choose **8** (8dp) for the top margin as shown below.



8. In a previous lesson you learned how to extract a string resource from a literal text string. Click the **Text** tab to switch to XML code, and extract the `"Droid Desserts"` string in the

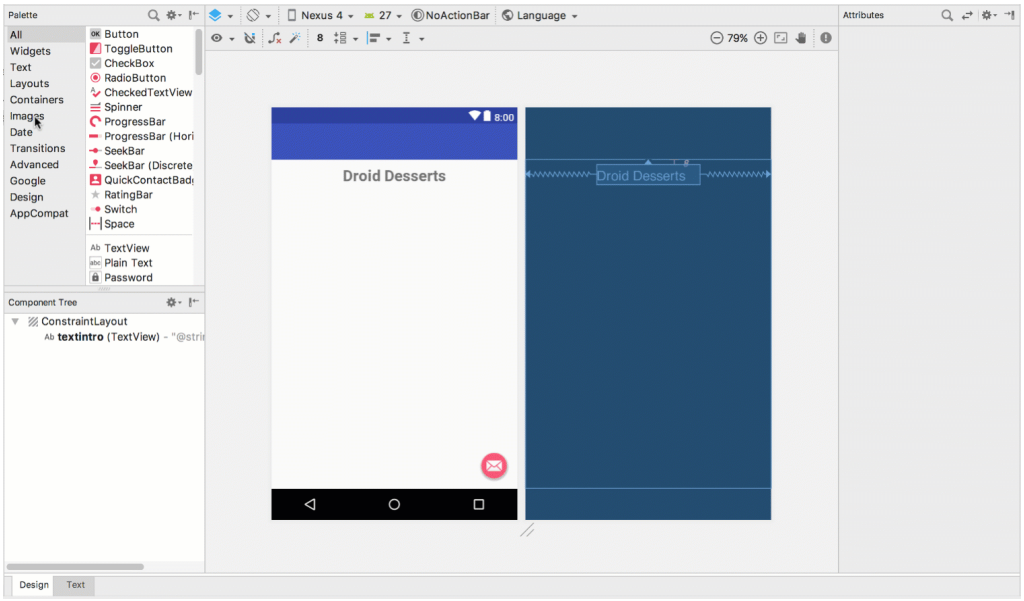
`TextView` and enter **intro_text** as the string resource name.

1.2 Add the images

Three images (`donut_circle.png`, `froyo_circle.png`, and `icecream_circle.png`) are provided for this example, which you can download. As an alternative, you can substitute your own images as PNG files, but they must be sized at about 113 x 113 pixels to use in this example.

This step also introduces a new technique in the layout editor: using the **Fix** button in warning messages to extract string resources.

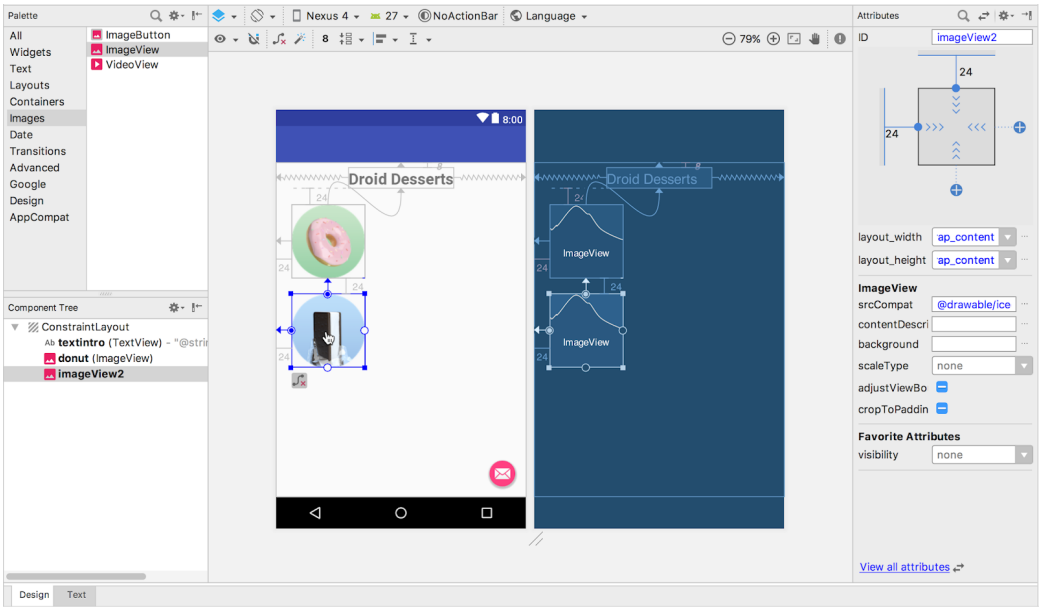
1. To copy the images to your project, first close the project.
2. Copy the image files into your project's **drawable** folder. Find the **drawable** folder in a project by using this path: *project_name* **> app > src > main > res > drawable**.
3. Reopen your project.
4. Open **content_main.xml** file, and click the **Design** tab (if it is not already selected).
5. Drag an `ImageView` to the layout, choose the **donut_circle** image for it, and constrain it to the top `TextView` and to the left side of the layout with a margin of **24** (24dp) for both constraints, as shown in the animated figure below.



6. In the **Attributes** pane, enter the following values for the attributes:

Attribute field	**Enter the following:**
`ID`	`donut`
contentDescription	`Donuts are glazed and sprinkled with candy.` (You can copy/paste the text into the field.)

7. Drag a second `ImageView` to the layout, choose the **icecream_circle** image for it, and constrain it to the bottom of the first `ImageView` and to the left side of the layout with a margin of **24** (24dp) for both constraints.



8. In the **Attributes** pane, enter the following values for the attributes:

Attribute field	**Enter the following:**
`ID`	`ice_cream`
`contentDescription`	`Ice cream sandwiches have chocolate wafers and vanilla filling.` (You can copy/paste the text into the field.)

9. Drag a third `ImageView` to the layout, choose the **froyo_circle** image for it, and constrain it to the bottom of the second `ImageView` and to the left side of the layout with a margin of **24** (24dp) for both constraints.

10. In the **Attributes** pane, enter the following values for the attributes:

Attribute field	**Enter the following:**
`ID`	`froyo`

`contentDescription`	`FroYo is premium self-serve frozen yogurt.` (You can copy/paste the text into the field.)

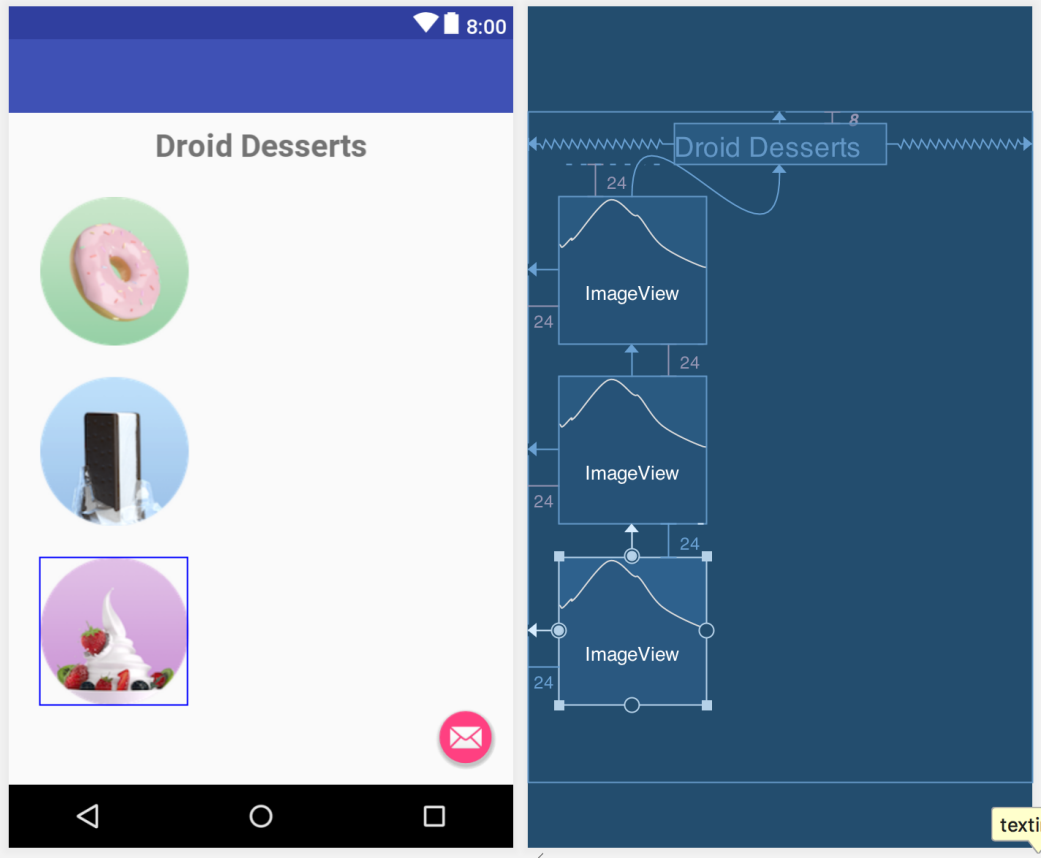
11. Click the warning icon in the upper left corner of the layout editor to open the warning pane, which should display warnings about hardcoded text:

12. Expand each **Hardcoded text** warning, scroll to the bottom of the warning message, and click the **Fix** button as shown below:

The fix for each hardcoded text warning extracts the string resource for the string. The **Extract Resource** dialog appears, and you can enter the name for the string resource. Enter the following names for the string resources:

String	**Enter the following name:**
`Donuts are glazed and sprinkled with candy.`	`donuts`
`Ice cream sandwiches have chocolate wafers and vanilla filling.`	`ice_cream_sandwiches`
`FroYo is premium self-serve frozen yogurt.`	`froyo`

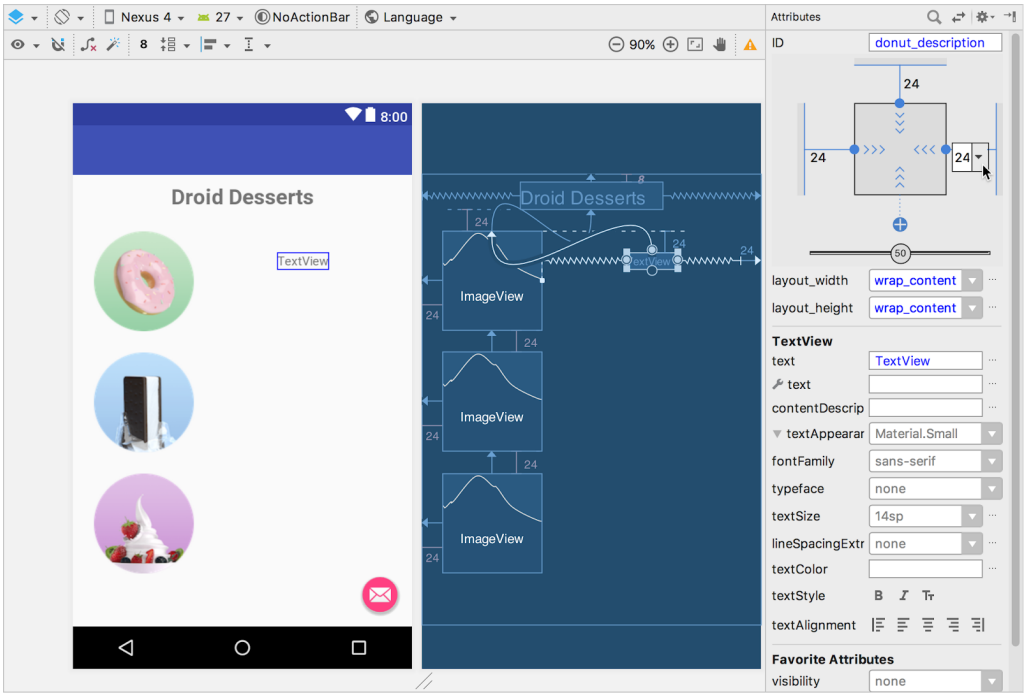
The layout should now look like the figure below.



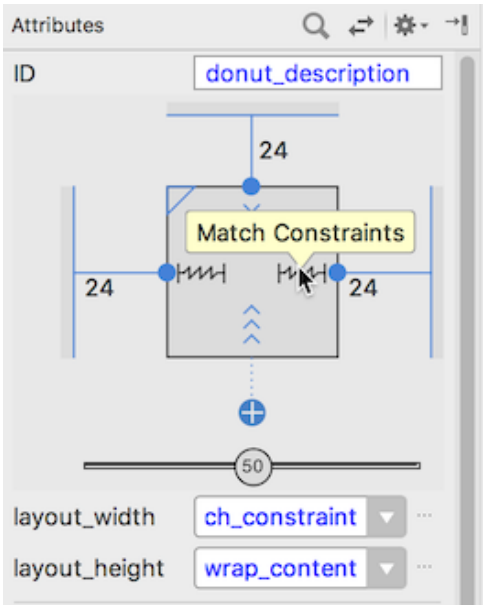
1.3 Add the text descriptions

In this step you add a text description (`TextView`) for each dessert. Because you have already extracted string resources for the `contentDescription` fields for the `ImageView` elements, you can use the same string resources for each description `TextView`.

1. Drag a `TextView` element to the layout.
2. Constrain the element's left side to the right side of the `donut ImageView` and its top to the top of the `donut ImageView`, both with a margin of **24** (24dp).
3. Constrain the element's right side to the right side of the layout, and use the same margin of **24** (24dp). Enter **donut_description** for the `ID` field in the **Attributes** pane. The new `TextView` should appear next to the donut image as shown in the figure below.

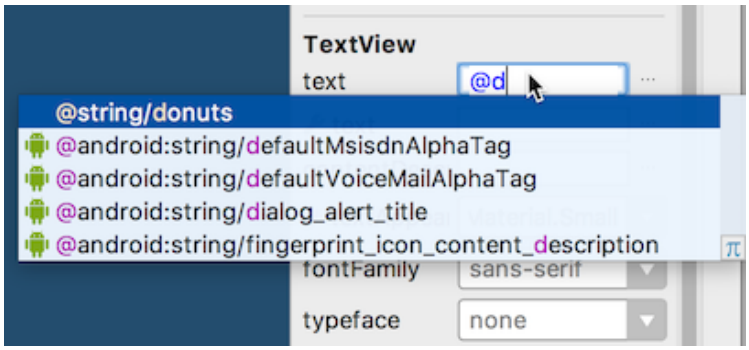


4. In the **Attributes** pane change the width in the inspector pane to **Match Constraints**:



5. In the **Attributes** pane, begin entering the string resource for the `text` field by prefacing it with the **@** symbol: **@d**. Click the string resource name (**@string/donuts**) which appears as a

suggestion:



6. Repeat the steps above to add a second `TextView` that is constrained to the right side and top of the `ice_cream` `ImageView`, and its right side to the right side of the layout. Enter the following in the **Attributes** pane:

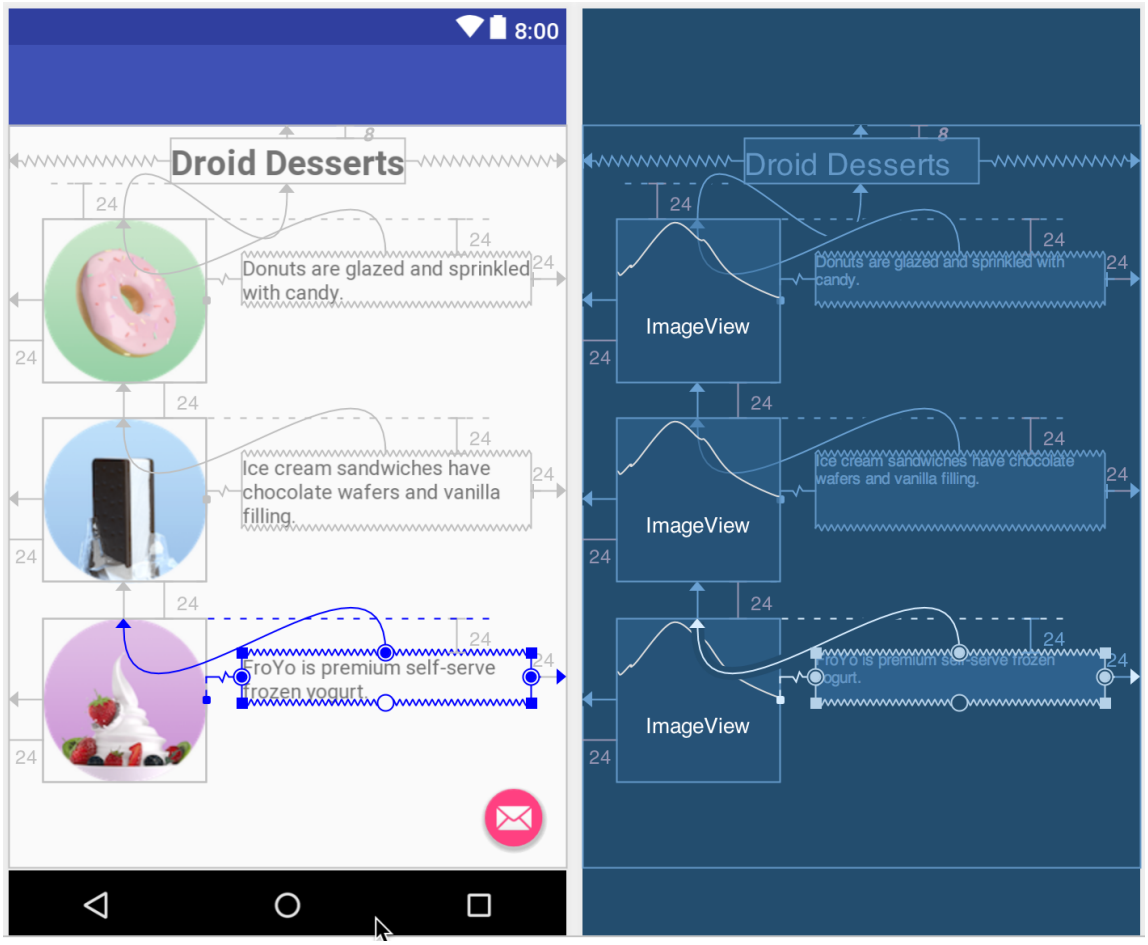
Attribute field	**Enter the following:**
`ID`	`ice_cream_description`
Left, right, and top margins	`24`
`layout_width`	`match_constraint`
`text`	`@string/ice_cream_sandwiches`

7. Repeat the steps above to add a third `TextView` that is constrained to the right side and top of the `froyo` `ImageView`, and its right side to the right side of the layout. Enter the following in the **Attributes** pane:

Attribute field	**Enter the following:**
`ID`	`froyo_description`
Left, right, and top margins	`24`
`layout_width`	`match_constraint`

`text`	`@string/froyo`

The layout should now look like the following:



Task 1 solution code

The XML layout for the content.xml file is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<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"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.example.android.droidcafe.MainActivity"
    tools:showIn="@layout/activity_main">

    <TextView
        android:id="@+id/textintro"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_regular"
        android:text="@string/intro_text"
        android:textSize="@dimen/text_heading"
        android:textStyle="bold"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <ImageView
        android:id="@+id/donut"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/margin_wide"
        android:layout_marginTop="@dimen/margin_wide"
        android:contentDescription="@string/donuts"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textintro"
        app:srcCompat="@drawable/donut_circle" />

    <ImageView
        android:id="@+id/ice_cream"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/margin_wide"
        android:layout_marginTop="@dimen/margin_wide"
        android:contentDescription="@string/ice_cream_sandwiches"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/donut"
```

```
        app:srcCompat="@drawable/icecream_circle" />

    <ImageView
        android:id="@+id/froyo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/margin_wide"
        android:layout_marginTop="@dimen/margin_wide"
        android:contentDescription="@string/froyo"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/ice_cream"
        app:srcCompat="@drawable/froyo_circle" />

    <TextView
        android:id="@+id/donut_description"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="@dimen/margin_wide"
        android:layout_marginStart="@dimen/margin_wide"
        android:layout_marginTop="@dimen/margin_wide"
        android:text="@string/donuts"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/donut"
        app:layout_constraintTop_toTopOf="@+id/donut" />

    <TextView
        android:id="@+id/ice_cream_description"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="@dimen/margin_wide"
        android:layout_marginStart="@dimen/margin_wide"
        android:layout_marginTop="@dimen/margin_wide"
        android:text="@string/ice_cream_sandwiches"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/ice_cream"
        app:layout_constraintTop_toTopOf="@+id/ice_cream" />

    <TextView
        android:id="@+id/froyo_description"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="@dimen/margin_wide"
        android:layout_marginStart="@dimen/margin_wide"
        android:layout_marginTop="@dimen/margin_wide"
        android:text="@string/froyo"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/froyo"
        app:layout_constraintTop_toTopOf="@+id/froyo" />
```

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

Task 2: Add onClick methods for images

To make a `View` *clickable* so that users can tap (or click) it, add the `android:onClick` attribute in the XML layout and specify the click handler. For example, you can make an `ImageView` act like a simple `Button` by adding `android:onClick` to the `ImageView`. In this task you make the images in your layout clickable.

2.1 Create a Toast method

In this task you add each method for the `android:onClick` attribute to call when each image is clicked. In this task, these methods simply display a `Toast` message showing which image was tapped. (In another chapter you modify these methods to launch another `Activity`.)

1. To use string resources in Java code, you should first add them to the `strings.xml` file. Expand **res > values** in the **Project > Android** pane, and open **strings.xml**. Add the following string resources for the strings to be shown in the `Toast` message:

```
<string name="donut_order_message">You ordered a donut.</string>
<string name="ice_cream_order_message">You ordered an ice cream sandwich.</string>
<string name="froyo_order_message">You ordered a FroYo.</string>
```

2. Open **MainActivity**, and add the following `displayToast()` method to the end of **MainActivity** (before the closing bracket):

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

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

2.2 Create click handlers

Each clickable image needs a click handler—a method for the `android:onClick` attribute to call. The click handler, if called from the `android:onClick` attribute, must be `public`, return `void`, and define a `View` as its only parameter. Follow these steps to add the click handlers:

1. Add the following `showDonutOrder()` method to **MainActivity**. For this task, use the previously created `displayToast()` method to display a `Toast` message:

```
/**
* Shows a message that the donut image was clicked.
*/
public void showDonutOrder(View view) {
    displayToast(getString(R.string.donut_order_message));
}
```

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

2. Add more methods to the end of **MainActivity** for each dessert:

```
/**
* Shows a message that the ice cream sandwich image was clicked.
*/
public void showIceCreamOrder(View view) {
    displayToast(getString(R.string.ice_cream_order_message));
}

/**
* Shows a message that the froyo image was clicked.
*/
public void showFroyoOrder(View view) {
    displayToast(getString(R.string.froyo_order_message));
}
```

3. (Optional) Choose **Code > Reformat Code** to reformat the code you added in `MainActivity` to conform to standards and make it easier to read.

2.3 Add the onClick attribute

In this step you add `android:onClick` to each of the `ImageView` elements in the `content_main.xml` layout. The `android:onClick` attribute calls the click handler for each element.

1. Open the **content_main.xml** file, and click the **Text** tab in the layout editor to show the XML code.
2. Add the `android:onClick` attribute to `donut` ImageView. As you enter it, suggestions appear showing the click handlers. Select the `showDonutOrder` click handler. The code should now look as follows:

```
<ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:padding="10dp"
        android:id="@+id/donut"
        android:layout_below="@id/choose_dessert"
        android:contentDescription="@string/donut"
        android:src="@drawable/donut_circle"
        android:onClick="showDonutOrder"/>
```

The last line (`android:onClick="showDonutOrder"`) assigns the click handler (`showDonutOrder`) to the `ImageView`.

3. (Optional) Choose **Code > Reformat Code** to reformat the XML code you added in `content_main.xml` to conform to standards and make it easier to read. Android Studio automatically moves the `android:onClick` attribute up a few lines to combine them with the other attributes that have `android:` as the preface.
4. Follow the same procedure to add the `android:onClick` attribute to the `ice_cream` and `froyo` `ImageView` elements. Select the `showDonutOrder` and `showFroyoOrder` click handlers. You can optionally choose **Code > Reformat Code** to reformat the XML code. The code should now look as follows:

```
<ImageView

        android:id="@+id/ice_cream"

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:layout_marginStart="@dimen/margin_wide"

        android:layout_marginTop="@dimen/margin_wide"

        android:contentDescription="@string/ice_cream_sandwiches"

        android:onClick="showIceCreamOrder"

        app:layout_constraintStart_toStartOf="parent"

        app:layout_constraintTop_toBottomOf="@+id/donut"

        app:srcCompat="@drawable/icecream_circle" />
```

```
<ImageView
        android:id="@+id/froyo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/margin_wide"
        android:layout_marginTop="@dimen/margin_wide"
        android:contentDescription="@string/froyo"
        android:onClick="showFroyoOrder"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/ice_cream"
        app:srcCompat="@drawable/froyo_circle" />
```

Note that the attribute `android:layout_marginStart` in each `ImageView` is underlined in red. This attribute determines the "start" margin for the `ImageView`, which is on the left side for most languages but on the right side for languages that read right-to-left (RTL).

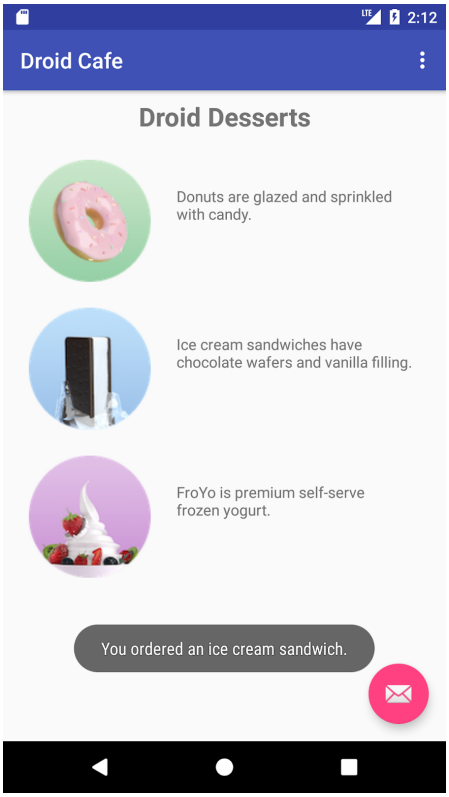
5. Click the `android:` preface part of the `android:layout_marginStart` attribute, and a red bulb warning appears next to it, as shown in the figure below.

```
<ImageView
    android:id="@+id/ice_cream"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="@dimen/margin_wide"
    android:layout_marginTop="@dimen/margin_wide"
    android:contentDescription="@string/ice_cream_sandwiches"
    android:onClick="showIceCreamOrder"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/donut"
    app:srcCompat="@drawable/icecream_circle" />
```

6. To make your app compatible with previous versions of Android, click the red bulb for each instance of this attribute, and choose **Set layout_marginLeft...** to set the `layout_marginLeft` to `"@dimen/margin_wide"`.

7. Run the app.

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



Task 2 solution code

The solution code for this task is included in the code and layout for `MainActivity` in the Android Studio project DroidCafe.

Task 3: Change the floating action button

When you click the floating action button with the email icon that appears at the bottom of the screen, the code in `MainActivity` displays a brief message in a drawer that opens from the bottom of the screen on a smartphone, or from the lower left corner on larger devices, and then closes after a few seconds. This is called a *snackbar*. It is used to provide feedback about an operation. For more information, see `Snackbar`.

Look at how other apps implement the floating action button. For example, the Gmail app provides a floating action button to create a new email message, and the Contacts app provides one to create a new contact. For more information about floating action buttons, see `FloatingActionButton`.

For this task you change the icon for the `FloatingActionButton` to a shopping cart  , and change the action for the `FloatingActionButton` to launch a new `Activity`.

3.1 Add a new icon

As you learned in another lesson, you can choose an icon from the set of icons in Android Studio. Follow these steps:

1. Expand **res** in the **Project > Android** pane, and right-click (or Control-click) the **drawable** folder.
2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.
3. Choose **Action Bar and Tab Icons** in the drop-down menu at the top of the dialog. (Note that the *action bar* is the same thing as the *app bar*.)
4. Change **ic_action_name** in the **Name** field to **ic_shopping_cart**.
5. Click the clip art image (the Android logo next to **Clipart:**) to select a clip art image as the icon. A page of icons appears. Click the icon you want to use for the floating action button, such as the shopping cart icon.

6. Choose **HOLO_DARK** from the **Theme** drop-down menu. This sets the icon to be white against a dark-colored (or black) background. Click **Next**.
7. Click **Finish** in the Confirm Icon Path dialog.

Tip: For a complete description for adding an icon, see Create app icons with Image Asset Studio.

3.2 Add an Activity

As you learned in a previous lesson, an `Activity` represents a single screen in your app in which your user can perform a single, focused task. You already have one activity, `MainActivity.java`. Now you add another activity called `OrderActivity.java`.

1. Right-click (or Control-click) the **com.example.android.droidcafe** folder in the left column and choose **New > Activity > Empty Activity**.
2. Edit the **Activity Name** to be **OrderActivity**, and the **Layout Name** to be **activity_order**. Leave the other options alone, and click **Finish**.

The `OrderActivity` class should now be listed along with `MainActivity` in the **java** folder, and `activity_order.xml` should now be listed in the **layout** folder. The Empty Activity template added these files.

3.3 Change the action

In this step you change the action for the `FloatingActionButton` to launch the new `Activity`.

1. Open **MainActivity**.
2. Change the `onClick(View view)` method to make an explicit intent to start `OrderActivity`:

```
public void onClick(View view) {

    Intent intent = new Intent(MainActivity.this, OrderActivity.class);

    startActivity(intent);

}
```

3. Run the app. Tap the floating action button that now uses the shopping cart icon. A blank `Activity` should appear (`OrderActivity`). Tap the Back button to go back to `MainActivity`.

Task 3 solution code

The solution code for this task is included in the code and layout for Android Studio project DroidCafe.

Coding challenge

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

Challenge: The DroidCafe app's `MainActivity` launches a second `Activity` called `OrderActivity`.

You learned in another lesson how to send data from an `Activity` to another `Activity`. Change the app to send the order message for the selected dessert in `MainActivity` to a new `TextView` at the top of the `OrderActivity` layout.

1. Add a `TextView` at the top of the `OrderActivity` layout with the id `order_textview`.
2. Create a member variable (`mOrderMessage`) in `MainActivity` for the order message that appears in the `Toast`.
3. Change the `showDonutOrder()`, `showIceCreamOrder()`, and `showFroyoOrder()` click handlers to assign the message string `mOrderMessage` before displaying the `Toast`. For example, the following assigns the `donut_order_message` string to `mOrderMessage` and displays the `Toast`:

```
mOrderMessage = getString(R.string.donut_order_message);
displayToast(mOrderMessage);
```

4. Add a `public static final String` called `EXTRA_MESSAGE` to the top of `MainActivity` to define the key for an `intent.putExtra`:

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

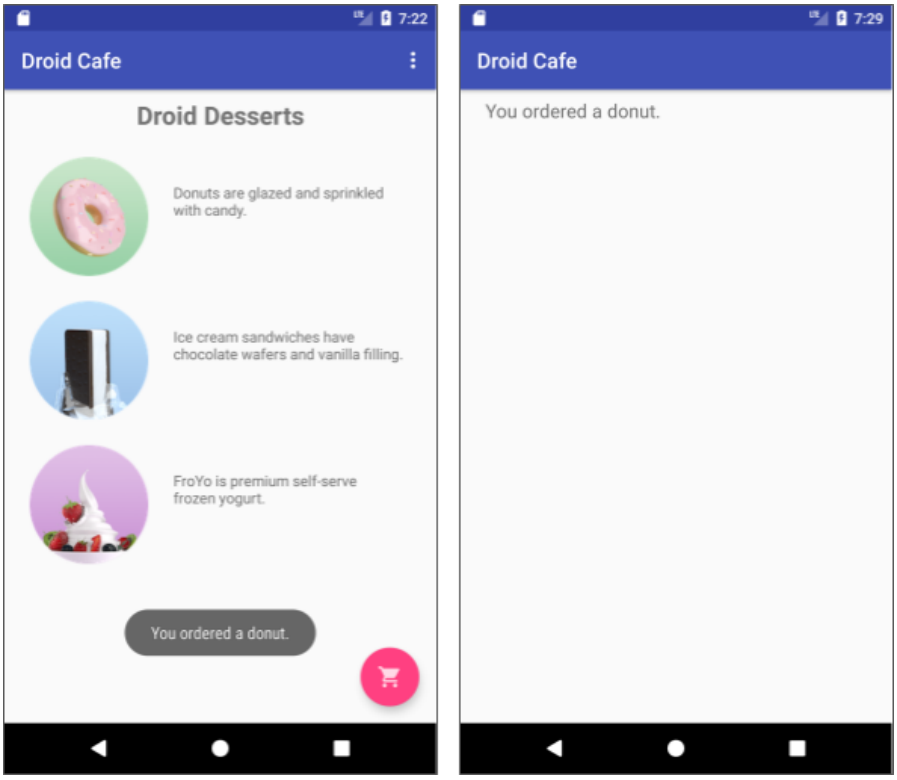
5. Change the `onClick()` method to include the `intent.putExtra` statement before launching `OrderActivity`:

```
public void onClick(View view) {
    Intent intent =
            new Intent(MainActivity.this, OrderActivity.class);
    intent.putExtra(EXTRA_MESSAGE, mOrderMessage);
    startActivity(intent);
}
```

6. In `OrderActivity`, add the following code to the `onCreate()` method to get the `Intent` that launched the `Activity`, extract the string message, and replace the text in the `TextView` with the message:

```
Intent intent = getIntent();
String message = "Order: " +
                  intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
TextView textView = findViewById(R.id.order_textview);
textView.setText(message);
```

7. Run the app. After choosing a dessert image, tap the floating action button to launch `OrderActivity`, which should include the order message as shown in the figure below.



Challenge solution code

Android Studio project: DroidCafeChallenge

Summary

- To use an image in a project, copy the image into the project's **drawable** folder (*project_name* **> app > src > main > res > drawable**).
- Define an `ImageView` to use it by dragging an `ImageView` to the layout and choosing the image for it.
- Add the `android:onClick` attribute to make an `ImageView` clickable like a button. Specify the name of the click handler.
- Create a click handler in the `Activity` to perform the action.
- Choose an icon: Expand **res** in the **Project > Android** pane, right-click (or Control-click) the **drawable** folder, and choose **New > Image Asset**. Choose **Action Bar and Tab Icons** in the drop-down menu, and click the clip art image (the Android logo next to **Clipart:**) to select a clip art image as the icon.
- Add another `Activity`: In the **Project > Android** pane, right-click (or Control-click) the package name folder within the **java** folder and choose **New > Activity** and a template for the `Activity` (such as **Empty Activity**).
- Display a `Toast` message:

```
Toast.makeText(getApplicationContext(), message,
                                          Toast.LENGTH_SHORT).show();
```

Related concept

The related concept documentation is in 4.1: Buttons and clickable images.

Learn more

Android Studio documentation:

- Android Studio User Guide
- Create app icons with Image Asset Studio

Android developer documentation:

- User interface & navigation
- Build a UI with Layout Editor
- Build a Responsive UI with `ConstraintLayout`
- Layouts
- `View`
- `Button`
- `ImageView`
- `TextView`
- Buttons
- Styles and themes

Other:

- Codelabs: Using `ConstraintLayout` to design your Android views

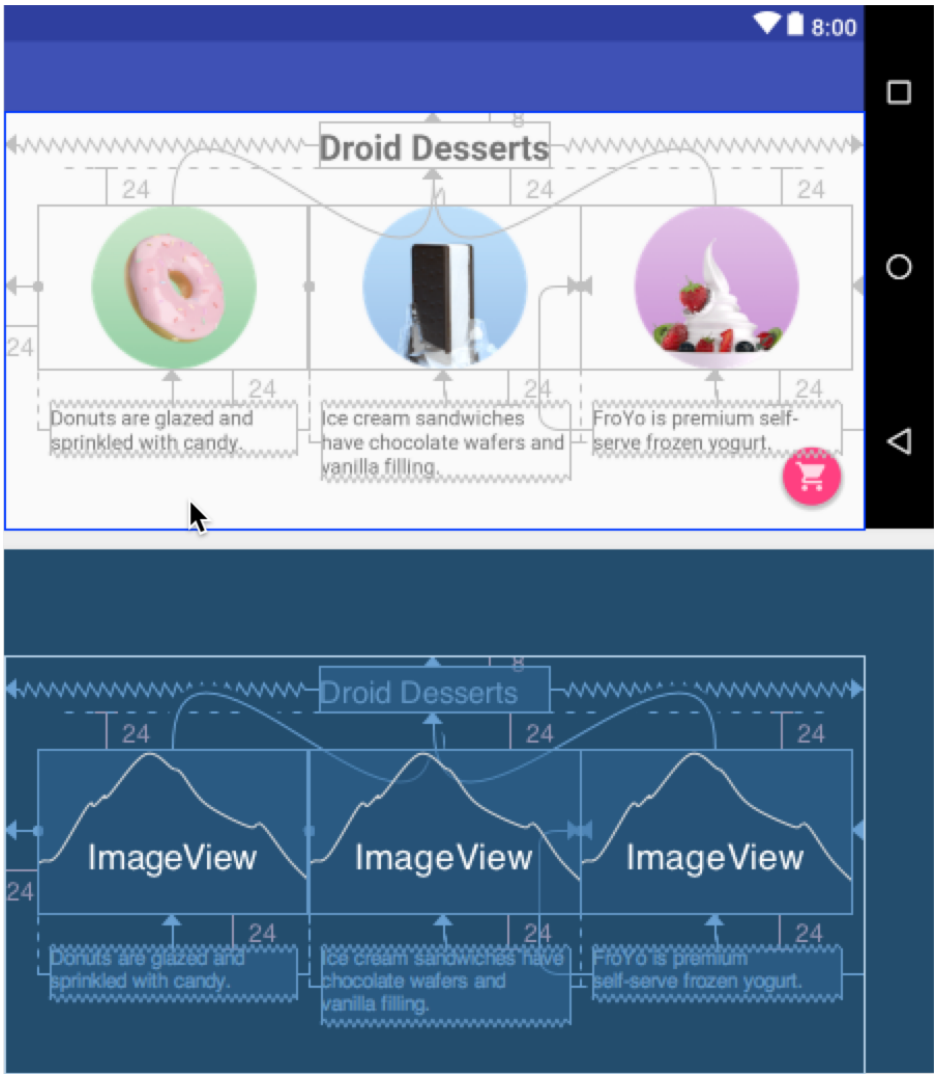
Homework

Change an app

The DroidCafe app looks fine when the device or emulator is oriented vertically. However, if you switch the device or emulator to horizontal orientation, the second and third images don't appear.

1. Open (or download) the DroidCafe app project.
2. Create a layout variant for horizontal orientation: `content_main.xml (land)`.
3. Remove constraints from the three images and three text descriptions.
4. Select all three images in the layout variant, and choose **Expand Horizontally** in the Pack button  to evenly distribute the images across the screen as shown in the figure below.
5. Constrain the text descriptions to the sides and bottoms of the images as shown in the

figure below.



Answer these questions

Question 1

How do you add images to an Android Studio project? Choose one:

- Drag each image to the layout editor.

- Copy the image files into your project's `drawable` folder.
- Drag an `ImageButton` to the layout editor.
- Choose **New > Image Asset** and then choose the image file.

Question 2

How do you make an `ImageView` clickable like a simple `Button`? Choose one:

- Add the `android:contentDescription` attribute to the `ImageView` in the layout and use it to call the click handler in the `Activity`.
- Add the `android:src` attribute to the `ImageView` in the layout and use it to call the click handler in the `Activity`.
- Add the `android:onClick` attribute to the `ImageView` in the layout and use it to call the click handler in the `Activity`.
- Add the `android:id` attribute to the `ImageView` in the layout and use it to call the click handler in the `Activity`.

Question 3

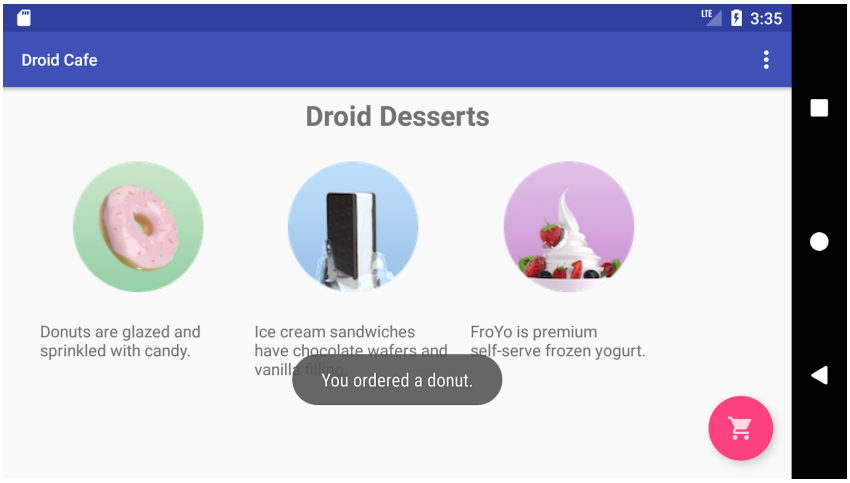
Which rule applies to a click handler called from the attribute in the layout? Choose one:

- The click handler method must include the event listener `View.OnClickListener`, which is an interface in the `View` class .
- The click handler method must be `public`, return `void`, and define a `View` as its only parameter.
- The click handler must customize the `View.OnClickListener` class and override its click handler to perform some action.
- The click handler method must be `private` and return a `View`.

Submit your app for grading

Guidance for graders

1. Run the app.
2. Switch to horizontal orientation to see the new layout variant. It should look like the figure below.



Lesson 4.2: Input controls

Introduction

To enable the user to enter text or numbers, you use an `EditText` element. Some input controls are `EditText` attributes that define the type of keyboard that appears, to make entering data easier for users. For example, you might choose `phone` for the `android:inputType` attribute to show a numeric keypad instead of an alphanumeric keyboard.

Other input controls make it easy for users to make choices. For example, `RadioButton` elements enable a user to select one (and only one) item from a set of items.

In this practical, you use attributes to control the on-screen keyboard appearance, and to set the type of data entry for an `EditText`. You also add radio buttons to the DroidCafe app so the user can select one item from a set of items.

What you should already know

You should be able to:

- Create an Android Studio project from a template and generate the main layout.
- Run apps on the emulator or a connected device.
- Create and edit UI elements using the layout editor and XML code.
- Access UI elements from your code using `findViewById()`.
- Convert the text in a `View` to a string using `getText()`.
- Create a click handler for a `Button` click.
- Display a `Toast` message.

What you'll learn

- How to change the input methods to enable suggestions, auto-capitalization, and password obfuscation.
- How to change the generic on-screen keyboard to a phone keypad or other specialized keyboards.
- How to add radio buttons for the user to select one item from a set of items.
- How to add a spinner to show a drop-down menu with values, from which the user can select one.

What you'll do

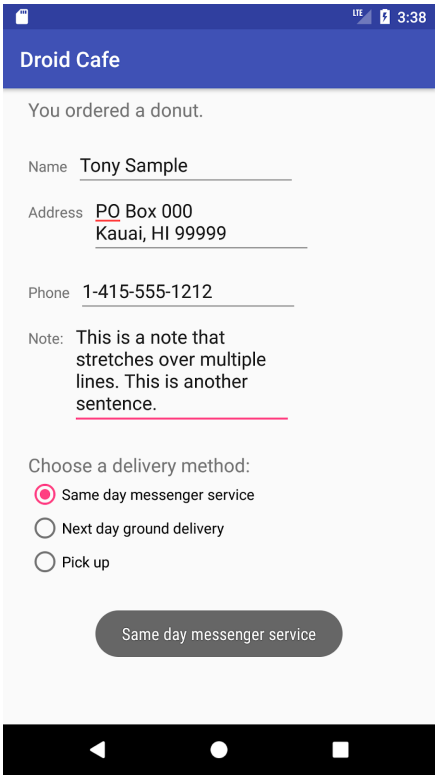
- Show a keyboard for entering an email address.
- Show a numeric keypad for entering phone numbers.
- Allow multiple-line text entry with automatic sentence capitalization.
- Add radio buttons for selecting an option.
- Set an `onClick` handler for the radio buttons.
- Add a spinner for the phone number field for selecting one value from a set of values.

App overview

In this practical, you add more features to the DroidCafe app from the lesson on using clickable images.

In the app's `OrderActivity` you experiment with the `android:inputType` attribute for `EditText` elements. You add `EditText` elements for a person's name and address, and use attributes to define single-line and multiple-line elements that make suggestions as you enter text. You also add an `EditText` that shows a numeric keypad for entering a phone number.

Other types of input controls include interactive elements that provide user choices. You add radio buttons to DroidCafe for choosing only one delivery option from several options. You also offer a spinner input control for selecting the label (**Home**, **Work**, **Other**, **Custom**) for the phone number.



Task 1: Experiment with text entry attributes

Touching an `EditText` editable text field places the cursor in the text field and automatically displays the on-screen keyboard so that the user can enter text.

An editable text field expects a certain type of text input, such as plain text, an email address, a phone number, or a password. It's important to specify the input type for each text field in your app so that the system displays the appropriate soft input method, such as an on-screen keyboard for plain text, or a numeric keypad for entering a phone number.

1.1 Add an EditText for entering a name

In this step you add a `TextView` and an `EditText` to the `OrderActivity` layout in the DroidCafe app so that the user can enter a person's name.

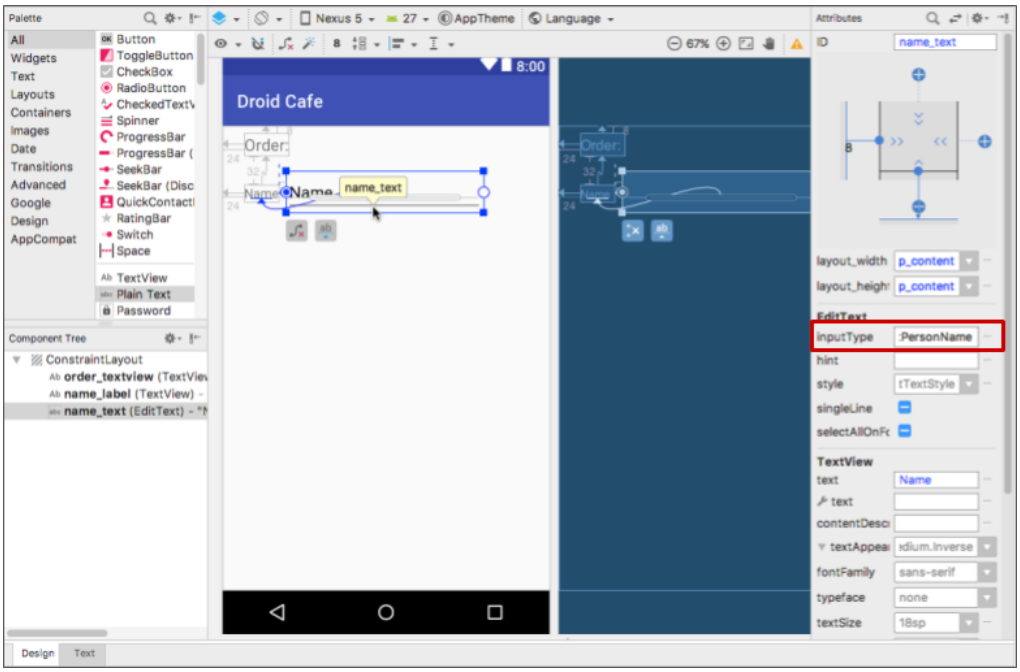
1. Make a copy of the DroidCafe app from the lesson on using clickable images, and rename the copy to **DroidCafeInput**. If you didn't complete the coding challenge in that lesson, download the DroidCafeChallenge project and rename it to **DroidCafeInput**.
2. Open the **activity_order.xml** layout file, which uses a `ConstraintLayout`.
3. Add a `TextView` to the `ConstraintLayout` in `activity_order.xml` under the `order_textview` element already in the layout. Use the following attributes for the new `TextView`:

TextView attribute	**Value**
`android:id`	`"@+id/name_label"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginStart`	`"24dp"`
`android:layout_marginLeft`	`"24dp"`

`android:layout_marginTop`	`"32dp"`
`android:text`	`"Name"`
`app:layout_constraintStart_toStartOf`	`"parent"`
`app:layout_constraintTop_toBottomOf`	`"@+id/order_textview"`

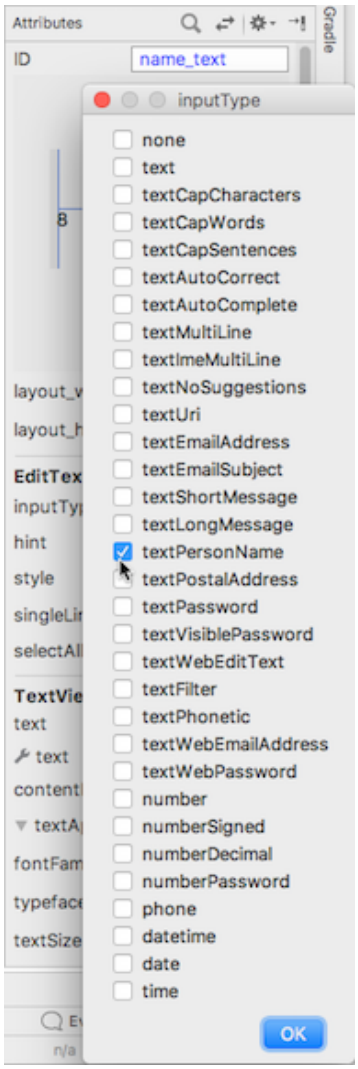
4. Extract the string resource for the `android:text` attribute value to create and entry for it called `name_label_text` in `strings.xml`.

5. Add an `EditText` element. To use the visual layout editor, drag a **Plain Text** element from the **Palette** pane to a position next to the `name_label` `TextView`. Then enter **name_text** for the **ID** field, and constrain the left side and baseline of the element to the `name_label` element right side and baseline as shown in the figure below:



6. The figure above highlights the **inputType** field in the **Attributes** pane to show that Android Studio automatically assigned the `textPersonName` type. Click the **inputType** field to see the

menu of input types:



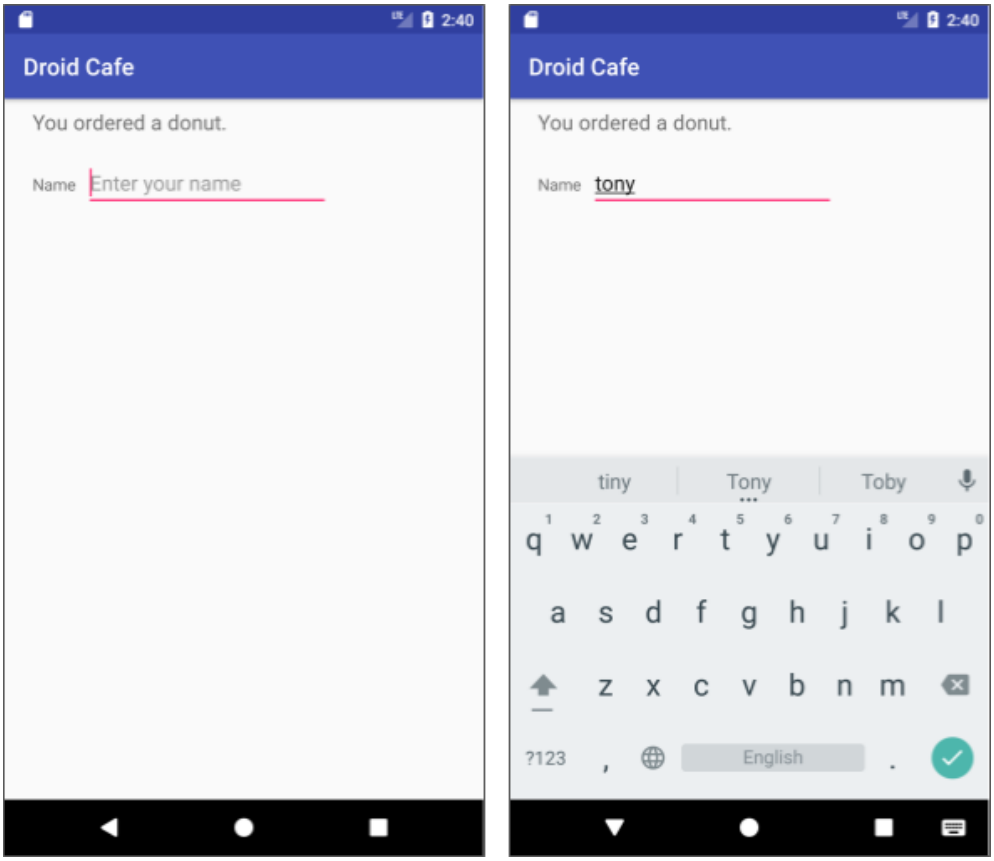
In the figure above, **textPersonName** is selected as the input type.

7. Add a hint for text entry, such as **Enter your name**, in the **hint** field in the **Attributes** pane, and delete the **Name** entry in the **text** field. As a hint to the user, the text "Enter your name" should be dimmed inside the `EditText`.
8. Check the XML code for the layout by clicking the **Text** tab. Extract the string resource for the `android:hint` attribute value to `enter_name_hint`. The following attributes should be set for the new `EditText` (add the `layout_marginLeft` attribute for compatibility with older versions of Android):

EditText attribute	**Value**
`android:id`	`"@+id/name_text"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginStart`	`8dp`
`android:layout_marginLeft`	`8dp`
`android:ems`	`"10"`
`android:hint`	`"@string/enter_name_hint"`
`android:inputType`	`"textPersonName"`
`app:layout_constraintBaseline_toBaselineOf`	`"@+id/name_label"`
`app:layout_constraintStart_toEndOf`	`"@+id/name_label"`

As you can see in the XML code, the `android:inputType` attribute is set to `textPersonName`.

9. Run the app. Tap the donut image on the first screen, and then tap the floating action button to see the next `Activity`. Tap inside the text entry field to show the keyboard and enter text, as shown in the figure below.



Note that suggestions automatically appear for words that you enter. Tap a suggestion to use it. This is one of the properties of the `textPersonName` value for the `android:inputType` attribute. The `inputType` attribute controls a variety of features, including keyboard layout, capitalization, and multiple lines of text.

10. To close the keyboard, tap the checkmark icon in a green circle  , which appears in the lower right corner of the keyboard. This is known as the **Done** key.

1.2 Add a multiple-line EditText

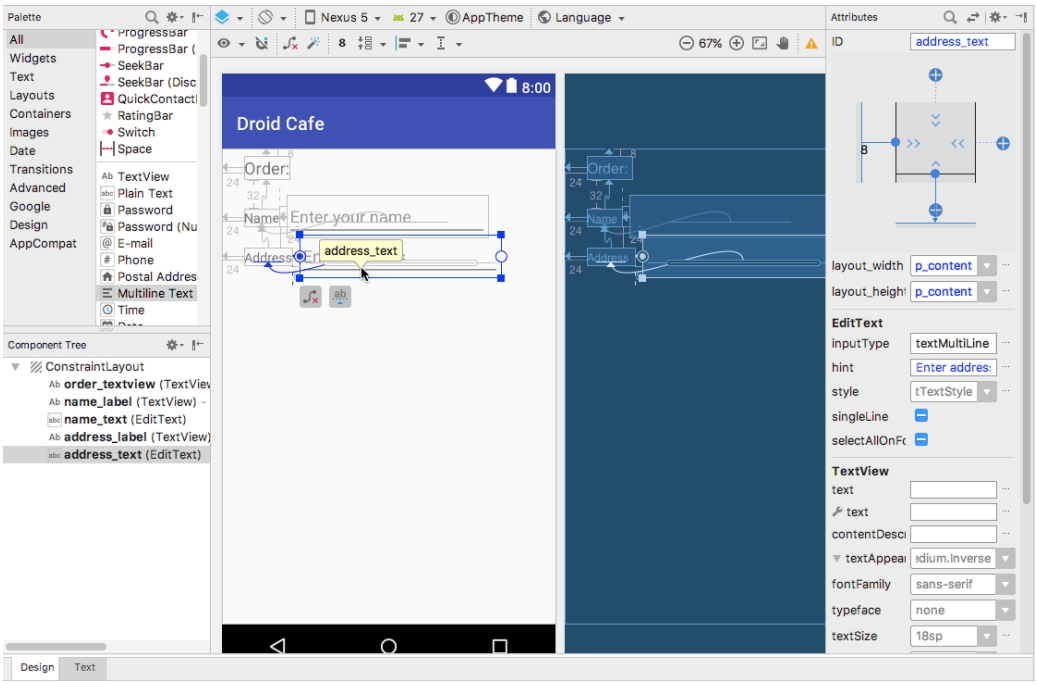
In this step you add another `EditText` to the `OrderActivity` layout in the DroidCafe app so that the user can enter an address using multiple lines.

1. Open the **activity_order.xml** layout file if it is not already open.
2. Add a `TextView` under the `name_label` element already in the layout. Use the following attributes for the new `TextView`:

TextView attribute	**Value**
`android:id`	`"@+id/address_label"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginStart`	`"24dp"`
`android:layout_marginLeft`	`"24dp"`
`android:layout_marginTop`	`"24dp"`
`android:text`	`"Address"`
`app:layout_constraintStart_toStartOf`	`"parent"`
`app:layout_constraintTop_toBottomOf`	`"@+id/name_label"`

3. Extract the string resource for the `android:text` attribute value to create and entry for it called `address_label_text` in `strings.xml`.
4. Add an `EditText` element. To use the visual layout editor, drag a **Multiline Text** element from the **Palette** pane to a position next to the `address_label` `TextView`. Then enter **address_text** for the **ID** field, and constrain the left side and baseline of the element to the

`address_label` element right side and baseline as shown in the figure below:

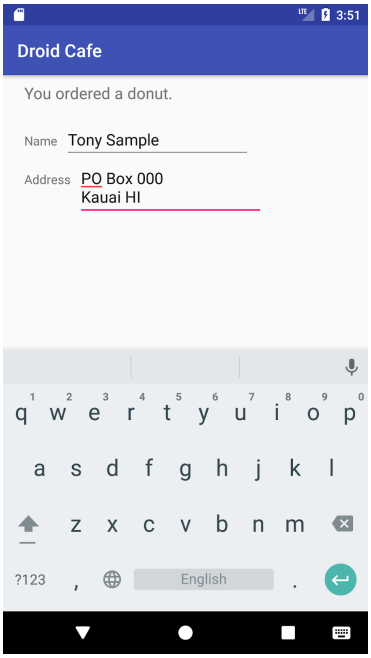


5. Add a hint for text entry, such as **Enter address**, in the **hint** field in the **Attributes** pane. As a hint to the user, the text "Enter address" should be dimmed inside the `EditText`.
6. Check the XML code for the layout by clicking the **Text** tab. Extract the string resource for the `android:hint` attribute value to `enter_address_hint`. The following attributes should be set for the new `EditText` (add the `layout_marginLeft` attribute for compatibility with older versions of Android):

EditText attribute	**Value**
`android:id`	`"@+id/address_text"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginStart`	`8dp`

`android:layout_marginLeft`	`8dp`
`android:ems`	`"10"`
`android:hint`	`"@string/enter_address_hint"`
`android:inputType`	`"textMultiLine"`
`app:layout_constraintBaseline_toBaselineOf`	`"@+id/address_label"`
`app:layout_constraintStart_toEndOf`	`"@+id/address_label"`

7. Run the app. Tap an image on the first screen, and then tap the floating action button to see the next `Activity`.
8. Tap inside the "Address" text entry field to show the keyboard and enter text, as shown in the figure below, using the Return key  in the lower right corner of the keyboard (also known as the Enter or New Line key) to start a new line of text. The Return key appears if you set the `textMultiLine` value for the `android:inputType` attribute.



9. To close the keyboard, tap the down-arrow button that appears instead of the Back button in the bottom row of buttons.

1.3 Use a keypad for phone numbers

In this step you add another `EditText` to the `OrderActivity` layout in the DroidCafe app so that the user can enter a phone number on a numeric keypad.

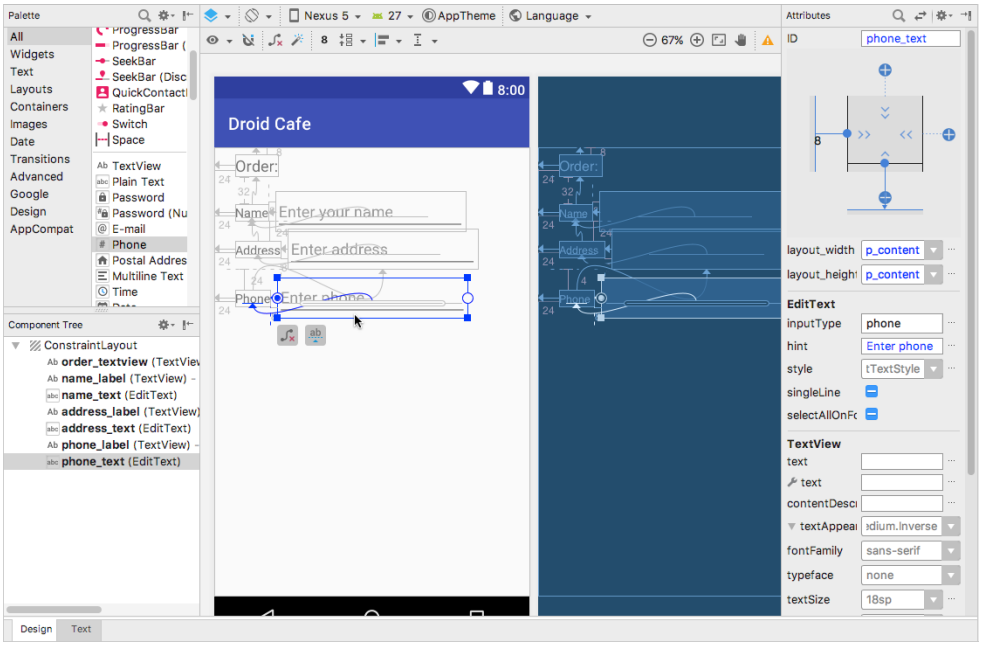
1. Open the **activity_order.xml** layout file if it is not already open.
2. Add a `TextView` under the `address_label` element already in the layout. Use the following attributes for the new `TextView`:

TextView attribute	**Value**
`android:id`	`"@+id/phone_label"`

`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginStart`	`"24dp"`
`android:layout_marginLeft`	`"24dp"`
`android:layout_marginTop`	`"24dp"`
`android:text`	`"Phone"`
`app:layout_constraintStart_toStartOf`	`"parent"`
`app:layout_constraintTop_toBottomOf`	`"@+id/address_text"`

Note that this `TextView` is constrained to the bottom of the multiple-line `EditText` (`address_text`). This is because `address_text` can grow to multiple lines, and this `TextView` should appear beneath it.

3. Extract the string resource for the `android:text` attribute value to create and entry for it called `phone_label_text` in `strings.xml`.
4. Add an `EditText` element. To use the visual layout editor, drag a **Phone** element from the **Palette** pane to a position next to the `phone_label` `TextView`. Then enter **phone_text** for the **ID** field, and constrain the left side and baseline of the element to the `phone_label` element right side and baseline as shown in the figure below:

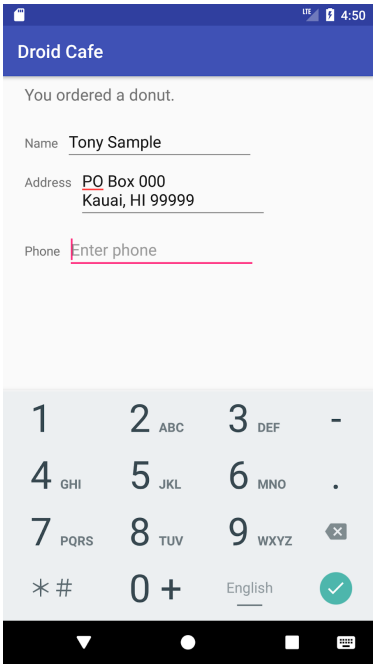


5. Add a hint for text entry, such as **Enter phone**, in the **hint** field in the **Attributes** pane. As a hint to the user, the text "Enter phone" should be dimmed inside the `EditText`.
6. Check the XML code for the layout by clicking the **Text** tab. Extract the string resource for the `android:hint` attribute value to `enter_phone_hint`. The following attributes should be set for the new `EditText` (add the `layout_marginLeft` attribute for compatibility with older versions of Android):

EditText attribute	**Value**
`android:id`	`"@+id/phone_text"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginStart`	`8dp`
`android:layout_marginLeft`	`8dp`

`android:ems`	`"10"`
`android:hint`	`"@string/enter_phone_hint"`
`android:inputType`	`"phone"`
`app:layout_constraintBaseline_toBaselineOf`	`"@+id/phone_label"`
`app:layout_constraintStart_toEndOf`	`"@+id/phone_label"`

7. Run the app. Tap an image on the first screen, and then tap the floating action button to see the next `Activity`.
8. Tap inside the "Phone" field to show the numeric keypad. You can then enter a phone number, as shown in the figure below.



9. To close the keyboard, tap the **Done** key  .

To experiment with `android:inputType` attribute values, change an `EditText` element's `android:inputType` values to the following to see the result:

- `textEmailAddress`: Tapping the field brings up the email keyboard with the **@** symbol located near the space key.
- `textPassword`: The characters the user enters turn into dots to conceal the entered password.

1.4 Combine input types in one EditText

You can combine `inputType` attribute values that don't conflict with each other. For example, you can combine the `textMultiLine` and `textCapSentences` attribute values for multiple lines of text in which each sentence starts with a capital letter.

1. Open the **activity_order.xml** layout file if it is not already open.
2. Add a `TextView` under the `phone_label` element already in the layout. Use the following attributes for the new `TextView`:

TextView attribute	**Value**
`android:id`	`"@+id/note_label"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginStart`	`"24dp"`
`android:layout_marginLeft`	`"24dp"`

`android:layout_marginTop`	`"24dp"`
`android:text`	`"Note"`
`app:layout_constraintStart_toStartOf`	`"parent"`
`app:layout_constraintTop_toBottomOf`	`"@+id/phone_label"`

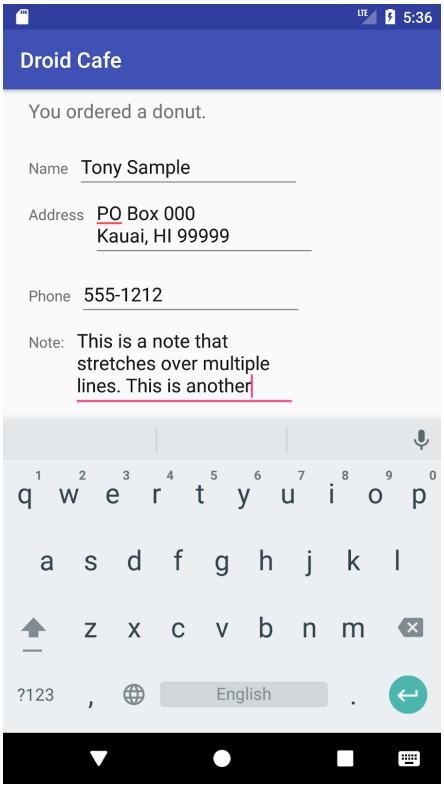
3. Extract the string resource for the `android:text` attribute value to create and entry for it called `note_label_text` in `strings.xml`.
4. Add an `EditText` element. To use the visual layout editor, drag a **Multiline Text** element from the **Palette** pane to a position next to the `note_label` `TextView`. Then enter **note_text** for the **ID** field, and constrain the left side and baseline of the element to the `note_label` element right side and baseline as you did previously with the other `EditText` elements.
5. Add a hint for text entry, such as **Enter note**, in the **hint** field in the **Attributes** pane.
6. Click inside the **inputType** field in the **Attributes** pane. The **textMultiLine** value is already selected. In addition, select **textCapSentences** to combine these attributes.
7. Check the XML code for the layout by clicking the **Text** tab. Extract the string resource for the `android:hint` attribute value to `enter_note_hint`. The following attributes should be set for the new `EditText` (add the `layout_marginLeft` attribute for compatibility with older versions of Android):

EditText attribute	**Value**
`android:id`	`"@+id/note_text"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginStart`	`8dp`
`android:layout_marginLeft`	`8dp`

`android:ems`	`"10"`
`android:hint`	`"@string/enter_note_hint"`
`android:inputType`	`"textCapSentences|textMultiLine"`
`app:layout_constraintBaseline_toBaselineOf`	`"@+id/note_label"`
`app:layout_constraintStart_toEndOf`	`"@+id/note_label"`

To combine values for the `android:inputType` attribute, concatenate them using the pipe (|) character.

8. Run the app. Tap an image on the first screen, and then tap the floating action button to see the next `Activity`.
9. Tap inside the "Note" field enter complete sentences, as shown in the figure below. Use the Return key to create a new line, or simply type to wrap sentences over multiple lines.



Task 2: Use radio buttons

Input controls are the interactive elements in your app's UI that accept data input. Radio buttons are input controls that are useful for selecting only one option from a set of options.

Tip: You should use radio buttons if you want the user to see all available options side-by-side. If it's not necessary to show all options side-by-side, you may want to use a `Spinner` instead, which

is described in another chapter.

In this task you add a group of radio buttons to the DroidCafeInput app for setting the delivery options for the dessert order. For an overview and more sample code for radio buttons, see Radio Buttons.

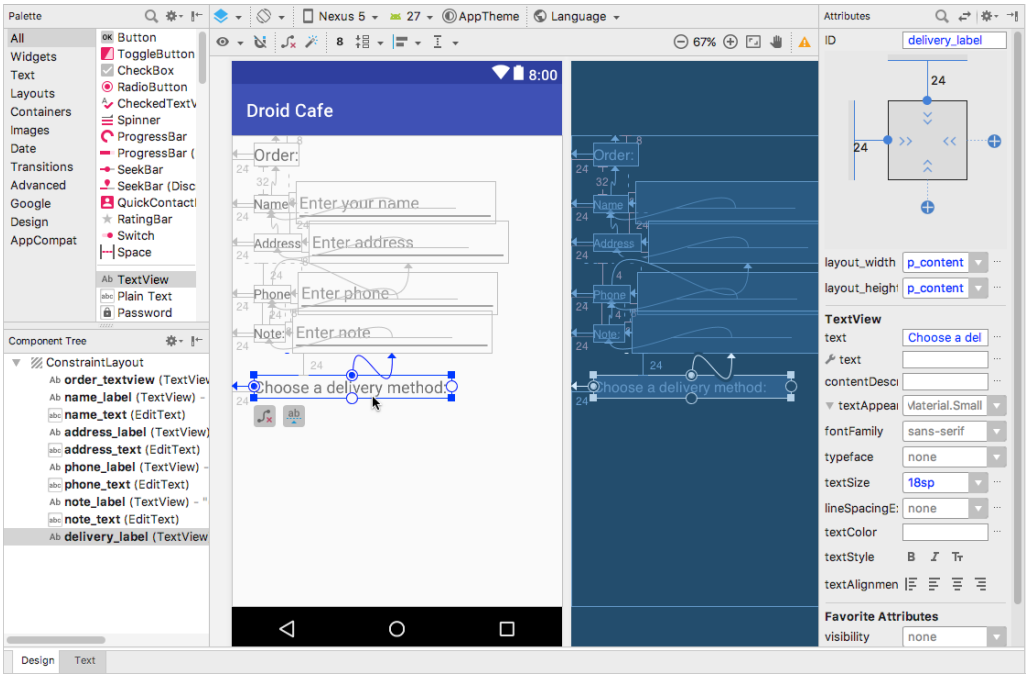
2.1 Add a RadioGroup and radio buttons

To add radio buttons to `OrderActivity` in the DroidCafeInput app, you create `RadioButton` elements in the `activity_order.xml` layout file. After editing the layout file, the layout for the radio buttons in `OrderActivity` will look something like the figure below.

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

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

1. Open **activity_order.xml** and add a `TextView` element constrained to the bottom of the `note_text` element already in the layout, and to the left margin, as shown in the figure below.



2. Switch to editing XML, and make sure that you have the following attributes set for the new `TextView`:

`TextView` attribute	**Value**
`android:id`	`"@+id/delivery_label"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginStart`	`"24dp"`
`android:layout_marginLeft`	`"24dp"`
`android:layout_marginTop`	`"24dp"`
`android:text`	`"Choose a delivery method: "`

`android:textSize`	`"18sp"`
`app:layout_constraintStart_toStartOf`	`"parent"`
`app:layout_constraintTop_toBottomOf`	`"@+id/note_text"`

3. Extract the string resource for `"Choose a delivery method:"` to be `choose_delivery_method`.

4. To add radio buttons, enclose them within a `RadioGroup`. Add the `RadioGroup` to the layout underneath the `TextView` you just added, enclosing three `RadioButton` elements as shown in the XML code below:

```
<RadioGroup
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="24dp"
        android:layout_marginStart="24dp"
        android:orientation="vertical"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/delivery_label">

        <RadioButton
            android:id="@+id/sameday"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="onRadioButtonClicked"
            android:text="Same day messenger service" />

        <RadioButton
            android:id="@+id/nextday"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="onRadioButtonClicked"
            android:text="Next day ground delivery" />

        <RadioButton
            android:id="@+id/pickup"
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="onRadioButtonClicked"
            android:text="Pick up" />
</RadioGroup>
```

The "onRadioButtonClicked" entry for the android:onClick attribute for each RadioButton will be underlined in red until you add that method in the next step of this task.

5. Extract the three string resources for the android:text attributes to the following names so that the strings can be translated easily: same_day_messenger_service, next_day_ground_delivery, and pick_up.

2.2 Add the radio button click handler

The android:onClick attribute for each radio button element specifies the onRadioButtonClicked() method to handle the click event. Therefore, you need to add a new onRadioButtonClicked() method in the OrderActivity class.

1. Open **activity_order.xml** (if it is not already open) and find one of the onRadioButtonClicked values for the android:onClick attribute that is underlined in red.
2. Click the onRadioButtonClicked value, and then click the red bulb warning icon in the left margin.
3. Choose **Create onRadioButtonClicked(View) in OrderActivity** in the red bulb's menu. Android Studio creates the onRadioButtonClicked(View view) method in OrderActivity:

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

In addition, the onRadioButtonClicked values for the other android:onClick attributes in activity_order.xml are resolved and no longer underlined.

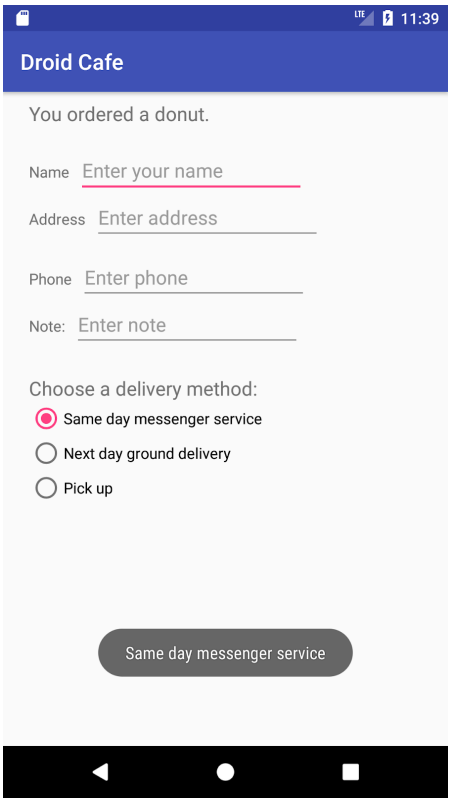
4. To display which radio button is clicked (that is, the type of delivery the user chooses), use a Toast message. Open **OrderActivity** and add the following displayToast method:

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

5. In the new `onRadioButtonClicked()` method, add a `switch case` block to check which radio button has been selected and to call `displayToast()` with the appropriate message. The code uses the `isChecked()` method of the `Checkable` interface, which returns `true` if the button is selected. It also uses the `View` `getId()` method to get the identifier for the selected radio button `view`:

```
public void onRadioButtonClicked(View view) {
        // Is the button now checked?
        boolean checked = ((RadioButton) view).isChecked();
        // Check which radio button was clicked.
        switch (view.getId()) {
            case R.id.sameday:
                if (checked)
                    // Same day service
                    displayToast(getString(R.string.same_day_messenger_service));
                break;
            case R.id.nextday:
                if (checked)
                    // Next day delivery
                    displayToast(getString(R.string.next_day_ground_delivery));
                break;
            case R.id.pickup:
                if (checked)
                    // Pick up
                    displayToast(getString(R.string.pick_up));
                break;
            default:
                // Do nothing.
                break;
        }
}
```

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



Task 2 solution code

Android Studio project: DroidCafeInput

Coding challenge

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

Challenge: The radio buttons for delivery choices in the DroidCafeInput app first appear unselected, which implies that there is no default delivery choice. Change the radio buttons so that one of them (such as `nextday`) is selected as the default when the radio buttons first appear.

Hint: You can accomplish this task entirely in the layout file. As an alternative, you can write code in `OrderActivity` to select one of the radio buttons when the `Activity` first appears.

Challenge solution code

Android Studio project: DroidCafeInput **(**see the second radio button in the `activity_order.xml` layout file)

Task 3: Use a spinner for user choices

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

Tip: For more information about spinners, see Spinners.

To provide a way to select a label for a phone number (such as **Home**, **Work**, **Mobile**, or **Other**), you can add a spinner to the `OrderActivity` layout in the DroidCafe app to appear right next to the phone number field.

3.1 Add a spinner to the layout

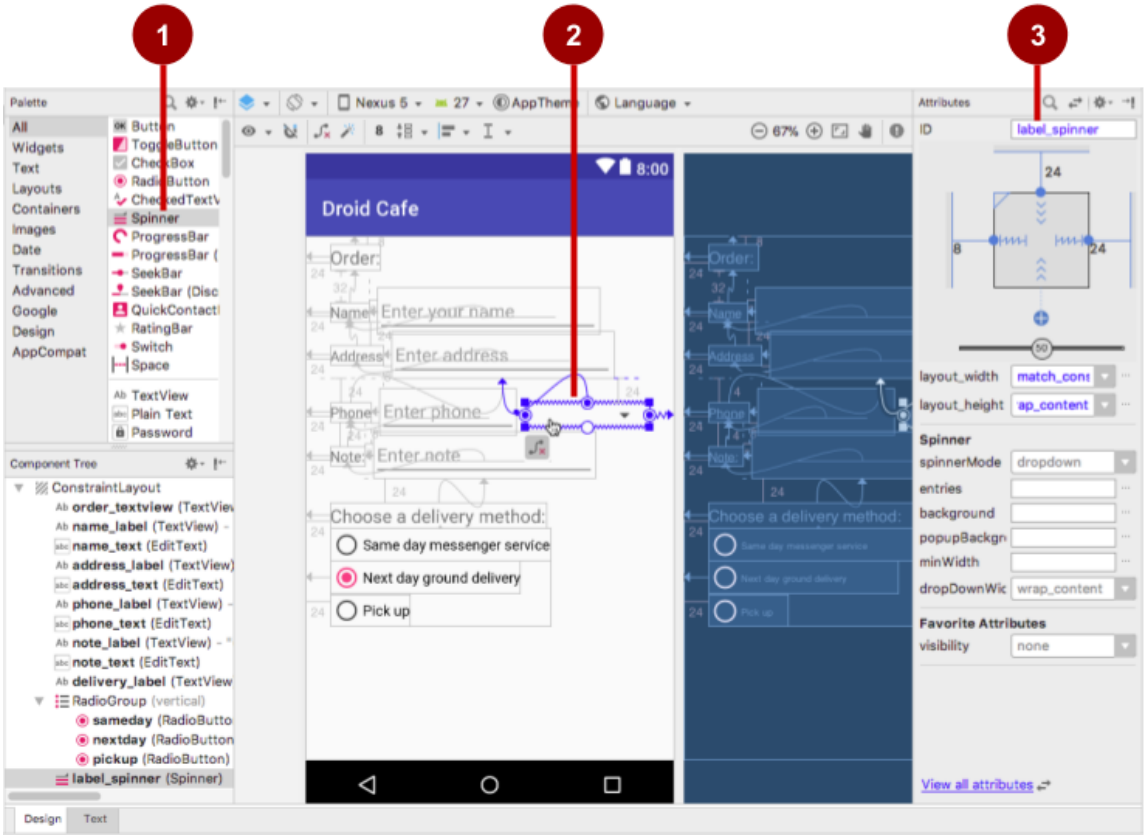
To add a spinner to the `OrderActivity` layout in the DroidCafe app, follow these steps, which are numbered in the figure below:

1. Open **activity_order.xml** and drag **Spinner** from the **Palette** pane to the layout.
2. Constrain the top of the `Spinner` element to the bottom of `address_text`, the right side to the right side of the layout, and the left side to `phone_text`.

   To align the `Spinner` and `phone_text` elements horizontally, use the pack button  in the toolbar, which provides options for packing or expanding selected UI elements.

   Select both the `Spinner` and `phone_text` elements in the **Component Tree**, click the pack button, and choose **Expand Horizontally**. As a result, both the `Spinner` and `phone_text` elements are set to fixed widths.
3. In the Attributes pane, set the `Spinner` **ID** to **label_spinner**, and set the top and right margins to **24**, and the left margin to **8**. Choose **match_constraint** for the **layout_width** drop-down menu, and **wrap_content** for the **layout_height** drop-down menu.

The layout should look like the figure below. The `phone_text` element's **layout_width** drop-down menu in the Attributes pane is set to `134dp`. You can optionally experiment with other width settings.



To look at the XML code for `activity_order.xml`, click the **Text** tab.

The `Spinner` should have the following attributes:

```
<Spinner
        android:id="@+id/label_spinner"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="24dp"
        android:layout_marginRight="24dp"
        android:layout_marginStart="8dp"
        android:layout_marginLeft="8dp"
        android:layout_marginTop="24dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/phone_text"
        app:layout_constraintTop_toBottomOf="@+id/address_text" />
```

Be sure to add the `android:layout_marginRight` and `android:layout_marginLeft` attributes shown in the code snippet above to maintain compatibility with older versions of Android.

The `phone_text` element should now have the following attributes (after using the pack tool):

```
<EditText
        android:id="@+id/phone_text"
        android:layout_width="134dp"
        android:layout_height="wrap_content"
        android:layout_marginLeft="8dp"
        android:layout_marginStart="8dp"
        android:ems="10"
        android:hint="@string/enter_phone_hint"
        android:inputType="phone"
        app:layout_constraintBaseline_toBaselineOf="@+id/phone_label"
        app:layout_constraintStart_toEndOf="@+id/phone_label" />
```

3.2 Add code to activate the Spinner and its listener

The choices for the `Spinner` are well-defined static strings such as “Home" and "Work," so you can use a text array defined in `strings.xml` to hold the values for it.

To activate the `Spinner` and its listener, implement the `AdapterView.OnItemSelectedListener` interface, which requires also adding the `onItemSelected()` and `onNothingSelected()` callback methods.

1. Open **strings.xml** and define the selectable values (**Home**, **Work**, **Mobile**, and **Other**) for the `Spinner` as the string array `labels_array`:

```
<string-array name="labels_array">
        <item>Home</item>
        <item>Work</item>
        <item>Mobile</item>
        <item>Other</item>
</string-array>
```

2. To define the selection callback for the `Spinner`, change your `OrderActivity` class to implement the `AdapterView.OnItemSelectedListener` interface as shown:

```
public class OrderActivity extends AppCompatActivity implements
            AdapterView.OnItemSelectedListener {
```

As you type **AdapterView.** in the statement above, Android Studio automatically imports the `AdapterView` widget. The reason why you need the `AdapterView` is because you need an adapter—specifically an `ArrayAdapter`—to assign the array to the `Spinner`. An *adapter* connects your data—in this case, the array of spinner items—to the `Spinner`. You learn more about this pattern of using an adapter to connect data in another practical. This line should appear in your block of import statements:

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

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

3. Click the light bulb and select **Implement methods**. The `onItemSelected()` and `onNothingSelected()` methods, which are required for `OnItemSelectedListener`, should be highlighted, and the "Insert @Override" option should be selected. Click **OK**.

   This step automatically adds empty `onItemSelected()` and `onNothingSelected()` callback methods to the bottom of the `OrderActivity` class. Both methods use the parameter `AdapterView<?>`. The `<?>` is a Java type wildcard, enabling the method to be flexible enough to accept any type of `AdapterView` as an argument.
4. Instantiate a `Spinner` in the `onCreate()` method using the `label_spinner` element in the layout, and set its listener (`spinner.setOnItemSelectedListener`) in the `onCreate()` method, as shown in the following code snippet:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   // ... Rest of onCreate code ...
   // Create the spinner.
   Spinner spinner = findViewById(R.id.label_spinner);
   if (spinner != null) {
            spinner.setOnItemSelectedListener(this);
   }
   // Create ArrayAdapter using the string array and default spinner layout.
```

5. Continuing to edit the `onCreate()` method, add a statement that creates the `ArrayAdapter` with the string array (`labels_array`) using the Android-supplied `Spinner` layout for each item (`layout.simple_spinner_item`):

```
// Create ArrayAdapter using the string array and default spinner layout.
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
```

```
                R.array.labels_array, android.R.layout.simple_spinner_item);
// Specify the layout to use when the list of choices appears.
```

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

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

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

3.3 Add code to respond to Spinner selections

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

In this step you fill in the code for the `onItemSelected()` method to retrieve the selected item in the `Spinner`, using `getItemAtPosition()`, and assign the item to the `spinnerLabel` variable:

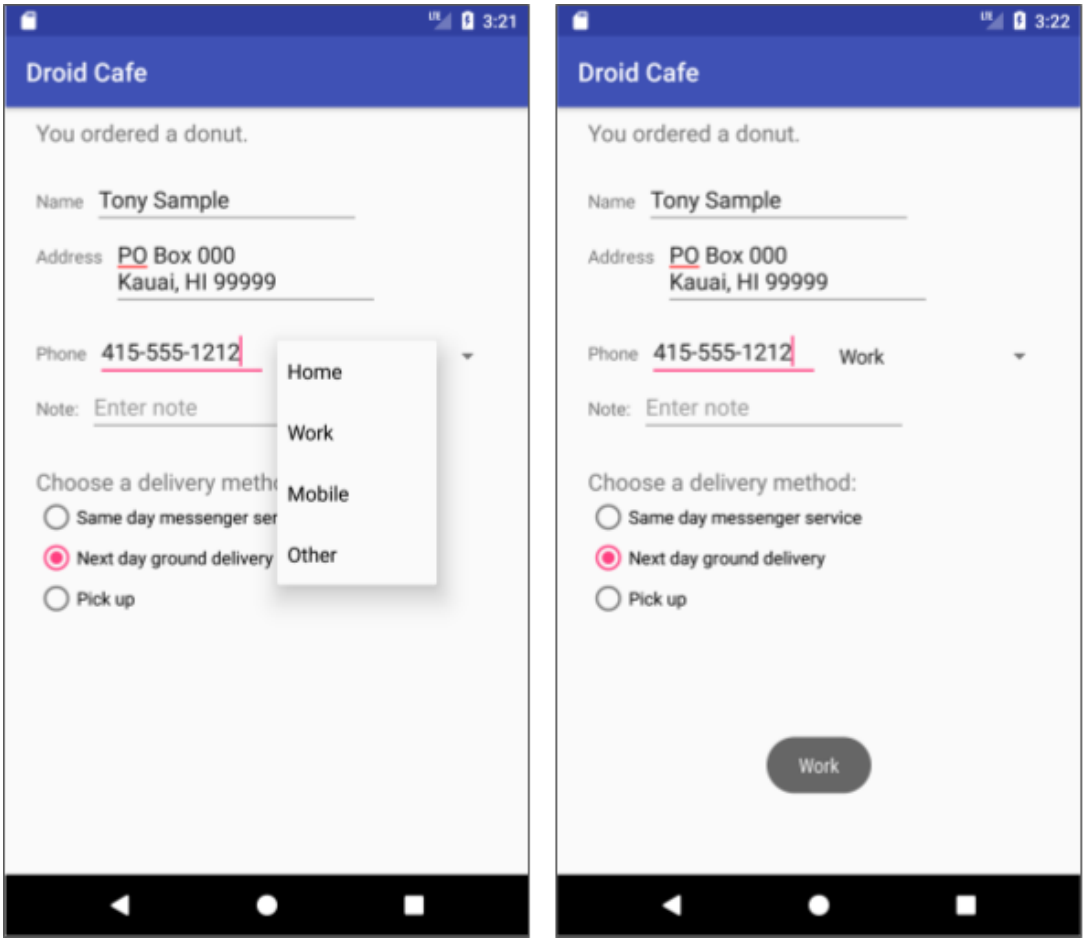
1. Add code to the empty `onItemSelected()` callback method, as shown below, to retrieve the user's selected item using `getItemAtPosition()`, and assign it to `spinnerLabel`. You can also add a call to the `displayToast()` method you already added to `OrderActivity`:

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

   There is no need to add code to the empty `onNothingSelected()` callback method for this example.

2. Run the app.

The `Spinner` appears next to the phone entry field and shows the first choice (**Home**). Tapping the `Spinner` reveals all the choices, as shown on the left side of the figure below. Tapping a choice in the `Spinner` shows a `Toast` message with the choice, as shown on the right side of the figure.



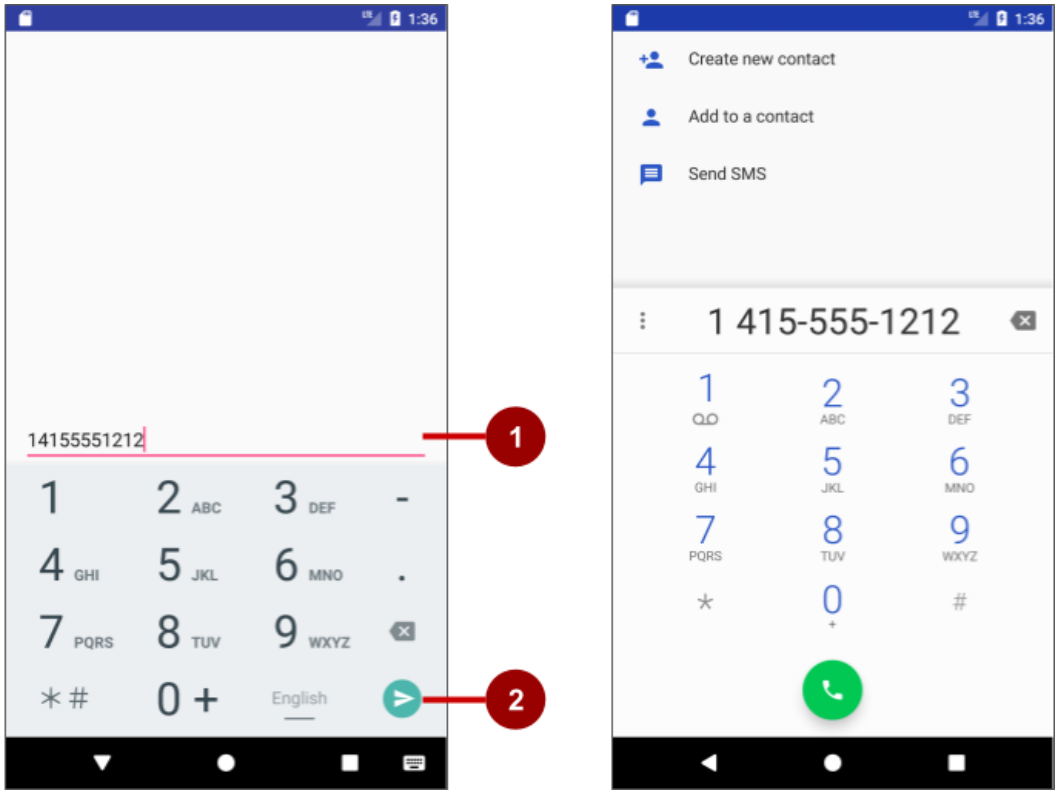
Task 3 solution code

Android Studio project: DroidCafeInput

Coding challenge 2

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

Challenge: Write code to perform an action directly from the keyboard by tapping a **Send** key, such as for dialing a phone number:



In the figure above:

1. Enter the phone number in the `EditText` field.
2. Tap the **Send** key to launch the phone dialer. The dialer appears on the right side of the figure.

For this challenge, create a new app project, and add an `EditText` that uses the `android:inputType` attribute set to `phone`. Use the `android:imeOptions` attribute for the `EditText` element with the `actionSend` value:

```
android:imeOptions="actionSend"
```

The user can now press the **Send** key to dial the phone number, as shown in the figure above.

In the `onCreate()` method for this `Activity`, you can use `setOnEditorActionListener()` to set the listener for the `EditText` to detect if the key is pressed:

```
EditText editText = findViewById(R.id.editText_main);
if (editText != null)
   editText.setOnEditorActionListener
                        (new TextView.OnEditorActionListener() {
      // If view is found, set the listener for editText.
   });
```

For help setting the listener, see Specify the input method type.

The next step is to override `onEditorAction()` and use the `IME_ACTION_SEND` constant in the `EditorInfo` class to respond to the pressed key. In the example below, the key is used to call the `dialNumber()` method to dial the phone number:

```
@Override
public boolean onEditorAction(TextView textView, int actionId, KeyEvent keyEvent) {
   boolean handled = false;
```

```
   if (actionId == EditorInfo.IME_ACTION_SEND) {
      dialNumber();
      handled = true;
   }
   return handled;
}
```

To finish the challenge, create the `dialNumber()` method, which uses an implicit intent with `ACTION_DIAL` to pass the phone number to another app that can dial the number. It should look like this:

```
private void dialNumber() {
   // Find the editText_main view.
   EditText editText = findViewById(R.id.editText_main);
   String phoneNum = null;
   // If the editText field is not null,
   // concatenate "tel: " with the phone number string.
   if (editText != null) phoneNum = "tel:" +
                                    editText.getText().toString();
   // Optional: Log the concatenated phone number for dialing.
   Log.d(TAG, "dialNumber: " + phoneNum);
   // Specify the intent.
   Intent intent = new Intent(Intent.ACTION_DIAL);
   // Set the data for the intent as the phone number.
   intent.setData(Uri.parse(phoneNum));
   // If the intent resolves to a package (app),
   // start the activity with the intent.
   if (intent.resolveActivity(getPackageManager()) != null) {
      startActivity(intent);
   } else {
       Log.d("ImplicitIntents", "Can't handle this!");
   }
}
```

Challenge 2 solution code

Android Studio project: KeyboardDialPhone

Summary

The following `android:inputType` attribute values affect the appearance of the on-screen keyboard:

- `textAutoCorrect`: Suggest spelling corrections.
- `textCapSentences`: Start each new sentence with a capital letter.
- `textPersonName`: Show a single line of text with suggestions as the user types, and the **Done** key for the user to tap when they're finished.
- `textMultiLine`: Enable multiple lines of text entry and a Return key to add a new line.
- `textPassword`: Hide a password when entering it.
- `textEmailAddress`: Show an email keyboard rather than a standard keyboard.
- `phone`: Show a phone keypad rather than a standard keyboard.

You set values for the `android:inputType` attribute in the XML layout file for an `EditText` element To combine values, concatenate them using the pipe (`|`) character.

Radio buttons are input controls that are useful for selecting only one option from a set of options:

- Group `RadioButton` elements together inside a `RadioGroup` so that only one `RadioButton` can be selected at a time.
- The order in which you list the `RadioButton` elements in the group determines the order that they appear on the screen.
- Use the `android:onClick` attribute for each `RadioButton` to specify the click handler.
- To find out if a button is selected, use the `isChecked()` method of the `Checkable` interface, which returns `true` if the button is selected.

A `Spinner` provides a drop-down menu:

- Add a `Spinner` to the layout.
- Use an `ArrayAdapter` to assign an array of text values as the `Spinner` menu items.

- Implement the `AdapterView.OnItemSelectedListener` interface, which requires also adding the `onItemSelected()` and `onNothingSelected()` callback methods to activate the `Spinner` and its listener.
- Use the `onItemSelected()` callback method to retrieve the selected item in the `Spinner` menu using `getItemAtPosition()`.

Related concept

The related concept documentation is in 4.2: Input controls.

Learn more

Android Studio documentation:

- Android Studio User Guide

Android developer documentation:

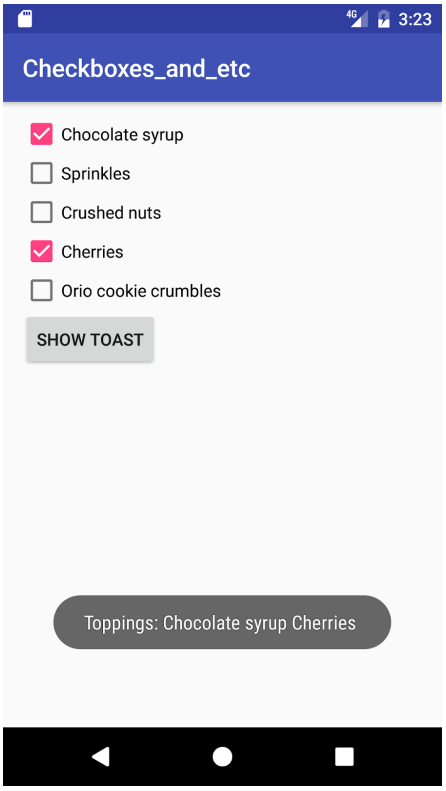
- Input events overview
- Specify the input method type
- Styles and themes
- Radio Buttons
- Spinners
- `View`
- `Button`
- `EditText`
- `android:inputType`
- `TextView`
- `RadioGroup`
- `Checkbox`
- `SeekBar`
- `ToggleButton`

- `Spinner`

Homework

Build and run an app

1. Create an app with five checkboxes and a **Show Toast** button, as shown below.
2. If the user selects a single checkbox and taps **Show Toast**, display a `Toast` message showing the checkbox that was selected.
3. If the user selects more than one checkbox and then taps **Show Toast**, display a `Toast` that includes the messages for all selected checkboxes, as shown in the figure below.



Answer these questions

Question 1

What's the most important difference between checkboxes and a `RadioGroup` of radio buttons? Choose one:

- The only difference is in how they appear: checkboxes show a checkmark when selected, while circular "radio" buttons appear filled when selected.
- `CheckBox` elements in the layout can use the `android:onClick` attribute to call a handler when selected.
- The major difference is that checkboxes enable multiple selections, while a `RadioGroup` allows only one selection.

Question 2

Which layout group lets you align a set of `CheckBox` elements vertically? Choose one:

- `RelativeLayout`
- `LinearLayout`
- `ScrollView`

Question 3

Which of the following is the method of the `Checkable` interface to check the state of a radio button (that is, whether it has been selected or not)?

- `getId()`
- `isChecked()`
- `onRadioButtonClicked()`
- `onClick()`

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- The layout includes five `CheckBox` views vertically aligned on the screen, and a **Show Toast** button.
- The `onSubmit()` method determines which checkbox is selected by using `findViewById()` with `isChecked()`.
- The strings describing toppings are concatenated into a `Toast` message.

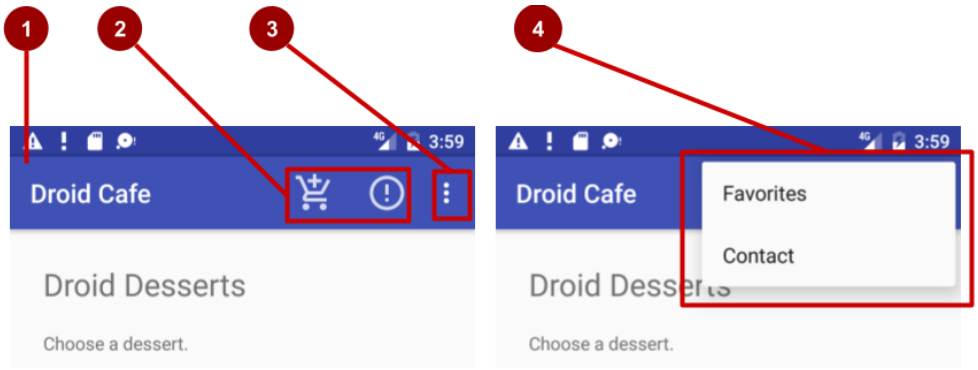
Lesson 4.3: Menus and pickers

Introduction

The *app bar* (also called the *action bar*) is a dedicated space at the top of each activity screen. When you create an `Activity` from the Basic Activity template, Android Studio includes an app bar.

The *options menu* in the app bar usually provides choices for navigation, such as navigation to another activity in the app. The menu might also provide choices that affect the use of the app itself, for example ways to change settings or profile information, which usually happens in a separate activity.

In this practical you learn about setting up the app bar and options menu in your app, as shown in the figure below.



In the figure above:

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

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

Tip: To provide a familiar and consistent user experience, use the Menu APIs to present user actions and other options in your activities. See Menus for details.

You also create an app that shows a dialog to request a user's choice, such as an alert that requires users to tap **OK** or **Cancel**. A *dialog* is a window that appears on top of the display or fills the display, interrupting the flow of activity. Android provides ready-to-use dialogs, called *pickers*, for picking a time or a date. You can use them to ensure that your users pick a valid time or date that is formatted correctly and adjusted to the user's local time and date. In this lesson you'll also create an app with the date picker.

What you should already know

You should be able to:

- Create and run apps in Android Studio.
- Create and edit UI elements using the layout editor.
- Edit XML layout code, and access elements from your Java code.
- Add a click handler to a `Button`.

What you'll learn

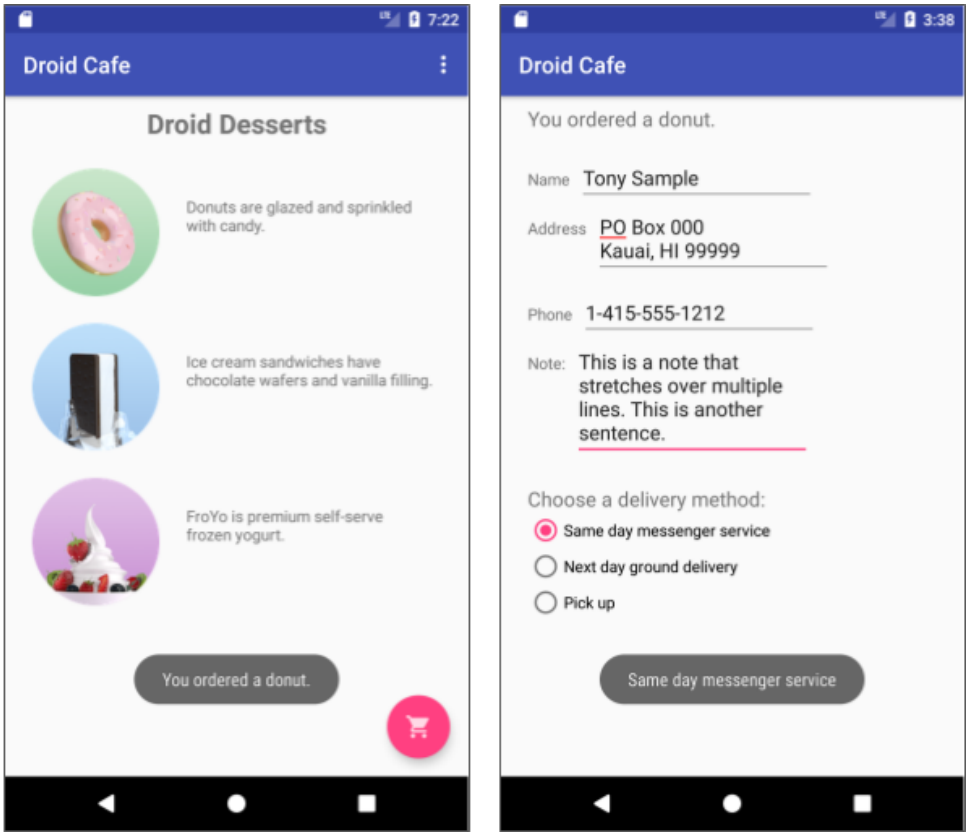
- How to add menu items to the options menu.
- How to add icons for items in the options menu.
- How to set menu items to show in the app bar.
- How to add click handlers for menu items.
- How to add a dialog for an alert.
- How to add the date picker.

What you'll do

- Continue adding features to the Droid Cafe project from the previous practical.
- Add menu items to the options menu.
- Add icons for menu items to appear in the app bar.
- Connect menu-item clicks to event handlers that process the click events.
- Use an alert dialog to request a user's choice.
- Use a date picker for date input.

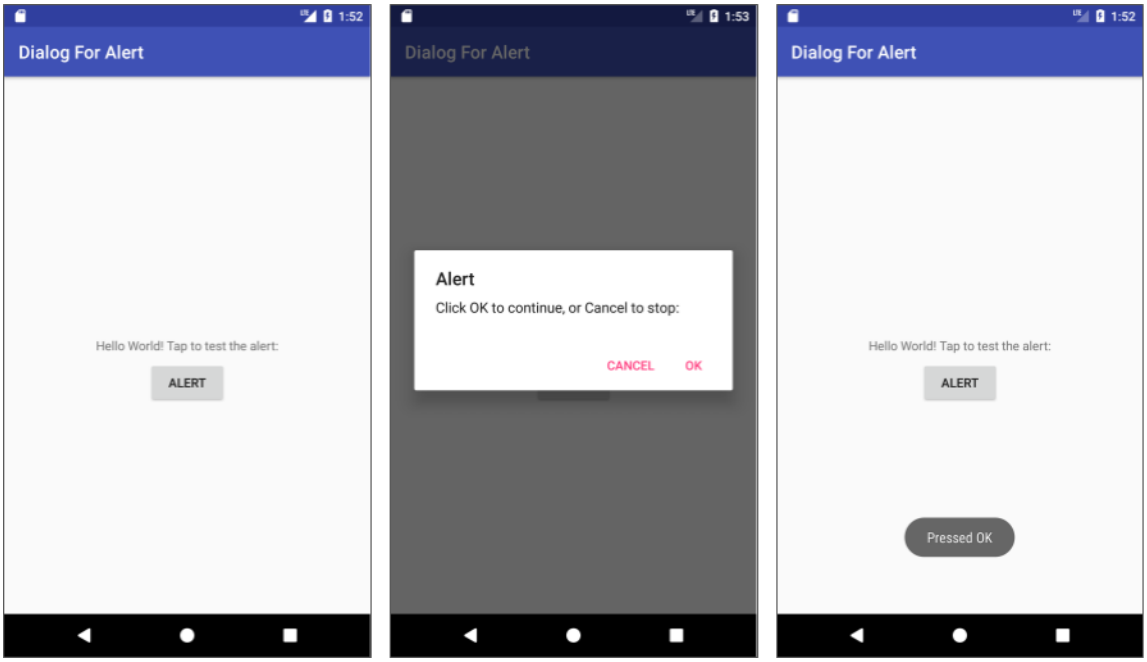
App overview

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

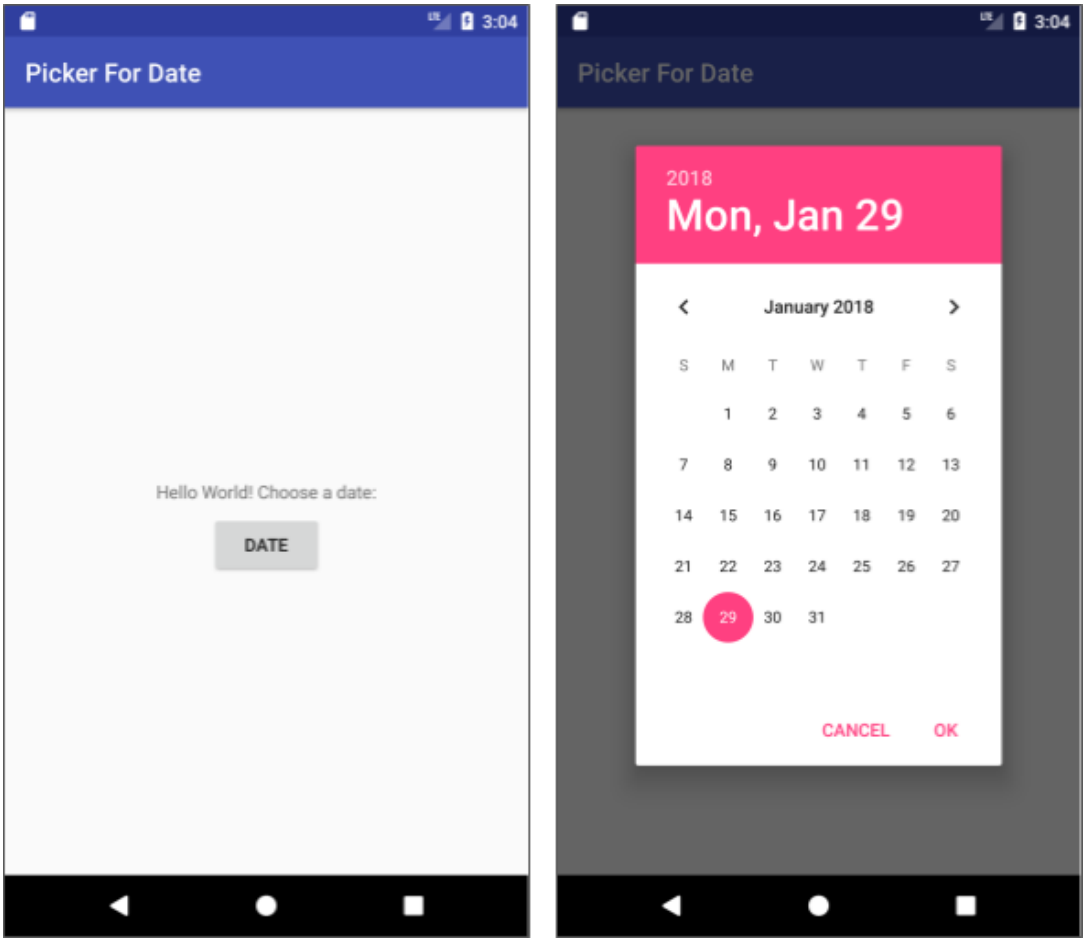


For this exercise you are using the v7 appcompat support library's `Toolbar` as an app bar, which works on the widest range of devices and also gives you room to customize your app bar later on as your app develops. To read more about design considerations for using the app bar, see Responsive layout grid in the Material Design specification.

You create a new app that displays an alert dialog. The dialog interrupts the user's workflow and requires the user to make a choice.



You also create an app that provides a `Button` to show the date picker, and converts the chosen date to a string to show in a `Toast` message.



Task 1: Add items to the options menu

In this task you open the DroidCafeInput project from the previous practical and add menu items to the options menu in the app bar at the top of the screen.

1.1 Examine the code

Open the DroidCafeInput app from the practical on using input controls and examine the following layout files in the **res > layout** folder:

- **activity_main.xml**: The main layout for `MainActivity`, the first screen the user sees.
- **content_main.xml**: The layout for the content of the `MainActivity` screen, which (as you will see shortly) is *included* within **activity_main.xml**.
- **activity_order.xml**: The layout for `OrderActivity`, which you added in the practical on using input controls.

Follow these steps:

1. Open **content_main.xml** and click the **Text** tab to see the XML code. The `app:layout_behavior` for the `ConstraintLayout` is set to `@string/appbar_scrolling_view_behavior`, which controls how the screen scrolls in relation to the app bar at the top. (This string resource is defined in a generated file called `values.xml`, which you should not edit.)

   For more about scrolling behavior, see Android Design Support Library in the Android Developers Blog. For design practices involving scrolling menus, see Scrolling in the Material Design specification.

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

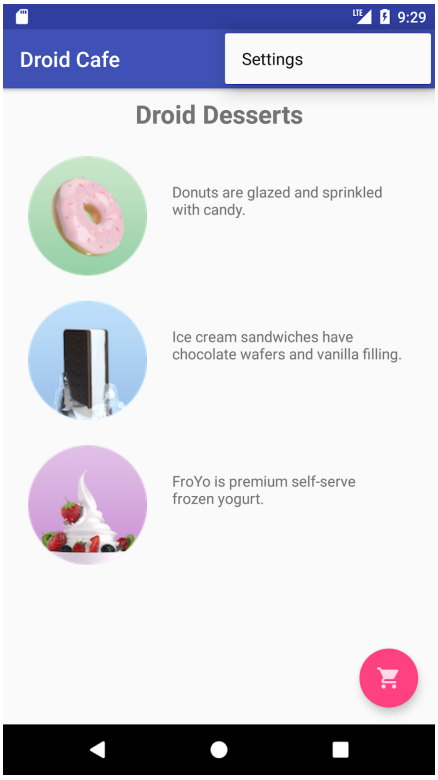
   `AppBarLayout` is like a vertical `LinearLayout`. It uses the `Toolbar` class in the support library, instead of the native `ActionBar`, to implement an app bar. The `Toolbar` within this layout has the id `toolbar`, and is also specified, like the `AppBarLayout`, with a fully qualified name (`android.support.v7.widget`).

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

   The **activity_main.xml** layout also uses an `include layout` statement to include the entire layout defined in **content_main.xml**. This separation of layout definitions makes it easier to

change the layout's *content* apart from the layout's toolbar definition and coordinator layout. This is a best practice for separating your content (which may need to be translated) from the format of your layout.

3. Run the app. Notice the bar at the top of the screen showing the name of the app (Droid Cafe). It also shows the *action overflow* button (three vertical dots) on the right side. Tap the overflow button to see the options menu, which at this point has only one menu option, **Settings**.



4. Examine the **AndroidManifest.xml** file. The `.MainActivity` activity is set to use the `NoActionBar` theme. This theme is defined in the `styles.xml` file (open **app > res >values > styles.xml** to see it). Styles are covered in another lesson, but you can see that the `NoActionBar` theme sets the `windowActionBar` attribute to `false` (no window app bar), and the `windowNoTitle` attribute to `true` (no title). These values are set because you are defining the app bar with `AppBarLayout`, rather than using an `ActionBar`. Using one of the `NoActionBar` themes prevents the app from using the native `ActionBar` class to provide the app bar.

5. Look at **MainActivity**, which extends `AppCompatActivity` and starts with the `onCreate()`

method, which sets the content view to the `activity_main.xml` layout and sets `toolbar` to be the `Toolbar` defined in the layout. It then calls `setSupportActionBar()` and passes `toolbar` to it, setting the `toolbar` as the app bar for the `Activity`.

For best practices about adding the app bar to your app, see Add the app bar.

1.2 Add more menu items to the options menu

You will add the following menu items to the options menu:

- **Order**: Navigate to `OrderActivity` to see the dessert order.
- **Status**: Check the status of an order.
- **Favorites**: Show favorite desserts.
- **Contact**: Contact the cafe. Because you don't need the existing **Settings** item, you will change **Settings** to **Contact**.

Android provides a standard XML format to define menu items. Instead of building a menu in your `Activity` code, you can define a menu and all of its menu items in an XML menu resource. You can then inflate the menu resource (load it as a `Menu` object) in your `Activity`:

1. Expand **res > menu** in the **Project > Android** pane, and open **menu_main.xml**. The only menu item provided from the template is `action_settings` (the **Settings** choice), which is defined as:

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

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

Attribute	**Value**
`android:id`	`"@+id/action_contact"`

`android:title`	`"Contact"`
`app:showAsAction`	`"never"`

3. Extract the hard-coded string `"Contact"` into the string resource `action_contact`.
4. Add a new menu item using the `<item>` tag within the `<menu>` block, and give the item the following attributes:

Attribute	**Value**
`android:id`	`"@+id/action_order"`
`android:orderInCategory`	`“10”`
`android:title`	`"Order"`
`app:showAsAction`	`"never"`

The `android:orderInCategory` attribute specifies the order in which the menu items appear in the menu, with the lowest number appearing higher in the menu. The **Contact** item is set to 100, which is a big number in order to specify that it shows up at the bottom rather than the top. You set the **Order** item to 10, which puts it above **Contact**, and leaves plenty of room in the menu for more items.

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

Status item attribute	**Value**
`android:id`	`"@+id/action_status"`
`android:orderInCategory`	`“20”`

`android:title`	`"Status"`
`app:showAsAction`	`"never"`

Favorites item attribute	**Value**
`android:id`	`"@+id/action_favorites"`
`android:orderInCategory`	`"30"`
`android:title`	`"Favorites"`
`app:showAsAction`	`"never"`

7. Extract `"Status"` into the resource `action_status`, and `"Favorites"` into the resource `action_favorites`.

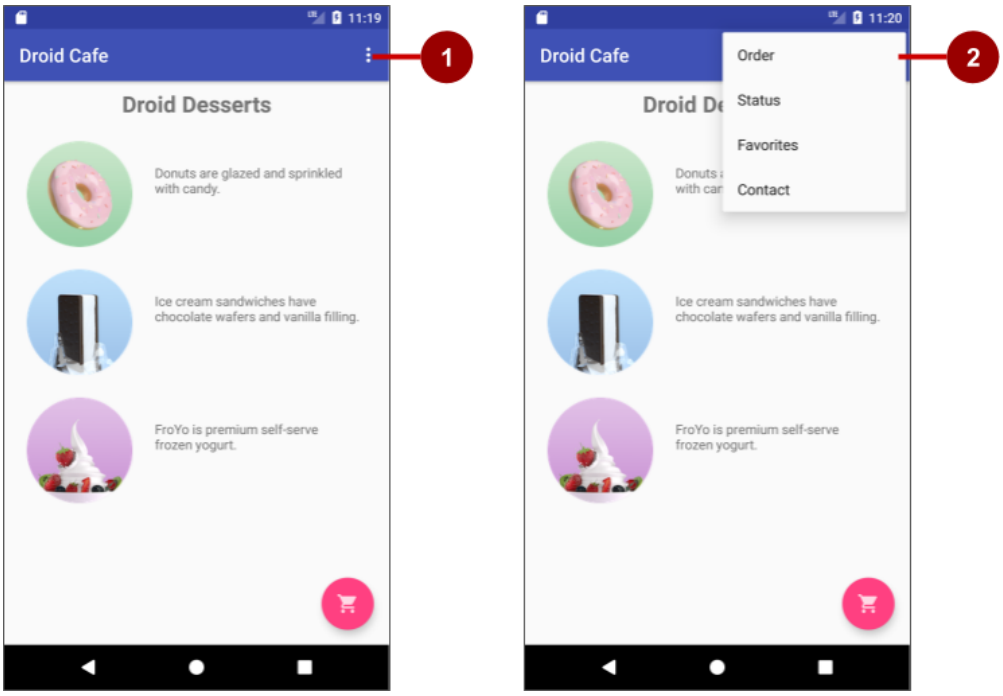
8. You will display a `Toast` message with an action message depending on which menu item the user selects. Open **strings.xml** and add the following string names and values for these messages:

```
<string name="action_order_message">You selected Order.</string>
<string name="action_status_message">You selected Status.</string>
<string name="action_favorites_message">You selected Favorites.</string>
<string name="action_contact_message">You selected Contact.</string>
```

9. Open **MainActivity**, and change the `if` statement in the `onOptionsItemSelected()` method replacing the id `action_settings` with the new id `action_order`:

```
if (id == R.id.action_order)
```

Run the app, and tap the action overflow icon, shown on the left side of the figure below, to see the options menu, shown on the right side of the figure below. You will soon add callbacks to respond to items selected from this menu.



In the figure above:

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

Notice the order of items in the options menu. You used the `android:orderInCategory` attribute to specify the priority of the menu items in the menu: The **Order** item is 10, followed by **Status** (20) and **Favorites** (30), and **Contact** is last (100). The following table shows the priority of items in the menu:

Menu item	**orderInCategory attribute**

Order	`10`
Status	`20`
Favorites	`30`
Contact	`100`

Task 2: Add icons for menu items

Whenever possible, you want to show the most frequently used actions using icons in the app bar so the user can click them without having to first click the overflow icon. In this task, you add icons for some of the menu items, and show some of menu items in the app bar at the top of the screen as icons.

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

- **Order** and **Status** should always be shown in the app bar.
- **Favorites** should be shown in the app bar if it will fit; if not, it should appear in the overflow menu.
- **Contact** should not appear in the app bar; it should *only* appear in the overflow menu.

2.1 Add icons for menu items

To specify icons for actions, you need to first add the icons as image assets to the **drawable** folder using the same procedure you used in the practical on using clickable images. You want to use the following icons (or similar ones):

-  **Order**: Use the same icon you added for the floating action button in the practical on using clickable images (ic_shopping_cart.png).

-  **Status**:
-  **Favorites**:
- **Contact**: No need for an icon because it will appear only in the overflow menu.

For the **Status** and **Favorites** icons, follow these steps:

1. Expand **res** in the **Project > Android** pane, and right-click (or Control-click) the **drawable** folder.
2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.
3. Choose **Action Bar and Tab Items** in the drop-down menu.
4. Change **ic_action_name** to another name (such as **ic_status_info** for the **Status** icon).
5. Click the clip art image (the Android logo next to **Clipart:**) to select a clip art image as the icon. A page of icons appears. Click the icon you want to use.
6. Choose **HOLO_DARK** from the **Theme** drop-down menu. This sets the icon to be white against a dark-colored (or black) background. Click **Next** and then click **Finish**.

Tip: See Create app icons with Image Asset Studio for a complete description.

2.2 Show the menu items as icons in the app bar

To show menu items as icons in the app bar, use the `app:showAsAction` attribute in `menu_main.xml`. The following values for the attribute specify whether or not the action should appear in the app bar as an icon:

- `"always"`: Always appears in the app bar. (If there isn't enough room it may overlap with other menu icons.)
- `"ifRoom"`: Appears in the app bar if there is room.
- `"never"`: Never appears in the app bar; its text appears in the overflow menu.

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

1. Open **menu_main.xml** again, and add the following attributes to the **Order**, **Status**, and **Favorites** items so that the first two (**Order** and **Status**) always appear, and the **Favorites** item appears only if there is room for it:

Order item attribute	**Old value**	**New value**
`android:icon`	*none*	`"@drawable/ic_shopping_cart"`
`app:showAsAction`	`"never"`	`“always”`

Status item attribute	**Old value**	**New value**
`android:icon`	*none*	`"@drawable/@drawable/ic_status_info"`
`app:showAsAction`	`"never"`	`"always"`

Favorites item attribute	**Old value**	**New value**
`android:icon`	*none*	`"@drawable/ic_favorite"`
`app:showAsAction`	`"never"`	`"ifRoom"`

2. Run the app. You should now see at least two icons in the app bar: the icon for **Order** and the icon for **Status** as shown on the left side of the figure below. (The **Favorites** and **Contact** options appear in the overflow menu.)

3. Rotate your device to the horizontal orientation, or if you’re running in the emulator, click the **Rotate Left** or **Rotate Right** icons to rotate the display into the horizontal orientation. You

should then see all three icons in the app bar for **Order**, **Status**, and **Favorites** as shown on the right side of the figure below.

How many action buttons will fit in the app bar? It depends on the orientation and the size of the device screen. Fewer buttons appear in a vertical orientation, as shown on the left side of the figure above, compared to a horizontal orientation as shown on the right side of the figure above. Action buttons may not occupy more than half of the main app bar width.

Task 3: Handle the selected menu item

In this task, you add a method to display a message about which menu item is tapped, and use the `onOptionsItemSelected()` method to determine which menu item was tapped.

3.1 Create a method to display the menu choice

1. Open **MainActivity**.
2. If you haven't already added the following method (in another lesson) for displaying a `Toast` message, add it now. You will use it as the action to take for each menu choice. (Normally you would implement an action for each menu item such as starting another `Activity`, as shown later in this lesson.)

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

3.2 Use the onOptionsItemSelected event handler

The `onOptionsItemSelected()` method handles selections from the options menu. You will add a `switch case` block to determine which menu item was selected and what action to take.

1. Find the `onOptionsItemSelected()` method provided by the template. The method determines whether a certain menu item was clicked, using the menu item's `id`. In the example below, the `id` is `action_order`:

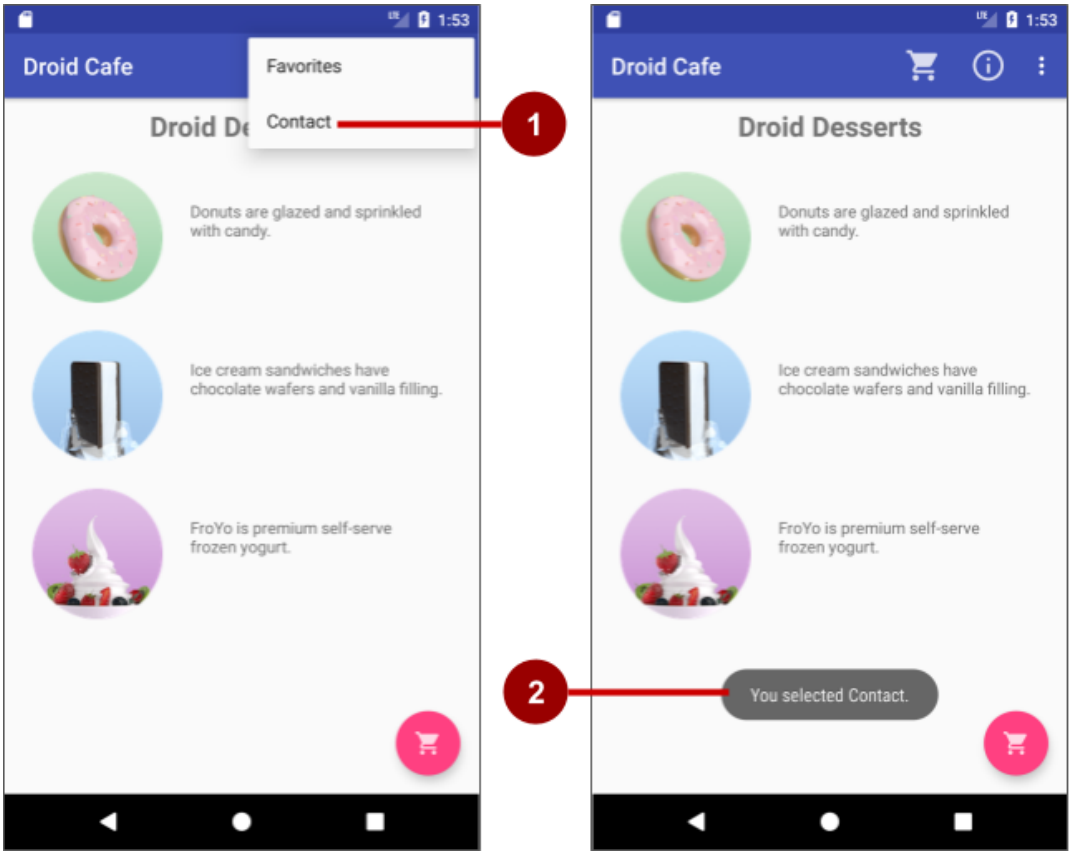
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.action_order) {
            return true;
        }
        return super.onOptionsItemSelected(item);
}
```

1. Replace the `int id` assignment statement and the `if` statement with the following `switch case` block, which sets the appropriate `message` based on the menu item's `id`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.action_order:
                displayToast(getString(R.string.action_order_message));
                return true;
            case R.id.action_status:
                displayToast(getString(R.string.action_status_message));
                return true;
            case R.id.action_favorites:
                displayToast(getString(R.string.action_favorites_message));
                return true;
            case R.id.action_contact:
```

```
                displayToast(getString(R.string.action_contact_message));
                return true;
            default:
                // Do nothing
        }
        return super.onOptionsItemSelected(item);
}
```

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



In the figure above:

1. Selecting the **Contact** item in the options menu.

2. The `Toast` message that appears.

3.3 Start an Activity from a menu item

Normally you would implement an action for each menu item, such as starting another `Activity`. Referring the snippet from the previous task, change the code for the `action_order` case to the following, which start `OrderActivity` (using the same code you used for the floating action button in the lesson on using clickable images):

```
switch (item.getItemId()) {
    case R.id.action_order:
        Intent intent = new Intent(MainActivity.this, OrderActivity.class);
        intent.putExtra(EXTRA_MESSAGE, mOrderMessage);
        startActivity(intent);
        return true;
    // ... code for other cases
}
```

Run the app. Clicking the shopping cart icon in the app bar (the **Order** item) takes you directly to the `OrderActivity` screen.

Task 3 solution code

Android Studio project: DroidCafeOptions

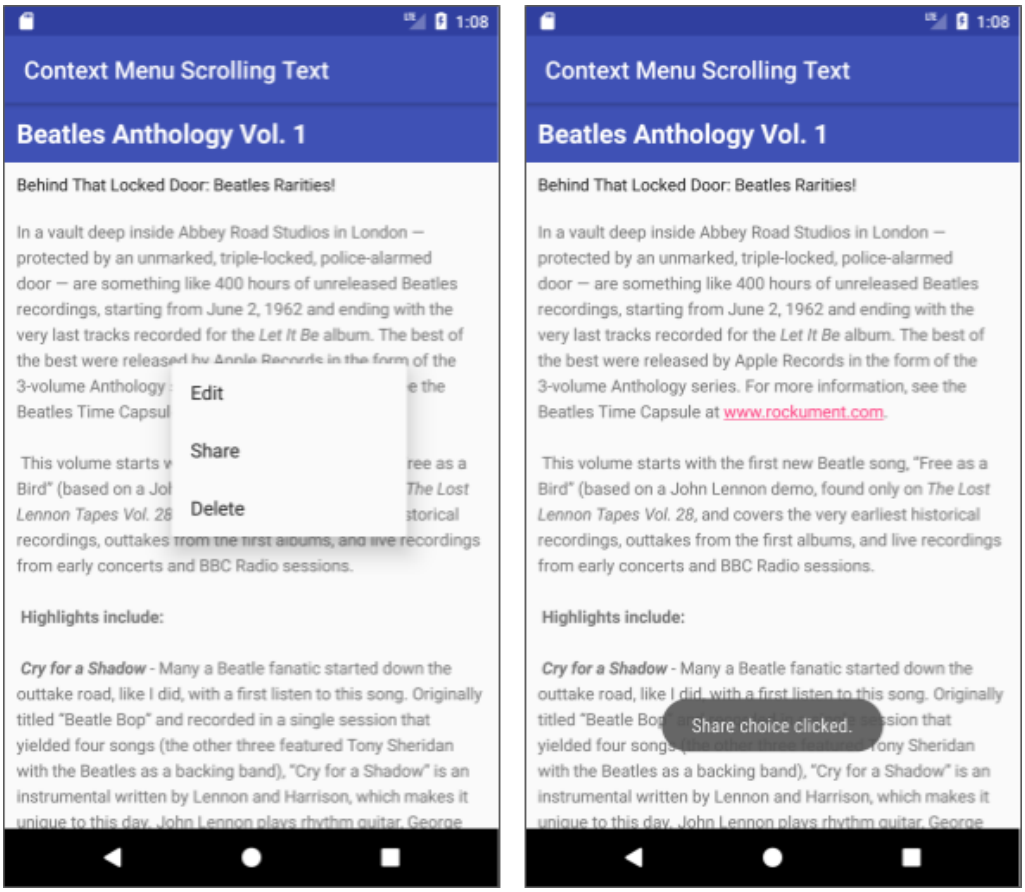
Coding challenge

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

Challenge: A *context menu* allows users to take an action on a selected `View`. While the options menu in the app bar usually provides choices for navigation to *another* activity, you use a context menu to allow the user to modify a `View` *within* the current activity.

Both menus are described in XML, both are inflated using `MenuInflater`, and both use an "on item selected" method—in this case, `onContextItemSelected()`. So the techniques for building and using the two menus are similar.

A context menu appears as a floating list of menu items when the user performs a touch & hold on a `View`, as shown in the left side of the figure below. For this challenge, add a context menu to the ScrollingText app to show three options: **Edit**, **Share**, and **Delete**, as shown in the figure below. The menu appears when the user performs a touch & hold on the `TextView`. The app then displays a `Toast` message showing the menu option chosen, as shown in the right side of the figure.



Hints

A context menu is similar to the options menu, with two critical differences:

- The context menu must be registered to a `View` so that the menu inflates when a touch & hold occurs on the `View`.
- Although the options menu code is supplied by the Basic Activity template, for a context menu you have to add the code and menu resource yourself.

To solve this challenge, follow these general steps:

1. Create an XML menu resource file for the menu items.

   Right-click the **res** folder and choose **New > Android Resource Directory**. Choose **menu** in the **Resource type** drop-down menu and click **OK**. Then right-click the new **menu** folder, choose **New > Menu resource file**, enter the name **menu_context**, and click **OK**. Open **menu_context** and enter the menu items as you did for an options menu.

2. Register the `View` to the context menu using the `registerForContextMenu()` method.

   In the `onCreate()` method, register the `TextView`:

```
TextView article_text = findViewById(R.id.article);
registerForContextMenu(article_text);
```

3. Implement the `onCreateContextMenu()` method in the `Activity` to inflate the menu.

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                    ContextMenu.ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_context, menu);
}
```

4. Implement the `onContextItemSelected()` method in the `Activity` to handle menu-item clicks. In this case, simply display a `Toast` with the menu choice.

```
@Override
public boolean onContextItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.context_edit:
                displayToast("Edit choice clicked.");
                return true;
            case R.id.context_share:
                displayToast("Share choice clicked.");
                return true;
            case R.id.context_delete:
                displayToast("Delete choice clicked.");
                return true;
            default:
                return super.onContextItemSelected(item);
        }
}
```

5. Run the app. If you tap and drag, the text scrolls as before. However, if you do a long tap, the contextual menu appears.

Challenge solution code

Android Studio project: ContextMenuScrollingText

Task 4: Use a dialog to request a user's choice

You can provide a dialog to request a user's choice, such as an alert that requires users to tap **OK** or **Cancel**. A *dialog* is a window that appears on top of the display or fills the display, interrupting the flow of activity.

For example, an alert dialog might require the user to click **Continue** after reading it, or give the user a choice to agree with an action by clicking a positive button (such as **OK** or **Accept**), or to disagree by clicking a negative button (such as **Cancel**). Use the `AlertDialog` subclass of the `Dialog` class to show a standard dialog for an alert.

Tip: Use dialogs sparingly, because they interrupt the user's workflow. For best design practices, see the Dialogs Material Design guide. For code examples, see Dialogs in the Android developer documentation.

In this practical, you use a `Button` to trigger a standard alert dialog. In a real-world app, you might trigger an alert dialog based on some condition, or based on the user tapping something.

4.1 Create a new app to show an alert dialog

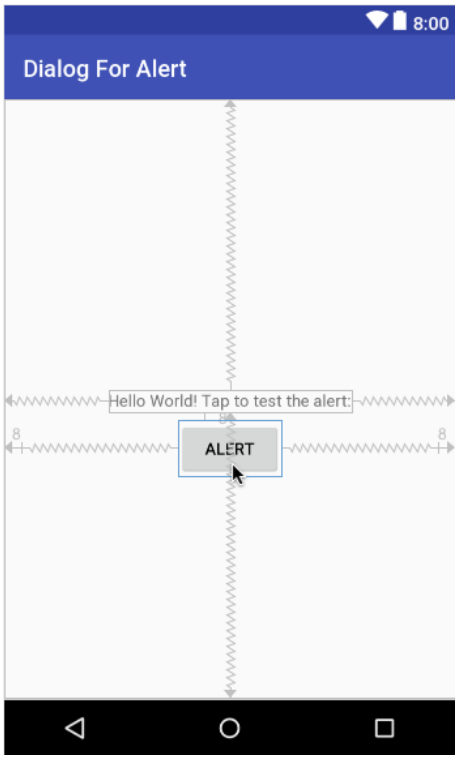
In this exercise, you build an alert with **OK** and **Cancel** buttons. The alert is triggered by the user tapping a button.

1. Create a new project called **Dialog For Alert** based on the Empty Activity template.
2. Open the **activity_main.xml** layout file to show the layout editor.
3. Edit the `TextView` element to say **Hello World! Tap to test the alert:** instead of "Hello World!"

4. Add a `Button` under the `TextView`. (Optional: Constrain the button to the bottom of the `TextView` and the sides of the layout, with margins set to 8dp.)
5. Set the text of the `Button` to **Alert**.
6. Switch to the **Text** tab, and extract the text strings for the `TextView` and `Button` to string resources.
7. Add the `android:onClick` attribute to the button to call the click handler `onClickShowAlert()`. After you enter it, the click handler is underlined in red because it has not yet been created.

```
android:onClick="onClickShowAlert"
```

You now have a layout similar to the following:



4.2 Add an alert dialog to the main activity

The *builder* design pattern makes it easy to create an object from a class that has a lot of required and optional attributes and would therefore require a lot of parameters to build. Without this pattern, you would have to create constructors for combinations of required and optional attributes; with this pattern, the code is easier to read and maintain. For more information about the builder design pattern, see Builder pattern.

The builder class is usually a static member class of the class it builds. Use `AlertDialog.Builder` to build a standard alert dialog, with `setTitle()` to set its title, `setMessage()` to set its message, and `setPositiveButton()` and `setNegativeButton()` to set its buttons.

To make the alert, you need to make an object of `AlertDialog.Builder`. You will add the `onClickShowAlert()` click handler for the **Alert** `Button`, which makes this object as its first order of business. That means that the dialog will be created only when the user clicks the **Alert** `Button`. While this coding pattern is logical for using a `Button` to test an alert, for other apps you may want to create the dialog in the `onCreate()` method so that it is always available for other code to trigger it.

1. Open **MainActivity** and add the beginning of the `onClickShowAlert()` method:

```
public void onClickShowAlert(View view) {

    AlertDialog.Builder myAlertBuilder = new

                                AlertDialog.Builder(MainActivity.this);

    // Set the dialog title and message.

}
```

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

2. Add the code to set the title and the message for the alert dialog to `onClickShowAlert()` after the comment:

```
// Set the dialog title and message.

myAlertBuilder.setTitle("Alert");

myAlertBuilder.setMessage("Click OK to continue, or Cancel to stop:");

// Add the dialog buttons.
```

3. Extract the strings above to string resources as `alert_title` and `alert_message`.
4. Add the **OK** and **Cancel** buttons to the alert with `setPositiveButton()` and `setNegativeButton()` methods:

```
// Add the dialog buttons.

myAlertBuilder.setPositiveButton("OK", new

                                DialogInterface.OnClickListener() {

    public void onClick(DialogInterface dialog, int which) {

        // User clicked OK button.

        Toast.makeText(getApplicationContext(), "Pressed OK",

                       Toast.LENGTH_SHORT).show();

    }

});

myAlertBuilder.setNegativeButton("Cancel", new

                                DialogInterface.OnClickListener() {

    public void onClick(DialogInterface dialog, int which) {

        // User cancelled the dialog.

        Toast.makeText(getApplicationContext(), "Pressed Cancel",

                       Toast.LENGTH_SHORT).show();
```

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

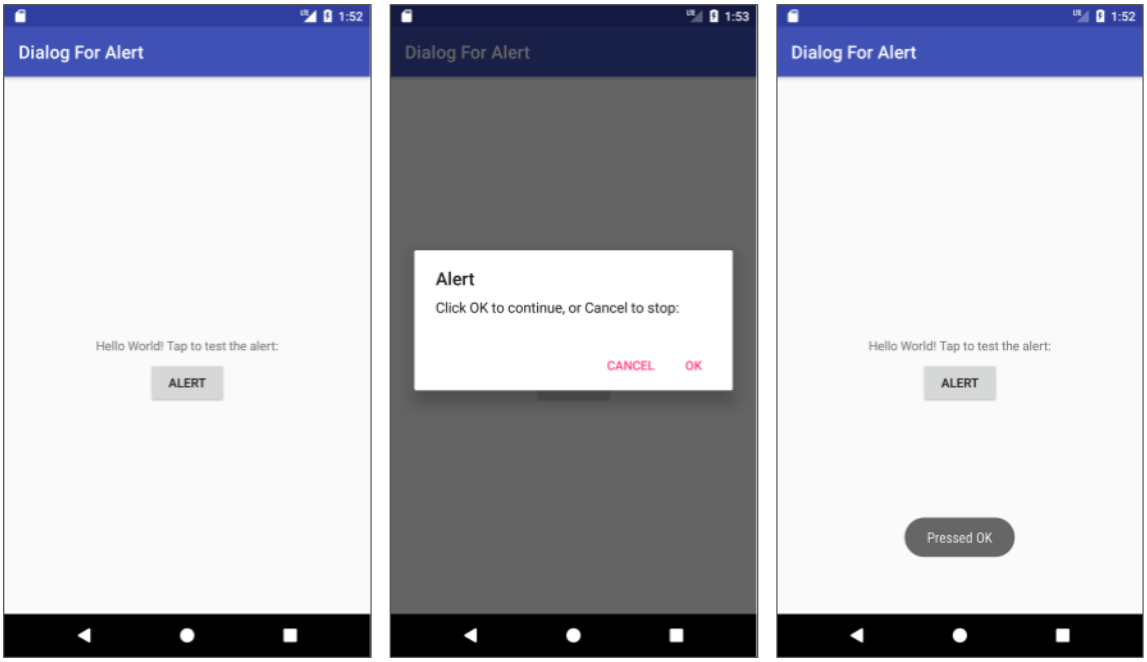
After the user taps the **OK** or **Cancel** button in the alert, you can grab the user's selection and use it in your code. In this example, you display a `Toast` message.

5. Extract the strings for **OK** and **Cancel** to string resources as `ok_button` and `cancel_button`, and extract the strings for the `Toast` messages.
6. At the end of the `onClickShowAlert()` method, add `show()`, which creates and then displays the alert dialog:

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

7. Run the app.

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



Task 4 solution code

Android Studio project: DialogForAlert

Task 5: Use a picker for user input

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

In this task you'll create a new project and add the date picker. You will also learn how to use a `Fragment`, which is a behavior or a portion of a UI within an `Activity`. It's like a mini-`Activity` within the main `Activity`, with its own lifecycle, and it's used for building a picker. All the work is done for you. To learn about the `Fragment` class, see Fragments in the API Guide.

One benefit of using a `Fragment` for a picker is that you can isolate the code sections for managing the date and the time for various locales that display date and time in different ways. The best practice to show a picker is to use an instance of `DialogFragment`, which is a subclass of `Fragment`. A

`DialogFragment` displays a dialog window floating on top of the `Activity` window. In this exercise, you'll add a `Fragment` for the picker dialog and use `DialogFragment` to manage the dialog lifecycle.

Tip: Another benefit of using a `Fragment` for a picker is that you can implement different layout configurations, such as a basic dialog on handset-sized displays or an embedded part of a layout on large displays.

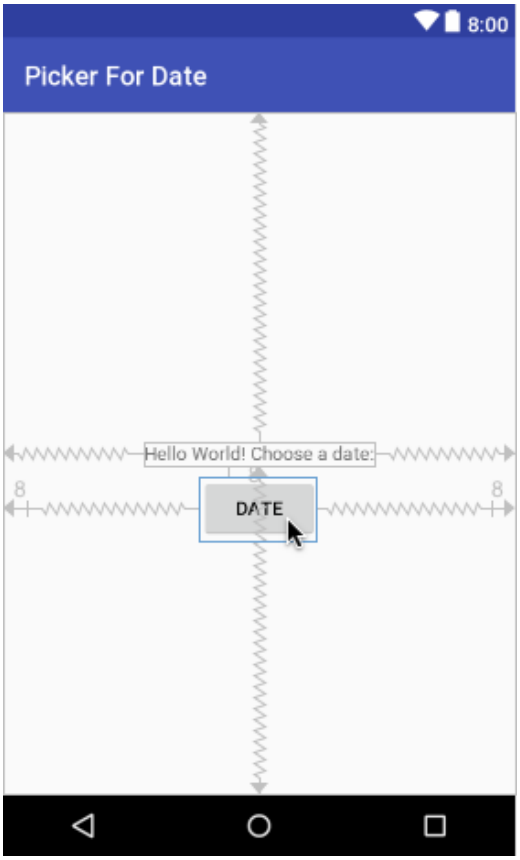
5.1 Create a new app to show a date picker

To start this task, create an app that provides a `Button` to show the date picker.

1. Create a new project called **Picker For Date** based on the Empty Activity template.
2. Open the **activity_main.xml** layout file to show the layout editor.
3. Edit the `TextView` element's "Hello World!" text to **Hello World! Choose a date:**.
4. Add a `Button` underneath the `TextView`. (Optional: Constrain the `Button` to the bottom of the `TextView` and the sides of the layout, with margins set to 8dp.)
5. Set the text of the `Button` to **Date**.
6. Switch to the **Text** tab, and extract the strings for the `TextView` and `Button` to string resources.
7. Add the `android:onClick` attribute to the `Button` to call the click handler `showDatePicker()`. After entering it, the click handler is underlined in red because it has not yet been created.

```
android:onClick="showDatePicker"
```

You should now have a layout similar to the following:



5.2 Create a new fragment for the date picker

In this step, you add a `Fragment` for the date picker.

1. Expand **app > java > com.example.android.pickerfordate** and select **MainActivity**.
2. Choose **File > New > Fragment > Fragment (Blank)**, and name the fragment **DatePickerFragment**. Clear all three checkboxes so that you don't create a layout XML, include fragment factory methods, or include interface callbacks. You don't need to create a layout for a standard picker. Click **Finish**.
3. Open **DatePickerFragment** and edit the `DatePickerFragment` class definition to extend `DialogFragment` and implement `DatePickerDialog.OnDateSetListener` to create a

standard date picker with a listener. See Pickers for more information about extending `DialogFragment` for a date picker:

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

As you enter **DialogFragment** and **DatePickerDialog.OnDateSetListener**, Android Studio automatically adds several `import` statements to the `import` block at the top, including:

```
import android.app.DatePickerDialog;

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

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

4. Click the red bulb icon and choose **Implement methods** from the popup menu. A dialog appears with `onDateSet()` already selected and the **Insert @Override** option selected. Click **OK** to create the empty `onDateSet()` method. This method will be called when the user sets the date.

   After adding the empty `onDateSet()` method, Android Studio automatically adds the following in the `import` block at the top:

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

The `onDateSet()` parameters should be `int i`, `int i1`, and `int i2`. Change the names of these parameters to ones that are more readable:

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

5. Remove the empty `public DatePickerFragment()` public constructor.

6. Replace the entire `onCreateView()` method with `onCreateDialog()` that returns `Dialog`, and annotate `onCreateDialog()` with `@NonNull` to indicate that the `return` value `Dialog` can't be null. Android Studio displays a red bulb next to the method because it doesn't `return` anything yet.

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

7. Add the following code to `onCreateDialog()` to initialize the `year`, `month`, and `day` from `Calendar`, and return the dialog and these values to the `Activity`. As you enter **Calendar.getInstance()**, specify the import to be **java.util.Calendar**.

```
// Use the current date as the default date in the picker.
final Calendar c = Calendar.getInstance();
int year = c.get(Calendar.YEAR);
int month = c.get(Calendar.MONTH);
int day = c.get(Calendar.DAY_OF_MONTH);

// Create a new instance of DatePickerDialog and return it.
return new DatePickerDialog(getActivity(), this, year, month, day);
```

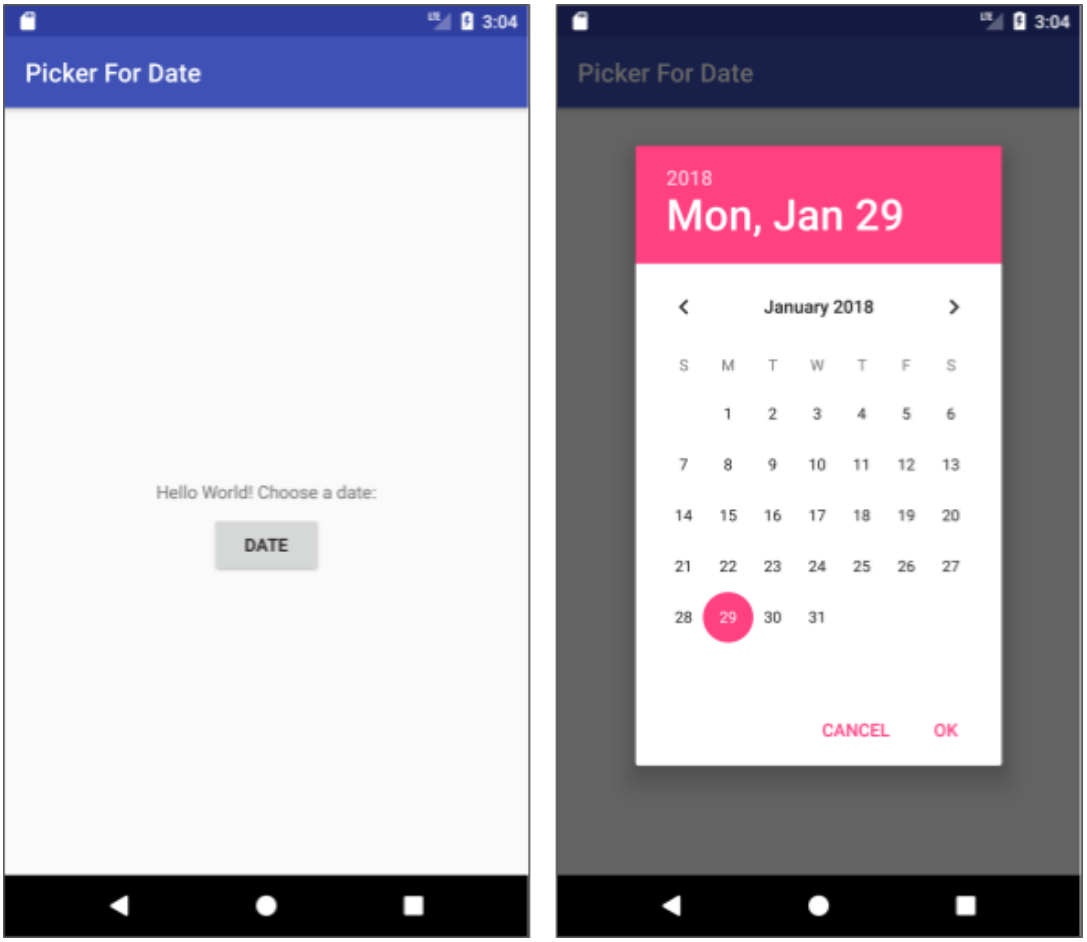
5.4 Modify the main activity

While much of the code in `MainActivity.java` stays the same, you need to add a method that creates an instance of `FragmentManager` to manage the `Fragment` and show the date picker.

1. Open **MainActivity**.
2. Add the `showDatePickerDialog()` handler for the **Date** `Button`. It creates an instance of `FragmentManager` using `getSupportFragmentManager()` to manage the `Fragment` automatically, and to show the picker. For more information about the `Fragment` class, see Fragments.

```
public void showDatePicker(View view) {
        DialogFragment newFragment = new DatePickerFragment();
        newFragment.show(getSupportFragmentManager(),"datePicker");
}
```

3. Extract the string `"datePicker"` to the string resource `datepicker`.
4. Run the app. You should see the date picker after tapping the **Date** button.



5.5 Use the chosen date

In this step you pass the date back to `MainActivity.java`, and convert the date to a string that you can show in a `Toast` message.

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

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

2. Add the following code to the `processDatePickerResult()` method to convert the `month`, `day`, and `year` to separate strings, and to concatenate the three strings with slash marks for the U.S. date format:

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

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

3. Add the following after the code above to display a `Toast` message:

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

4. Extract the hard-coded string `"Date: "` into a string resource named `date`.

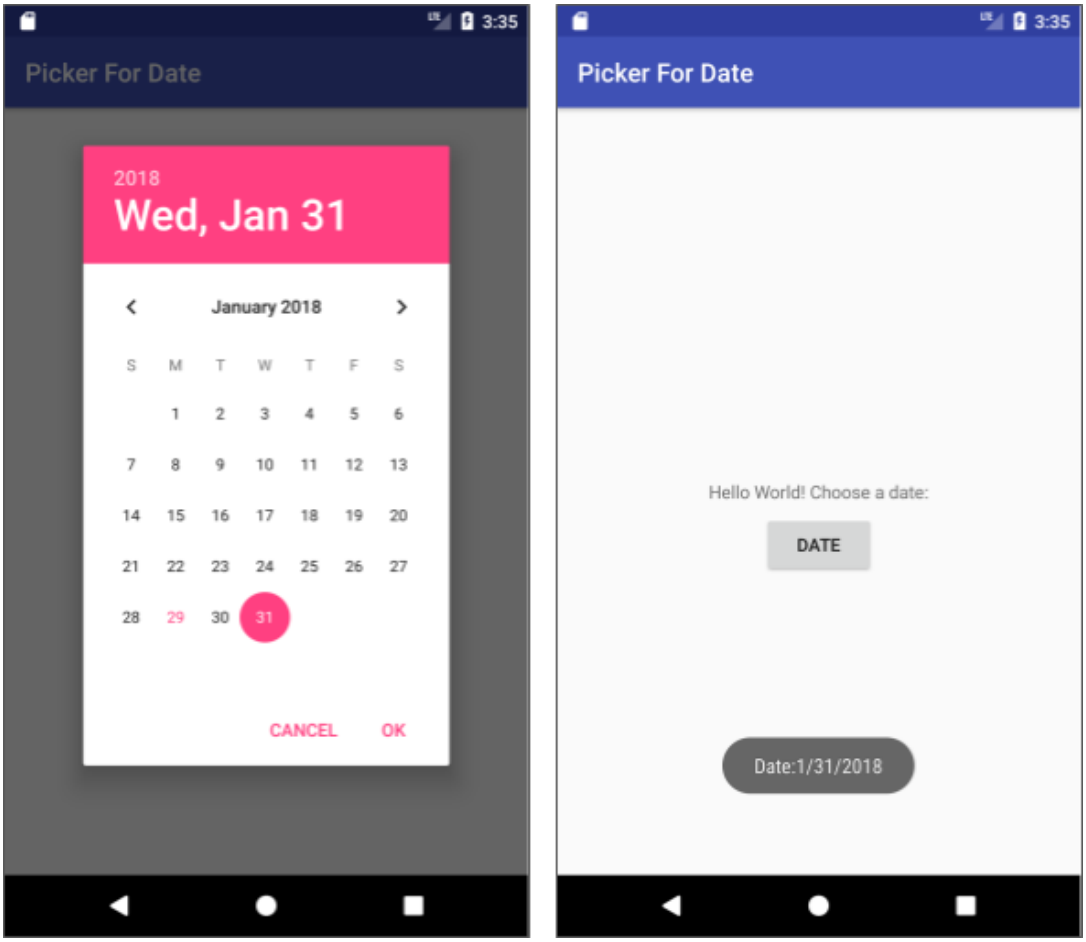
5. Open **DatePickerFragment**, and add the following to the `onDateSet()` method to invoke `processDatePickerResult()` in `MainActivity` and pass it the `year`, `month`, and `day`:

```
@Override
public void onDateSet(DatePicker datePicker,
                                int year, int month, int day) {
```

```
    MainActivity activity = (MainActivity) getActivity();
    activity.processDatePickerResult(year, month, day);
}
```

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

6. Run the app. After selecting the date, the date appears in a `Toast` message as shown on the right side of the following figure.



Task 5 solution code

Android Studio project: PickerForDate

Coding challenge 2

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

Challenge: Create an app called **Picker For Time** that implements the time picker using the same technique you just learned for adding a date picker.

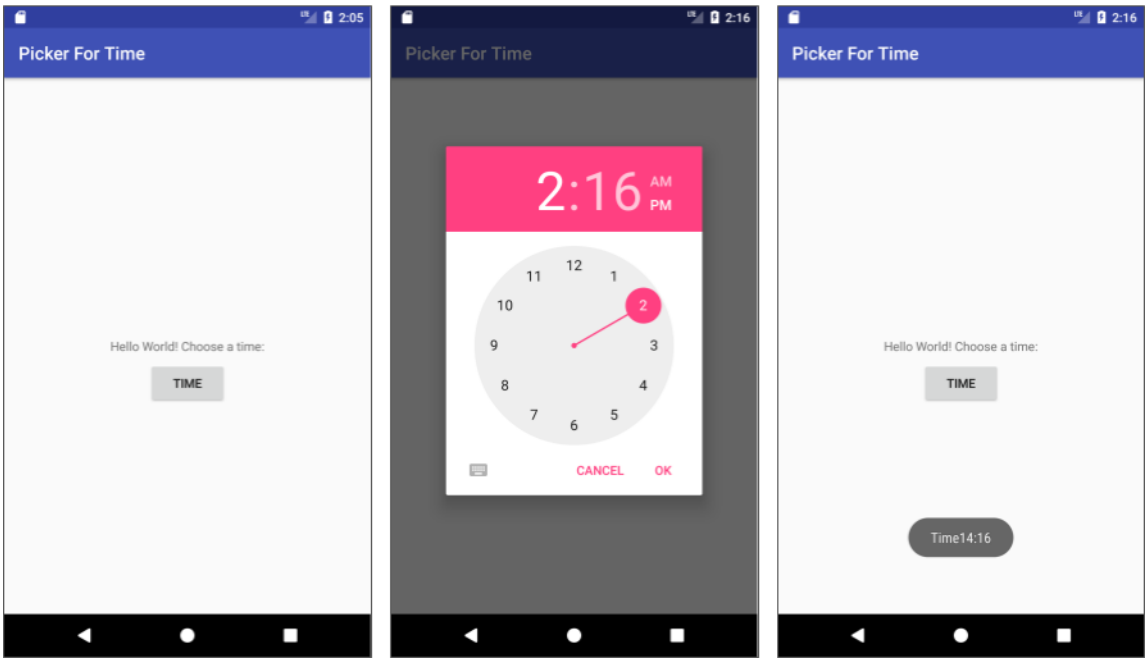
Hints:

- Implement `TimePickerDialog.OnTimeSetListener` to create a standard time picker with a listener.
- Change the `onTimeSet()` method's parameters from `int i` to `int hourOfDay` and `int i1` to `int minute`.
- Get the current hour and minute from `Calendar`:

```
final Calendar c = Calendar.getInstance();
int hour = c.get(Calendar.HOUR_OF_DAY);
int minute = c.get(Calendar.MINUTE);
```

- Create a `processTimePickerResult()` method similar to `processDatePickerResult()` in the previous task that converts the time elements to strings and displays the result in a `Toast` message.

Run the app, and click the **Time** button as shown in the left side of the figure below. The time picker should appear, as shown in the center of the figure. Choose a time and click **OK**. The time should appear in a Toast message at the bottom of the screen, as shown in the right side of the figure.



Challenge 2 solution code

Android Studio project: PickerForTime

Summary

Provide an options menu and app bar:

- Start your app or `Activity` with the Basic Activity template to automatically set up the app bar, the options menu, and a floating action button.
- The template sets up a `CoordinatorLayout` layout with an embedded `AppBarLayout` layout. `AppBarLayout` is like a vertical `LinearLayout`. It uses the `Toolbar` class in the support library, instead of the native `ActionBar`, to implement an app bar.

- The template modifies the `AndroidManifest.xml` file so that the `.MainActivity` `Activity` is set to use the `NoActionBar` theme. This theme is defined in the `styles.xml` file.

- The template sets `MainActivity` to extend `AppCompatActivity` and starts with the `onCreate()` method, which sets the content view and `Toolbar`. It then calls `setSupportActionBar()` and passes `toolbar` to it, setting the `toolbar` as the app bar for the `Activity`.
- Define menu items in the `menu_main.xml` file. The `android:orderInCategory` attribute specifies the order in which the menu items appear in the menu, with the lowest number appearing higher in the menu.
- Use the `onOptionsItemSelected()` method to determine which menu item was tapped.

Add an icon for an options menu item:

- Expand **res** in the **Project > Android** pane, and right-click (or Control-click) the **drawable** folder. Choose **New > Image Asset**.
- Choose **Action Bar and Tab Items** in the drop-down menu, and change the name of the image file.
- Click the clip art image to select a clip art image as the icon. Choose an icon.
- Choose **HOLO_DARK** from the **Theme** drop-down menu.

Show menu items as icons in the app bar:

- Use the `app:showAsAction` attribute in `menu_main.xml` with the following values.
- `"always"`: Always appears in the app bar. (If there isn't enough room it may overlap with other menu icons.)
- `"ifRoom"`: Appears in the app bar if there is room.
- `"never"`: Never appears in the app bar; its text appears in the overflow menu.

Use an alert dialog:

- Use a dialog to request a user's choice, such as an alert that requires users to tap **OK** or **Cancel**. Use dialogs sparingly as they interrupt the user's workflow.

- Use the `AlertDialog` subclass of the `Dialog` class to show a standard dialog for an alert.
- Use `AlertDialog.Builder` to build a standard alert dialog, with `setTitle()` to set its title, `setMessage()` to set its message, and `setPositiveButton()` and `setNegativeButton()` to set its buttons.

Use a picker for user input:

- Use `DialogFragment`, a subclass of `Fragment`, to build a picker such as the date picker or time picker.
- Create a `DialogFragment`, and implement `DatePickerDialog.OnDateSetListener` to create a standard date picker with a listener. Include `onDateSet()` in this `Fragment`.
- Replace the `onCreateView()` method with `onCreateDialog()` that returns `Dialog`. Initialize the date for the date picker from `Calendar`, and return the dialog and these values to the `Activity`.
- Create an instance of `FragmentManager` using `getSupportFragmentManager()` to manage the `Fragment` and show the date picker.

Related concept

The related concept documentation is in 4.3: Menus and pickers.

Learn more

Android Studio documentation:

- Android Studio User Guide
- Create App Icons with Image Asset Studio

Android developer documentation:

- Add the app bar
- Menus
- `Toolbar`
- v7 appcompat support library
- `AppBarLayout`
- `onOptionsItemSelected()`
- `View`
- `MenuInflater`
- `registerForContextMenu()`
- `onCreateContextMenu()`
- `onContextItemSelected()`
- Dialogs
- `AlertDialog`
- Pickers
- Fragments
- `DialogFragment`
- `FragmentManager`
- `Calendar`

Material Design spec:

- Responsive layout grid
- Dialogs

Other:

- Android Developers Blog: Android Design Support Library
- Builder pattern in Wikipedia

Homework

Build and run an app

Open the DroidCafeOptions app that you created in this lesson.

1. Add a Date button under the delivery options that shows the date picker.
2. Show the user's chosen date in a `Toast` message.

Answer these questions

Question 1

What is the name of the file in which you create options menu items? Choose one:

- menu.java
- menu_main.xml
- activity_main.xml
- content_main.xml

Question 2

Which method is called when an options menu item is clicked? Choose one:

- `onOptionsItemSelected(MenuItem item)`
- `onClick(View view)`
- `onContextItemSelected()`
- `onClickShowAlert()`

Question 3

Which of the following statements sets the title for an alert dialog? Choose one:

- `myAlertBuilder.setMessage("Alert");`
- `myAlertBuilder.setPositiveButton("Alert");`

- `myAlertBuilder.setTitle("Alert");`
- `AlertDialog.Builder myAlertBuilder = new AlertDialog.Builder("Alert");`

Question 4

Where do you create a `DialogFragment` for a date picker? Choose one:

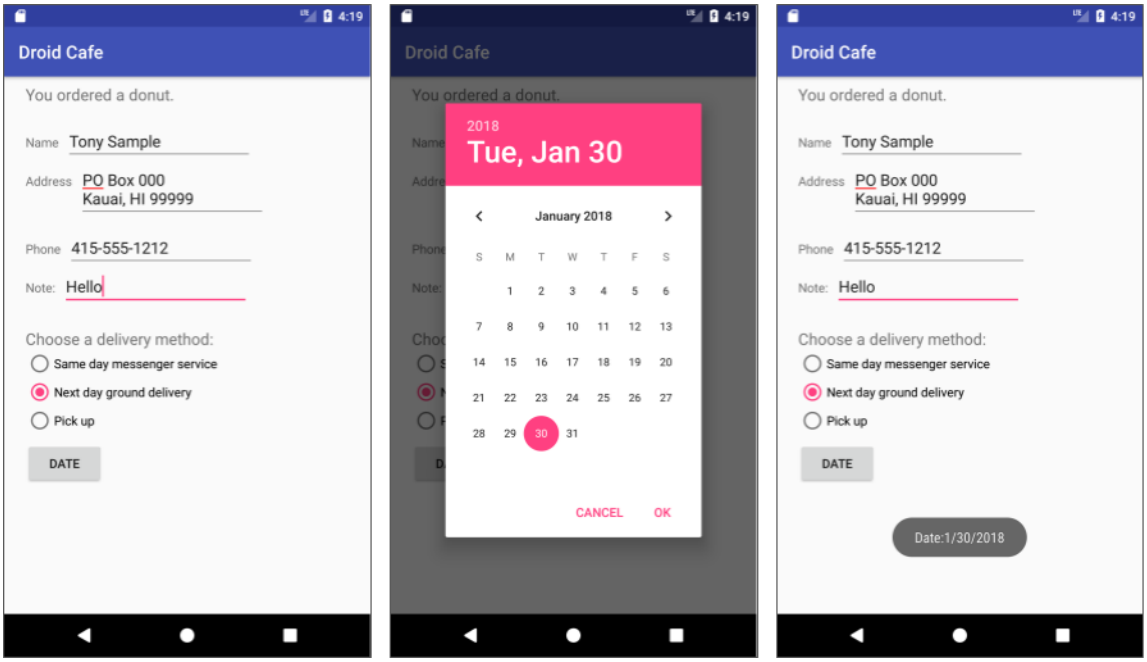
- In the `onCreate()` method in the hosting `Activity`.
- In the `onCreateContextMenu()` method in `Fragment`.
- In the `onCreateView()` method in the extension of `DialogFragment`.
- In the `onCreateDialog()` method in the extension of `DialogFragment`.

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- The date picker is added as a `DialogFragment`.
- Clicking the **Date** button (refer to the left side of the figure below) in `OrderActivity` shows the date picker (refer to the center of the figure).
- Clicking the **OK** button in the date picker shows a `Toast` message in `OrderActivity` with the chosen date (refer to the right side of the figure).

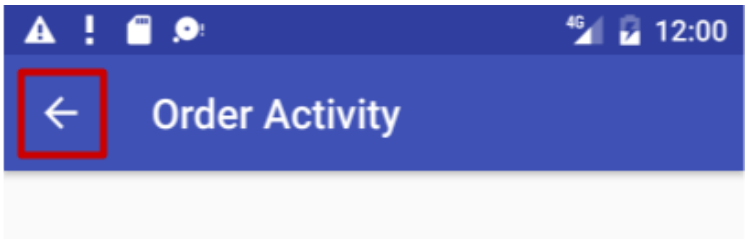


Lesson 4.4: User navigation

Introduction

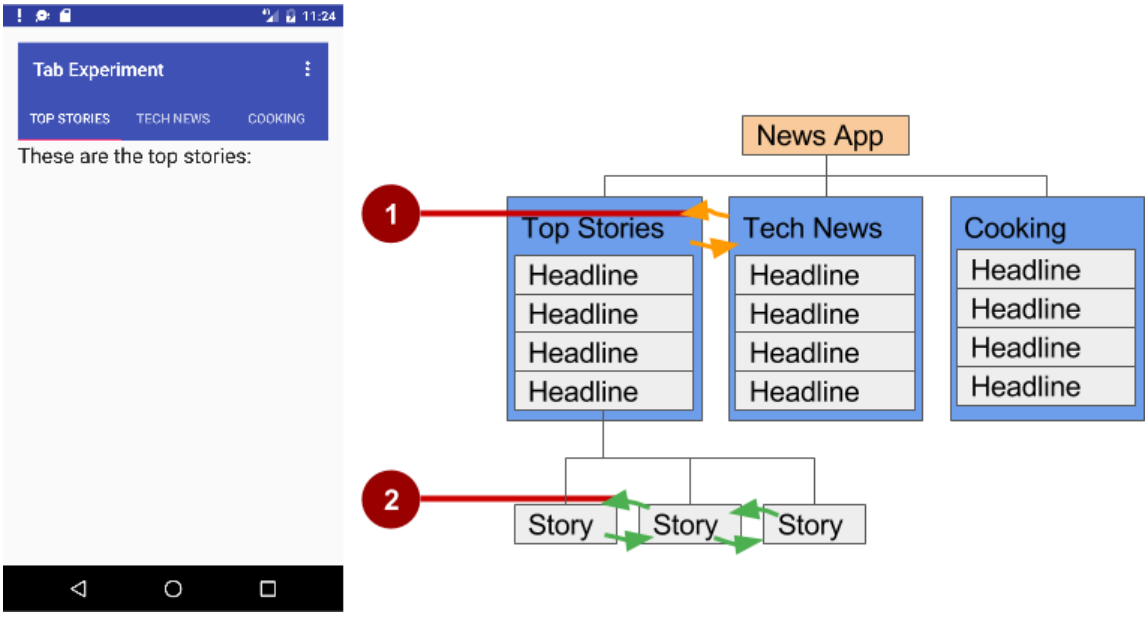
In the early stages of developing an app, you should determine the path you want users to take through your app to do each task. (The tasks are things like placing an order or browsing content.) Each path enables users to navigate across, into, and out of the tasks and pieces of content within the app.

In this practical, you learn how to add an **Up** button (a left-facing arrow) to the app bar, as shown below.



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

This practical also introduces *tab navigation*, in which tabs appear across the top of a screen, providing navigation to other screens. Tab navigation is a popular way to create lateral navigation from one child screen to a sibling child screen, as shown in the figure below.



In the figure above:

1. Lateral navigation from one category screen (**Top Stories**, **Tech News**, and **Cooking**) to another
2. Lateral navigation from one story screen (**Story**) to another

With the tabs, the user can navigate to and from the sibling screens without navigating up to the parent screen. Tabs can also provide navigation to and from stories, which are sibling screens under the **Top Stories** parent.

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

What you should already know

You should be able to:

- Create and run apps in Android Studio.
- Create and edit UI elements using the layout editor.
- Edit XML layout code, and access elements from your Java code.
- Add menu items and icons to the options menu in the app bar.

What you'll learn

- How to add the **Up** button to the app bar.
- How to set up an app with tab navigation and swipe views.

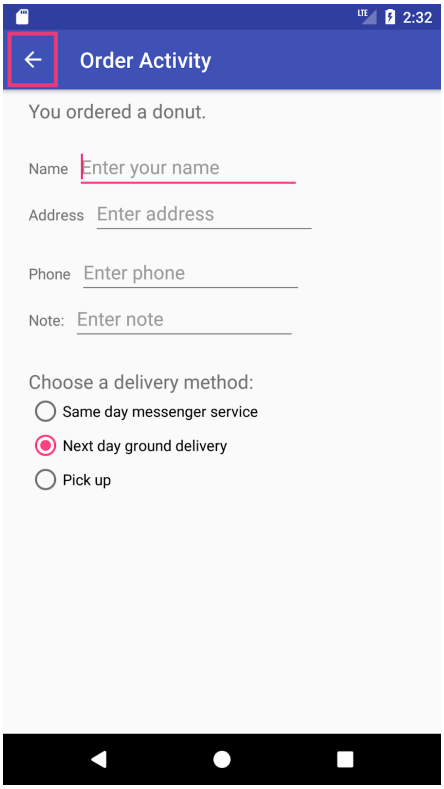
What you'll do

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

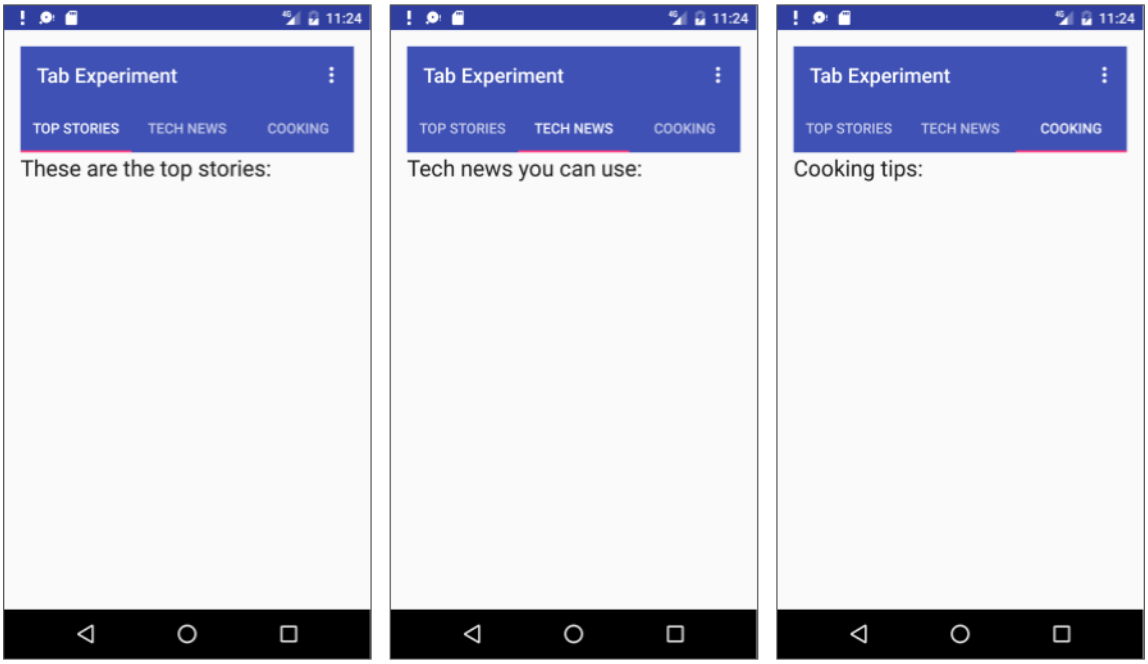
App overview

In the previous practical on using the options menu, you worked on an app called Droid Cafe that was created using the Basic Activity template. This template provides an app bar at the top of the screen. You will learn how to add an **Up** button (a left-facing arrow) to the app bar for up navigation from the second `Activity` (`OrderActivity`) to the parent `Activity` (`MainActivity`). This will complete the Droid Cafe app.

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



You will also create an app for tab navigation that shows three tabs below the app bar to navigate to sibling screens. When the user taps a tab, the screen shows a content screen, depending on which tab the user tapped. The user can also swipe left and right to visit the content screens. The `ViewPager` class automatically handles user swipes to screens or `View` elements.

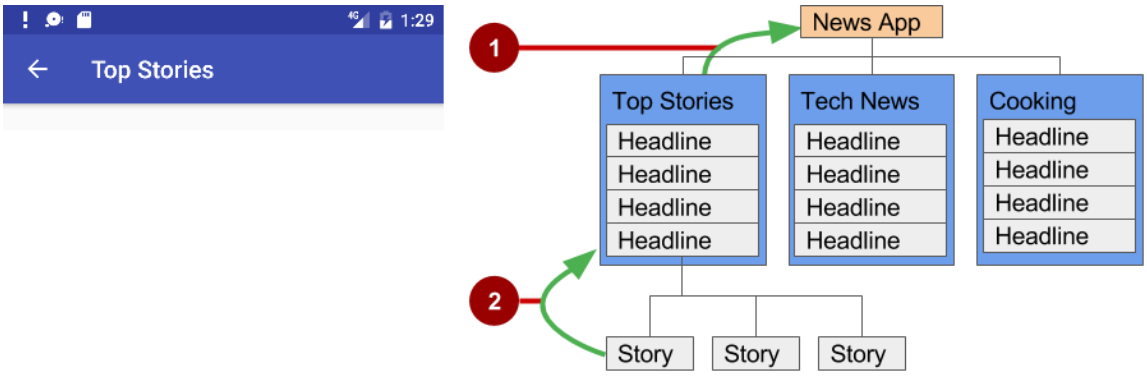


Task 1: Add an Up button for ancestral navigation

Your app should make it easy for users to find their way back to the app's main screen, which is usually the parent `Activity`. One way to do this is to provide an **Up** button in the app bar for each `Activity` that is a child of the parent `Activity`.

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

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



In the figure above:

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

Tip: The Back button (the triangle at the bottom of the device) and the **Up** button in the UI are two different things:

The Back button provides navigation to the screen that was most recently viewed. If you have several child screens that the user can navigate using a lateral navigation pattern (as described in the next section), the Back button sends the user back to the previous child screen, not to the parent screen.
To provide navigation from a child screen back to the parent screen, use an **Up** button. For more about Up navigation, see Providing Up navigation.

As you learned previously, when adding activities to an app, you can add **Up**-button navigation to a child `Activity` such as `OrderActivity` by declaring the parent of the `Activity` to be `MainActivity` in the `AndroidManifest.xml` file. You can also set the `android:label` attribute for a title for the `Activity` screen, such as `"Order Activity"`. Follow these steps:

1. If you don't already have the Droid Cafe app open from the previous practical, download the Android Studio project DroidCafeOptions and open the project.

2. Open **AndroidManifest.xml** and change the `Activity` element for `OrderActivity` to the following:

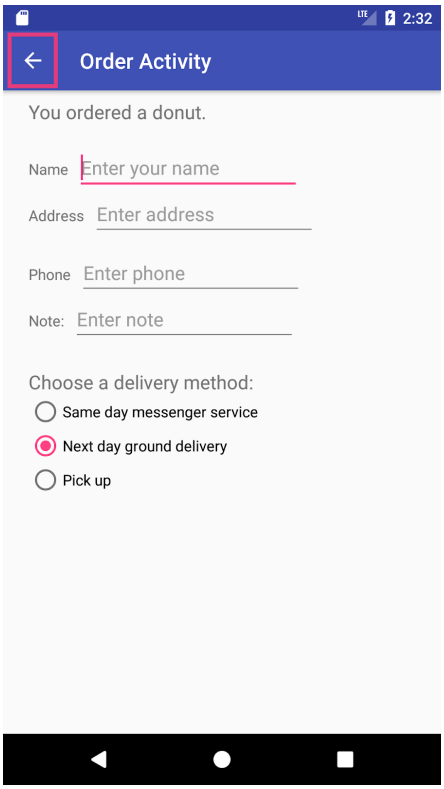
```
<activity android:name="com.example.android.droidcafeinput.OrderActivity"
```

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

3. Extract the android:label value `"Order Activity"` to a string resource named `title_activity_order`.

4. Run the app.

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

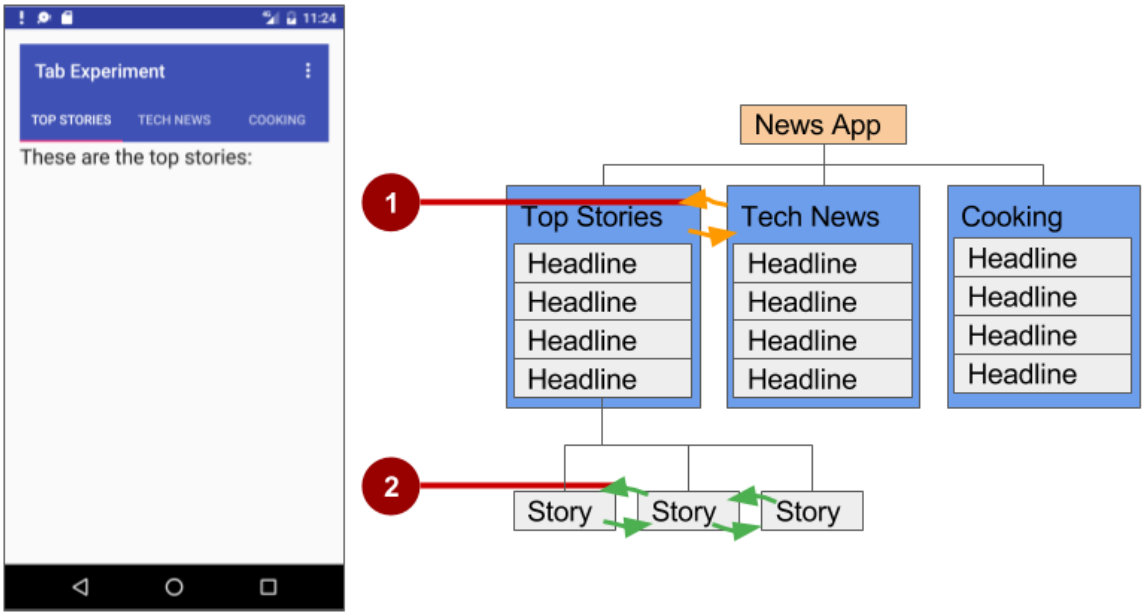


Task 1 solution code

Android Studio project: DroidCafeOptionsUp

Task 2: Use tab navigation with swipe views

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



In the figure above:

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

You can implement lateral navigation with *tabs* that represent each screen. Tabs appear across the top of a screen, as shown on the left side of the figure above, in order to provide navigation to other screens. Tab navigation is a very popular solution for lateral navigation from one child screen to another child screen that is a *sibling*—in the same position in the hierarchy and sharing the same

parent screen. Tab navigation is often combined with the ability to swipe child screens left-to-right and right-to-left.

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

You implement `PagerAdapter` to generate the screens that the view shows. `ViewPager` is most often used in conjunction with `Fragment`. By using a `Fragment`, you have a convenient way to manage the lifecycle of a screen "page".

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

The following are standard adapters for using fragments with the `ViewPager`:

- `FragmentPagerAdapter`: Designed for navigating between sibling screens (pages) representing a fixed, small number of screens.
- `FragmentStatePagerAdapter`: Designed for paging across a collection of screens (pages) for which the number of screens is undetermined. It destroys each `Fragment` as the user navigates to other screens, minimizing memory usage. The app for this task uses `FragmentStatePagerAdapter`.

2.1 Create the project and layout

1. Create a new project using the Empty Activity template. Name the app **Tab Experiment**.
2. Edit the **build.gradle (Module: app)** file, and add the following line to the `dependencies` section for the Android Design Support Library, which you need in order to use a `TabLayout`:

```
implementation 'com.android.support:design:26.1.0'
```

If Android Studio suggests a version with a higher number, edit the line above to update the version.

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

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

4. Open **activity_main.xml** layout file, and click the **Text** tab to view the XML code.
5. Change the `ConstraintLayout` to **RelativeLayout**, as you've done in previous exercises.
6. Add the `android:id` attribute and `android:padding` of `16dp` to the `RelativeLayout`.
7. Remove the `TextView` supplied by the template, and add a `Toolbar`, a `TabLayout`, and a `ViewPager` within the `RelativeLayout` as shown in the code below.

```
<RelativeLayout 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:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    tools:context="com.example.android.tabexperiment.MainActivity">

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:background="?attr/colorPrimary"
        android:minHeight="?attr/actionBarSize"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>

    <android.support.design.widget.TabLayout
        android:id="@+id/tab_layout"
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/toolbar"
        android:background="?attr/colorPrimary"
        android:minHeight="?attr/actionBarSize"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"/>

    <android.support.v4.view.ViewPager
        android:id="@+id/pager"
        android:layout_width="match_parent"
        android:layout_height="fill_parent"
        android:layout_below="@id/tab_layout"/>

</RelativeLayout>
```

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

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

You can click `app` and press `Option+Enter` (or `Alt+Enter`), and Android Studio automatically adds the statement.

2.2 Create a class and layout for each fragment

To add a fragment representing each tabbed screen, follow these steps:

1. Click **com.example.android.tabexperiment** in the **Android > Project** pane.
2. Choose **File > New > Fragment > Fragment (Blank)**.
3. Name the fragment **TabFragment1**.
4. Select the **Create layout XML?** option.
5. Change the **Fragment Layout Name** for the XML file to **tab_fragment1**.

6. Clear the **Include fragment factory methods?** option and the **Include interface callbacks?** option. You don't need these methods.
7. Click **Finish**.

Repeat the steps above, using **TabFragment2** and **TabFragment3** for Step 3, and **tab_fragment2** and **tab_fragment3** for Step 4.

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

For example, **TabFragment1** looks like this:

```
public class TabFragment1 extends Fragment {

    public TabFragment1() {
        // 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.tab_fragment1, container, false);
    }

}
```

2.3 Edit the fragment layout

Edit each `Fragment` layout XML file (**tab_fragment1**, **tab_fragment2**, and **tab_fragment3**):

1. Change the `FrameLayout` to **RelativeLayout**.
2. Change the `TextView` text to **"These are the top stories: "** and the `layout_width` and `layout_height` to **wrap_content**.
3. Set the text appearance with `android:textAppearance="?android:attr/textAppearanceLarge"`.

Repeat the steps above for each fragment layout XML file, entering different text for the `TextView` in

step 2:

- Text for the `TextView` in `tab_fragment2.xml`: **"Tech news you can use: "**
- Text for the `TextView` in `tab_fragment3.xml`: **"Cooking tips: "**

Examine each fragment layout XML file. For example, **tab_fragment1** should look like this:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.android.tabexperiment.TabFragment1">

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

</RelativeLayout>
```

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

2.3 Add a PagerAdapter

The adapter-layout manager pattern lets you provide different screens of content within an `Activity`:

- Use an *adapter* to fill the content screen to show in the `Activity`.
- Use a *layout manager* that changes the content screens depending on which tab is selected.

Follow these steps to add a new `PagerAdapter` class to the app that extends `FragmentStatePagerAdapter` and defines the number of tabs (`mNumOfTabs`):

1. Click **com.example.android.tabexperiment** in the **Android > Project** pane.
2. Choose **File > New > Java Class**.
3. Name the class **PagerAdapter**, and enter **FragmentStatePagerAdapter** into the Superclass field. This entry changes to `android.support.v4.app.FragmentStatePagerAdapter`.
4. Leave the **Public** and **None** options selected, and click **OK**.
5. Open **PagerAdapter** in the **Project > Android** pane. A red light bulb should appear next to the class definition. Click the bulb and choose **Implement methods**, and then click **OK** to implement the already selected `getItem()` and `getCount()` methods.
6. Another red light bulb should appear next to the class definition. Click the bulb and choose **Create constructor matching super**.
7. Add an integer member variable `mNumOfTabs`, and change the constructor to use it. The code should now look as follows:

```
public class PagerAdapter extends FragmentStatePagerAdapter {
    int mNumOfTabs;

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

    /**
     * Return the Fragment associated with a specified position.
     *
     * @param position
     */
    @Override
    public Fragment getItem(int position) {
        return null;
    }

    /**
     * Return the number of views available.
     */
    @Override
    public int getCount() {
        return 0;
    }
}
```

While entering the code above, Android Studio automatically imports the following:

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

If `FragmentManager` in the code is in red, a red light bulb icon should appear when you click it. Click the bulb icon and choose **Import class**. Import choices appear. Select **FragmentManager (android.support.v4)**.

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

```
@Override
public Fragment getItem(int position) {
        switch (position) {
            case 0: return new TabFragment1();
            case 1: return new TabFragment2();
            case 2: return new TabFragment3();
            default: return null;
        }
}
```

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

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

2.4 Inflate the Toolbar and TabLayout

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

1. Open **MainActivity** and add the following code inside the `onCreate()` method to inflate the `Toolbar` using `setSupportActionBar()`:

```
protected void onCreate(Bundle savedInstanceState) {
    // ... Code inside onCreate() method
    android.support.v7.widget.Toolbar toolbar =
                                  findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    // Create an instance of the tab layout from the view.
}
```

2. Open **strings.xml**, and create the following string resources:

```
<string name="tab_label1">Top Stories</string>
<string name="tab_label2">Tech News</string>
<string name="tab_label3">Cooking</string>
```

3. At the end of the `onCreate()` method, create an instance of the tab layout from the `tab_layout` element in the layout, and set the text for each tab using addTab():

```
// Create an instance of the tab layout from the view.
TabLayout tabLayout = findViewById(R.id.tab_layout);
// Set the text for each tab.
tabLayout.addTab(tabLayout.newTab().setText(R.string.tab_label1));
tabLayout.addTab(tabLayout.newTab().setText(R.string.tab_label2));
tabLayout.addTab(tabLayout.newTab().setText(R.string.tab_label3));
```

```
// Set the tabs to fill the entire layout.
tabLayout.setTabGravity(TabLayout.GRAVITY_FILL);
// Use PagerAdapter to manage page views in fragments.
```

2.5 Use PagerAdapter to manage screen views

1. Below the code you added to the `onCreate()` method in the previous task, add the following code to use `PagerAdapter` to manage screen (page) views in the fragments:

```
// Use PagerAdapter to manage page views in fragments.
// Each page is represented by its own fragment.
final ViewPager viewPager = findViewById(R.id.pager);
final PagerAdapter adapter = new PagerAdapter
              (getSupportFragmentManager(), tabLayout.getTabCount());
viewPager.setAdapter(adapter);
// Setting a listener for clicks.
```

2. At the end of the `onCreate()` method, set a listener (`TabLayoutOnPageChangeListener`) to detect if a tab is clicked, and create the `onTabSelected()` method to set the `ViewPager` to the appropriate tabbed screen. The code should look as follows:

```
// Setting a listener for clicks.
viewPager.addOnPageChangeListener(new
                TabLayout.TabLayoutOnPageChangeListener(tabLayout));
tabLayout.addOnTabSelectedListener(new
                TabLayout.OnTabSelectedListener() {
    @Override
    public void onTabSelected(TabLayout.Tab tab) {
        viewPager.setCurrentItem(tab.getPosition());
    }

    @Override
    public void onTabUnselected(TabLayout.Tab tab) {
    }
```

```
    @Override
    public void onTabReselected(TabLayout.Tab tab) {
    }
});
```

3. Run the app. Tap each tab to see each "page" (screen). You should also be able to swipe left and right to visit the different "pages".

Task 2 solution code

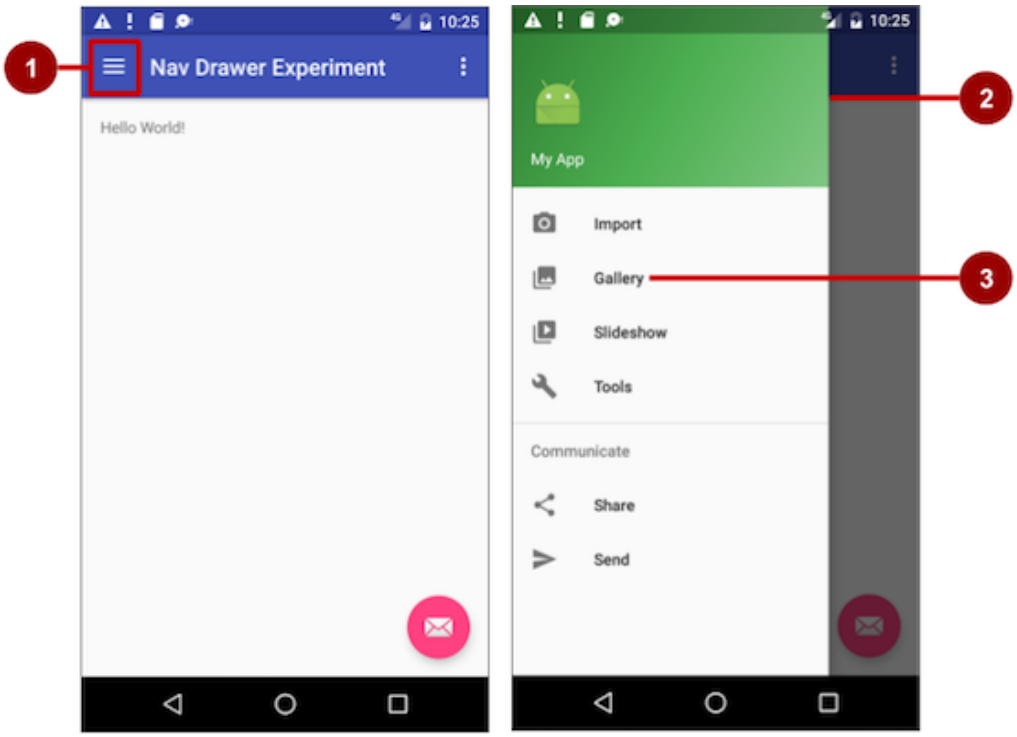
Android Studio project: TabExperiment

Coding challenge

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

Challenge: Create a new app with a navigation drawer. When the user taps a navigation drawer choice, close the drawer and display a `Toast` message showing which choice was selected.

A *navigation drawer* is a panel that usually displays navigation options on the left edge of the screen, as shown on the right side of the figure below. It is hidden most of the time, but is revealed when the user swipes a finger from the left edge of the screen or touches the navigation icon in the app bar, as shown on the left side of the figure below.



In the figure above:

1. Navigation icon in the app bar
2. Navigation drawer
3. Navigation drawer menu item

To make a navigation drawer in your app, you need to create the following layouts:

- A navigation drawer as the `Activity` layout root `ViewGroup`.
- A navigation `View` for the drawer itself.
- An app bar layout that will include a navigation icon button.
- A content layout for the `Activity` that displays the navigation drawer.
- A layout for the navigation drawer header.

After creating the layouts, you need to:

- Populate the navigation drawer menu with item titles and icons.
- Set up the navigation drawer and item listeners in the activity code.

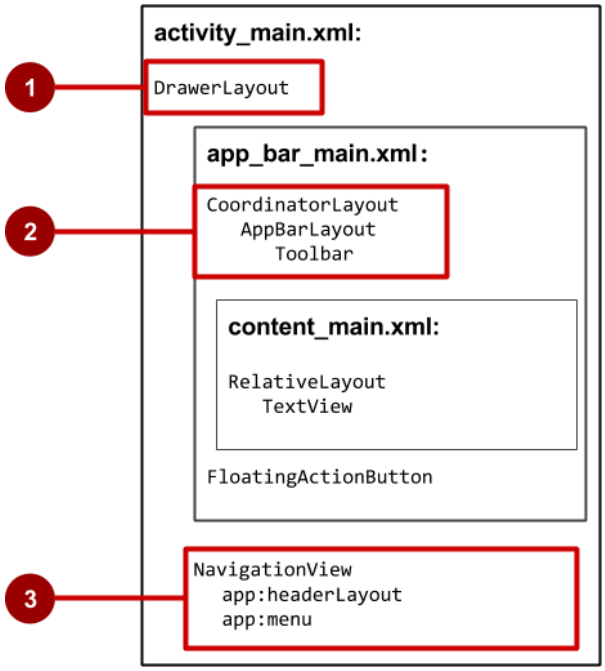
- Handle the navigation menu item selections.

To create a navigation drawer layout, use the `DrawerLayout` APIs available in the Support Library. For design specifications, follow the design principles for navigation drawers in the Navigation Drawer design guide.

To add a navigation drawer, use a `DrawerLayout` as the root view of your `Activity` layout. Inside the `DrawerLayout`, add one `View` that contains the main content for the screen (your primary layout when the drawer is hidden) and another `View`, typically a `NavigationView`, that contains the contents of the navigation drawer.

Tip: To make your layouts simpler to understand, use the `include` tag to include an XML layout within another XML layout.

The figure below is a visual representation of the `activity_main.xml` layout and the XML layouts that it includes:



In the figure above:

1. `DrawerLayout` is the root view of the `Activity` layout.
2. The included `app_bar_main.xml` uses a `CoordinatorLayout` as its root, and defines the app bar layout with a `Toolbar` which will include the navigation icon to open the drawer.
3. The `NavigationView` defines the navigation drawer layout and its header, and adds menu items to it.

Challenge solution code

Android Studio project: NavDrawerExperiment

Summary

App bar navigation:

- Add Up-button navigation to a child `Activity` by declaring the parent `Activity` in the `AndroidManifest.xml` file.
- Declare the child's parent `Activity` within the child's `<activity ... </activity>` section:

```
android:parentActivityName=".MainActivity">
<meta-data android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
```

Tab navigation:

- Tabs are a good solution for "lateral navigation" between sibling views.
- The primary class used for tabs is `TabLayout` in the Android Design Support Library.
- `ViewPager` is a layout manager that allows the user to flip left and right through pages of data. `ViewPager` is most often used in conjunction with `Fragment`.
- Use one of the two standard adapters for using `ViewPager`: `FragmentPagerAdapter` or `FragmentStatePagerAdapter`.

Related concept

The related concept documentation is in 4.4: User navigation.

Learn more

Android developer documentation:

- User Interface & Navigation
- Designing effective navigation
- Implementing effective navigation
- Creating swipe views with tabs
- Create a navigation drawer
- Designing Back and Up navigation
- Providing Up navigation
- Implementing Descendant Navigation
- `TabLayout`
- Navigation Drawer
- `DrawerLayout`
- Support Library

Material Design spec:

- Understanding navigation
- Responsive layout grid

Android Developers Blog: Android Design Support Library

Other:

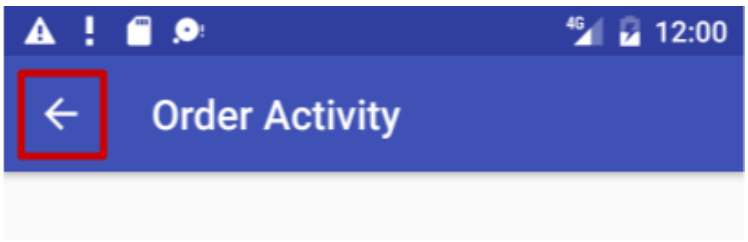
- AndroidHive: Android Material Design working with Tabs
- Truiton: Android Tabs Example – With Fragments and ViewPager

Homework

Build and run an app

Create an app with a main `Activity` and at least three other `Activity` children. Each `Activity` should have an options menu and use the v7 appcompat support library `Toolbar` as the app bar, as shown below.

1. In the main `Activity`, build a grid layout with images of your own choosing. Three images (`donut_circle.png`, `froyo_circle.png`, and `icecream_circle.png`), which you can download, are provided as part of the DroidCafe app.
2. Resize the images if necessary, so that three of them fit horizontally on the screen in the grid layout.
3. Enable each image to provide navigation to a child `Activity`. When the user taps the image, it starts a child `Activity`. From each child `Activity`, the user should be able to tap the **Up** button in the app bar (highlighted in the figure below) to return to the main `Activity`.



Answer these questions

Question 1

Which template provides an `Activity` with an options menu and the v7 appcompat support library `Toolbar` as the app bar? Choose one:

- Empty Activity template
- Basic Activity template
- Navigation Drawer Activity template
- Bottom Navigation Activity

Question 2

Which dependency do you need in order to use a `TabLayout`? Choose one:

- `com.android.support:design`
- `com.android.support.constraint:constraint-layout`
- `junit:junit:4.12`
- `com.android.support.test:runner`

Question 3

Where do you define each child `Activity` and parent `Activity` to provide **Up** navigation? Choose one:

- To provide the **Up** button for a child screen `Activity`, declare the parent `Activity` in the child `Activity` section of the `activity_main.xml` file.
- To provide the **Up** button for a child screen `Activity`, declare the parent `Activity` in the "main" XML layout file for the child screen `Activity`.
- To provide the **Up** button for a child screen `Activity`, declare the parent `Activity` in the child `Activity` section of the `AndroidManifest.xml` file.
- To provide the **Up** button for a child screen `Activity`, declare the parent `Activity` in the parent `Activity` section of the `AndroidManifest.xml` file.

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- A `GridLayout` in the `content_main.xml` file.
- A new `Intent` and `startActivity()` method for each navigation element in the grid.
- A separate `Activity` for each navigation element in the grid.

Lesson 4.5: RecyclerView

Introduction

Letting the user display, scroll, and manipulate a list of similar data items is a common app feature. Examples of scrollable lists include contact lists, playlists, saved games, photo directories, dictionaries, shopping lists, and indexes of documents.

In the practical on scrolling views, you use `ScrollView` to scroll a `View` or `ViewGroup`. `ScrollView` is easy to use, but it's not recommended for long, scrollable lists.

`RecyclerView` is a subclass of `ViewGroup` and is a more resource-efficient way to display scrollable lists. Instead of creating a `View` for each item that may or may not be visible on the screen, `RecyclerView` creates a limited number of list items and reuses them for visible content.

In this practical you do the following:

- Use `RecyclerView` to display a scrollable list.
- Add a click handler to each list item.
- Add items to the list using a floating action button (FAB), the pink button in the screenshot in the app overview section. Use a FAB for the primary action that you want the user to take.

What you should already know

You should be able to:

- Create and run apps in Android Studio.
- Create and edit UI elements using the layout editor, entering XML code directly, and accessing elements from your Java code.
- Create and use string resources.
- Convert the text in a `View` to a string using `getText()`.
- Add an `onClick()` handler to a `View`.
- Display a `Toast` message.

What you'll learn

- How to use the `RecyclerView` class to display items in a scrollable list.
- How to dynamically add items to the `RecyclerView` as they become visible through scrolling.
- How to perform an action when the user taps a specific item.
- How to show a FAB and perform an action when the user taps it.

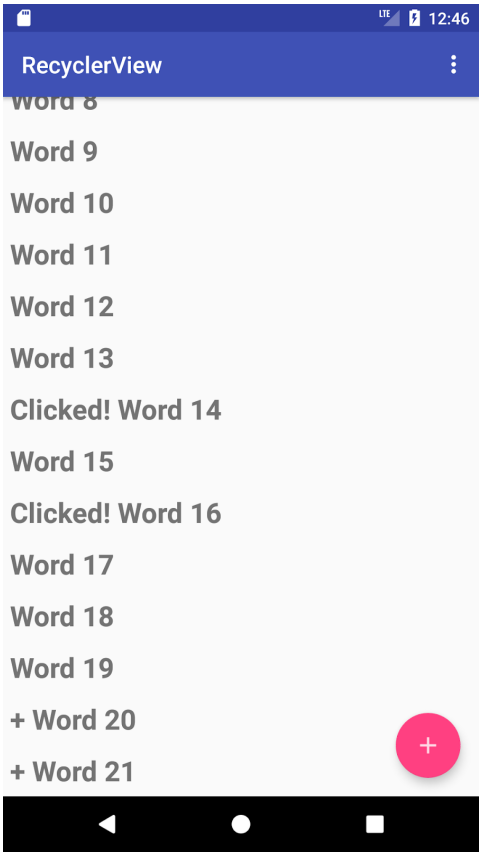
What you'll do

- Create a new app that uses a `RecyclerView` to display a list of items as a scrollable list and associate click behavior with the list items.
- Use a FAB to let the user add items to the `RecyclerView`.

App Overview

The RecyclerView app demonstrates how to use a `RecyclerView` to display a long scrollable list of words. You create the dataset (the words), the `RecyclerView` itself, and the actions the user can take:

- Tapping a word marks it clicked.
- Tapping the floating action button (FAB) adds a new word.



Task 1: Create a new project and dataset

Before you can display a `RecyclerView`, you need data to display. In this task, you will create a new project for the app and a dataset. In a more sophisticated app, your data might come from internal storage (a file, SQLite database, saved preferences), from another app (Contacts, Photos), or from the internet (cloud storage, Google Sheets, or any data source with an API). Storing and retrieving data is a topic of its own covered in the data storage chapter. For this exercise, you will simulate data by creating it in the `onCreate()` method of `MainActivity`.

1.1. Create the project and layout

1. Start Android Studio.
2. Create a new project with the name **RecyclerView**, select the **Basic Activity** template, and generate the layout file.

   The Basic Activity template, introduced in the chapter on using clickable images, provides a floating action button (FAB) and app bar in the `Activity` layout (`activity_main.xml`), and a layout for the `Activity` content (`content_main.xml`).

3. Run your app. You should see the RecyclerView app title and "Hello World" on the screen.

   If you encounter Gradle-related errors, sync your project as described in the practical on installing Android Studio and running Hello World.

1.2. Add code to create data

In this step you create a `LinkedList` of 20 word strings that end in increasing numbers, as in ["Word 1", "Word 2", "Word 3", ...].

1. Open **MainActivity** and add a private member variable for the `mWordList` linked list.

```
public class MainActivity extends AppCompatActivity {
    private final LinkedList<String> mWordList = new LinkedList<>();
    // ... Rest of MainActivity code ...
}
```

2. Add code within the `onCreate()` method that populates `mWordList` with words:

```
// Put initial data into the word list.
for (int i = 0; i < 20; i++) {
            mWordList.addLast("Word " + i);
}
```

The code concatenates the string `"Word "` with the value of `i` while increasing its value. This is all you need as a dataset for this exercise.

1.3. Change the FAB icon

For this practical, you will use a FAB to generate a new word to insert into the list. The Basic Activity template provides a FAB, but you may want to change its icon. As you learned in another lesson, you can choose an icon from the set of icons in Android Studio for the FAB. Follow these steps:

1. Expand **res** in the **Project > Android** pane, and right-click (or Control-click) the **drawable** folder.
2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.
3. Choose **Action Bar and Tab Items** in the drop-down menu at the top of the dialog.
4. Change **ic_action_name** in the **Name** field to **ic_add_for_fab**.
5. Click the clip art image (the Android logo next to **Clipart:**) to select a clip art image as the icon. A page of icons appears. Click the icon you want to use for the FAB, such as the plus (**+**) sign.
6. Choose **HOLO_DARK** from the **Theme** drop-down menu. This sets the icon to be white against a dark-colored (or black) background. Click **Next**.
7. Click **Finish** in the Confirm Icon Path dialog.

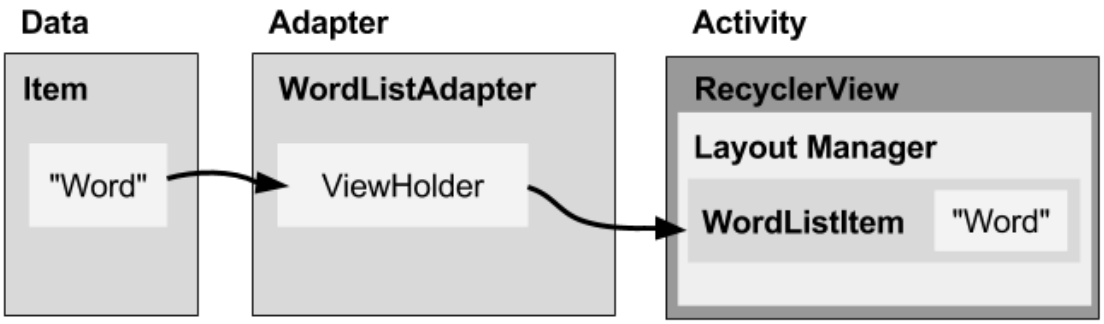
Tip: For a complete description of adding an icon, see Create app icons with Image Asset Studio.

Task 2: Create a RecyclerView

In this practical, you display data in a `RecyclerView`. You need the following:

- Data to display: Use the `mWordList`.
- A `RecyclerView` for the scrolling list that contains the list items.
- Layout for one item of data. All list items look the same.
- A layout manager. `RecyclerView.LayoutManager` handles the hierarchy and layout of `View` elements. `RecyclerView` requires an explicit layout manager to manage the arrangement of list items contained within it. This layout could be vertical, horizontal, or a grid. You will use a vertical `LinearLayoutManager`.
- An adapter. `RecyclerView.Adapter` connects your data to the `RecyclerView`. It prepares the data in a `RecyclerView.ViewHolder`. You will create an adapter that inserts into and updates your generated words in your views.
- A `ViewHolder`. Inside your adapter, you will create a `ViewHolder` that contains the `View` information for displaying one item from the item's layout.

The diagram below shows the relationship between the data, the adapter, the `ViewHolder`, and the layout manager.



To implement these pieces, you will need to:

1. Add a `RecyclerView` element to the `MainActivity` XML content layout (`content_main.xml`) for the RecyclerView app.
2. Create an XML layout file (`wordlist_item.xml`) for one list item, which is `WordListItem`.
3. Create an adapter (`WordListAdapter`) with a `ViewHolder` (`WordViewHolder`). Implement the method that takes the data, places it in the `ViewHolder`, and lets the layout manager know to display it.
4. In the `onCreate()` method of `MainActivity`, create a `RecyclerView` and initialize it with the adapter and a standard layout manager.

Let's do these one at a time.

2.1. Modify the layout in content_main.xml

To add a `RecyclerView` element to the XML layout, follow these steps:

1. Open **content_main.xml** in your RecyclerView app. It shows a "Hello World" `TextView` at the center of a `ConstraintLayout`.
2. Click the **Text** tab to show the XML code.
3. Replace the entire `TextView` element with the following:

```
<android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
</android.support.v7.widget.RecyclerView>
```

You need to specify the full path (`android.support.v7.widget.RecyclerView`), because `RecyclerView` is part of the Support Library.

2.2. Create the layout for one list item

The adapter needs the layout for one item in the list. All the items use the same layout. You need to specify that list item layout in a separate layout resource file, because it is used by the adapter, separately from the `RecyclerView`.

Create a simple word item layout using a vertical `LinearLayout` with a `TextView`:

1. Right-click the **app > res > layout** folder and choose **New > Layout resource file**.
2. Name the file **wordlist_item** and click **OK**.
3. In the new layout file, click the **Text** tab to show the XML code.
4. Change the `ConstraintLayout` that was created with the file to a `LinearLayout` with the following attributes (extract resources as you go):

LinearLayout attribute	**Value**
`android:layout_width`	`"match_parent"`
`android:layout_height`	`"wrap_content"`
`android:orientation`	`"vertical"`
`android:padding`	`"6dp"`

5. Add a `TextView` for the word to the `LinearLayout`. Use `word` as the ID of the word:

Attribute	**Value**
`android:id`	`"@+id/word"`
`android:layout_width`	`"match_parent"`
`android:layout_height`	`"wrap_content"`
`android:textSize`	`"24sp"`
`android:textStyle`	`"bold"`

2.3 Create a style from the TextView attributes

You can use styles to allow elements to share groups of display attributes. An easy way to create a style is to extract the style of a UI element that you already created. To extract the style information for the `word TextView` in `wordlist_item.xml`:

1. Open **wordlist_item.xml** if it is not already open.
2. Right-click (or Control-click) the `TextView` you just created in `wordlist_item.xml`, and choose **Refactor > Extract > Style**. The **Extract Android Style** dialog appears.
3. Name your style **word_title** and leave all other options selected. Select the **Launch 'Use Style Where Possible**' option. Then click **OK**.
4. When prompted, apply the style to the **Whole Project**.
5. Find and examine the `word_title` style in **values > styles.xml**.
6. Reopen **wordlist_item.xml** if it is not already open. The `TextView` now uses the style in place of individual styling properties, as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="6dp">

    <TextView
        android:id="@+id/word"
        style="@style/word_title" />

</LinearLayout>
```

2.4. Create an adapter

Android uses adapters (from the `Adapter` class) to connect data with `View` items in a list. There are many different kinds of adapters available, and you can also write custom adapters. In this task you will create an adapter that associates your list of words with word list `View` items.

To connect data with `View` items, the adapter needs to know about the `View` items. The adapter uses a `ViewHolder` that describes a `View` item and its position within the `RecyclerView`.

First, you will build an adapter that bridges the gap between the data in your word list and the `RecyclerView` that displays it:

1. Right-click **java/com.android.example.recyclerview** and select **New > Java Class**.
2. Name the class **WordListAdapter**.
3. Give `WordListAdapter` the following signature:

```
public class WordListAdapter extends
    RecyclerView.Adapter<WordListAdapter.WordViewHolder>  {}
```

`WordListAdapter` extends a generic adapter for `RecyclerView` to use a `View` holder that is specific for your app and defined inside `WordListAdapter`. `WordViewHolder` shows an error, because you have not yet defined it.

4. Click the class declaration (**WordListAdapter**), then click the red light bulb on the left side of the pane. Choose **Implement methods**.

   A dialog appears that asks you to choose which methods to implement. Choose all three methods and click **OK**.

   Android Studio creates empty placeholders for all the methods. Note how `onCreateViewHolder` and `onBindViewHolder` both reference the `WordViewHolder`, which hasn't been implemented yet.

2.5 Create the ViewHolder for the adapter

To create the `ViewHolder`, follow these steps:

1. Inside the `WordListAdapter` class, add a new `WordViewHolder` inner class with this signature:

```
class WordViewHolder extends RecyclerView.ViewHolder {}
```

You will see an error about a missing default constructor. You can see details about the errors by hovering your mouse cursor over the red-underlined code or over any red horizontal line on the right margin of the editor pane.

2. Add variables to the `WordViewHolder` inner class for the `TextView` and the adapter:

```
public final TextView wordItemView;
final WordListAdapter mAdapter;
```

3. In the inner class `WordViewHolder`, add a constructor that initializes the `ViewHolder` `TextView` from the `word` XML resource, and sets its adapter:

```
public WordViewHolder(View itemView, WordListAdapter adapter) {
   super(itemView);
   wordItemView = itemView.findViewById(R.id.word);
   this.mAdapter = adapter;
}
```

4. Run your app to make sure that you have no errors. Your will still see only a blank view.
5. Click the **Logcat** tab to see the **Logcat** pane, and note the `E/RecyclerView: No adapter attached; skipping layout` warning. You will attach the adapter to the `RecyclerView` in another step.

2.6 Storing your data in the adapter

You need to hold your data in the adapter, and `WordListAdapter` needs a constructor that initializes the word list from the data. Follow these steps:

1. To hold your data in the adapter, create a private linked list of strings in `WordListAdapter` and call it `mWordList`.

```
private final LinkedList<String> mWordList;
```

2. You can now fill in the `getItemCount()` method to return the size of `mWordList`:

```
@Override
public int getItemCount() {
   return mWordList.size();
}
```

3. `WordListAdapter` needs a constructor that initializes the word list from the data. To create a `View` for a list item, the `WordListAdapter` needs to inflate the XML for a list item. You use a *layout inflator* for that job. `LayoutInflator` reads a layout XML description and converts it into the corresponding `View` items. Start by creating a member variable for the inflater in `WordListAdapter`:

```
private LayoutInflater mInflater;
```

4. Implement the constructor for `WordListAdapter`. The constructor needs to have a context parameter, and a linked list of words with the app's data. The method needs to instantiate a `LayoutInflator` for `mInflater` and set `mWordList` to the passed in data:

```
public WordListAdapter(Context context,
                               LinkedList<String> wordList) {
   mInflater = LayoutInflater.from(context);
   this.mWordList = wordList;
}
```

5. Fill out the `onCreateViewHolder()` method with this code:

```
@Override
public WordViewHolder onCreateViewHolder(ViewGroup parent,
                                                    int viewType) {
   View mItemView = mInflater.inflate(R.layout.wordlist_item,
                                                    parent, false);
   return new WordViewHolder(mItemView, this);
}
```

The `onCreateViewHolder()` method is similar to the `onCreate()` method. It inflates the item layout, and returns a `ViewHolder` with the layout and the adapter.

6. Fill out the `onBindViewHolder()` method with the code below:

```
@Override
public void onBindViewHolder(WordViewHolder holder, int position) {
   String mCurrent = mWordList.get(position);
   holder.wordItemView.setText(mCurrent);
}
```

The `onBindViewHolder()` method connects your data to the view holder.

7. Run your app to make sure that there are no errors.

2.7. Create the RecyclerView in the Activity

Now that you have an adapter with a `ViewHolder`, you can finally create a `RecyclerView` and connect all the pieces to display your data.

1. Open **MainActivity**.
2. Add member variables for the `RecyclerView` and the adapter.

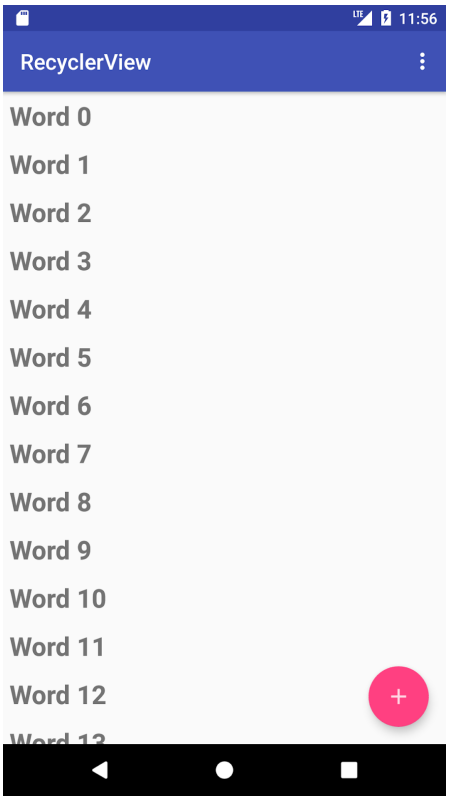
```
private RecyclerView mRecyclerView;
private WordListAdapter mAdapter;
```

3. In the `onCreate()` method of `MainActivity`, add the following code that creates the `RecyclerView` and connects it with an adapter and the data. The comments explain each line. You must insert this code *after* the `mWordList` initialization.

```
// Get a handle to the RecyclerView.
mRecyclerView = findViewById(R.id.recyclerview);
// Create an adapter and supply the data to be displayed.
mAdapter = new WordListAdapter(this, mWordList);
// Connect the adapter with the RecyclerView.
mRecyclerView.setAdapter(mAdapter);
// Give the RecyclerView a default layout manager.
mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
```

4. Run your app.

   You should see your list of words displayed, and you can scroll the list.



Task 3: Make the list interactive

Looking at lists of items is interesting, but it's a lot more fun and useful if your user can interact with them. To see how the `RecyclerView` can respond to user input, you will attach a click handler to each item. When the item is tapped, the click handler is executed, and that item's text will change.

The list of items that a `RecyclerView` displays can also be modified dynamically—it doesn't have to be a static list. There are several ways to add additional behaviors. One is to use the floating action button (FAB). For example, in Gmail, the FAB is used to compose a new email. For this practical, you will generate a new word to insert into the list. For a more useful app, you would get data from your users.

3.1. Make items respond to clicks

1. Open **WordListAdapter**.
2. Change the `WordViewHolder` class signature to implement `View.onClickListener`:

```
class WordViewHolder extends RecyclerView.ViewHolder
                             implements View.OnClickListener {
```

3. Click the class header and on the red light bulb to implement stubs for the required methods, which in this case is just the `onClick()` method.
4. Add the following code to the body of the `onClick()` method.

```
// Get the position of the item that was clicked.
int mPosition = getLayoutPosition();
// Use that to access the affected item in mWordList.
String element = mWordList.get(mPosition);
// Change the word in the mWordList.
mWordList.set(mPosition, "Clicked! " + element);
// Notify the adapter, that the data has changed so it can
// update the RecyclerView to display the data.
mAdapter.notifyDataSetChanged();
```

5. Connect the `onClickListener` with the `View`. Add this code to the `WordViewHolder` constructor (below the `this.mAdapter = adapter` line):

```
itemView.setOnClickListener(this);
```

6. Run your app. Click items to see the text change.

3.2. Add behavior to the FAB

In this task you will implement an action for the FAB to:

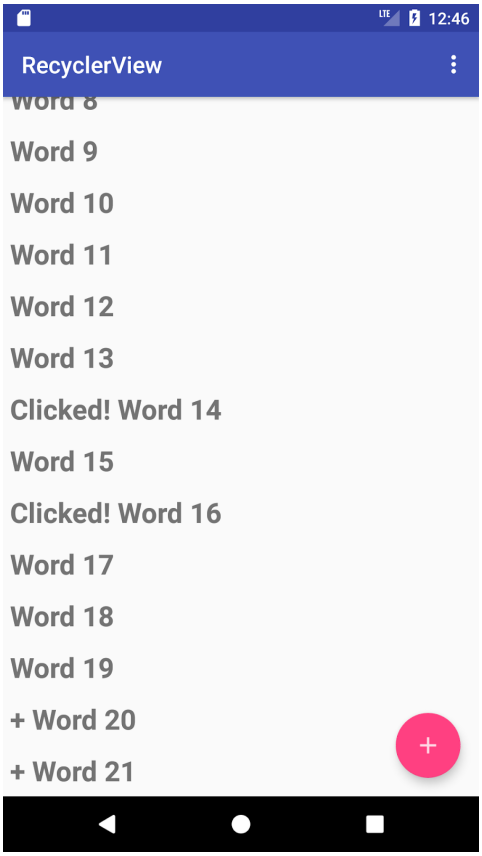
- Add a word to the end of the list of words.
- Notify the adapter that the data has changed.
- Scroll to the inserted item.

Follow these steps:

1. Open **MainActivity**. The `onCreate()` method sets an `OnClickListener()` to the `FloatingActionButton` with an `onClick()` method for taking an action. Change the `onClick()` method to the following:

```
@Override
public void onClick(View view) {
    int wordListSize = mWordList.size();
    // Add a new word to the wordList.
    mWordList.addLast("+ Word " + wordListSize);
    // Notify the adapter, that the data has changed.
    mRecyclerView.getAdapter().notifyItemInserted(wordListSize);
    // Scroll to the bottom.
    mRecyclerView.smoothScrollToPosition(wordListSize);
}
```

2. Run the app.
3. Scroll the list of words and click items.
4. Add items by clicking the FAB.



What happens if you rotate the screen? You will learn in a later lesson how to preserve the state of an app when the screen is rotated.

Solution code

Android Studio project: RecyclerView

Coding challenges

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

Challenge 1: Change the options menu to show only one option: **Reset**. This option should return the list of words to its original state, with nothing clicked and no extra words.

Challenge 2: Creating a click listener for each item in the list is easy, but it can hurt the performance of your app if you have a lot of data. Research how you could implement this more efficiently. This is an advanced challenge. Start by thinking about it conceptually, and then search for an implementation example.

Summary

- `RecyclerView` is a resource-efficient way to display a scrollable list of items.
- To create a `View` for each list item, the adapter inflates an XML layout resource for a list item using `LayoutInflator`.
- `LinearLayoutManager` is a `RecyclerView` layout manager that shows items in a vertical or horizontal scrolling list.
- `GridLayoutManager` is a `RecyclerView` layout manager that shows items in a grid
- `StaggeredGridLayoutManager` is a `RecyclerView` layout manager that shows items in a staggered grid.
- Use `RecyclerView.Adapter` to connect your data to the `RecyclerView`. It prepares the data in a `RecyclerView.ViewHolder` that describes a `View` item and its position within the `RecyclerView`.
- Implement `View.onClickListener` to detect mouse clicks in a `RecyclerView`.

Related concept

The related concept documentation is 4.5: RecyclerView.

Learn more

Android Studio documentation:

- Android Studio User Guide
- Create app icons with Image Asset Studio

Android developer documentation:

- `RecyclerView`
- `LayoutInflator`
- `RecyclerView.LayoutManager`
- `LinearLayoutManager`
- `GridLayoutManager`
- `StaggeredGridLayoutManager`
- `CoordinatorLayout`
- `ConstraintLayout`
- `RecyclerView.Adapter`
- `RecyclerView.ViewHolder`
- `View.onClickListener`
- Create a list with `RecyclerView`

Video:

- RecyclerView Animations and Behind the Scenes (Android Dev Summit 2015)

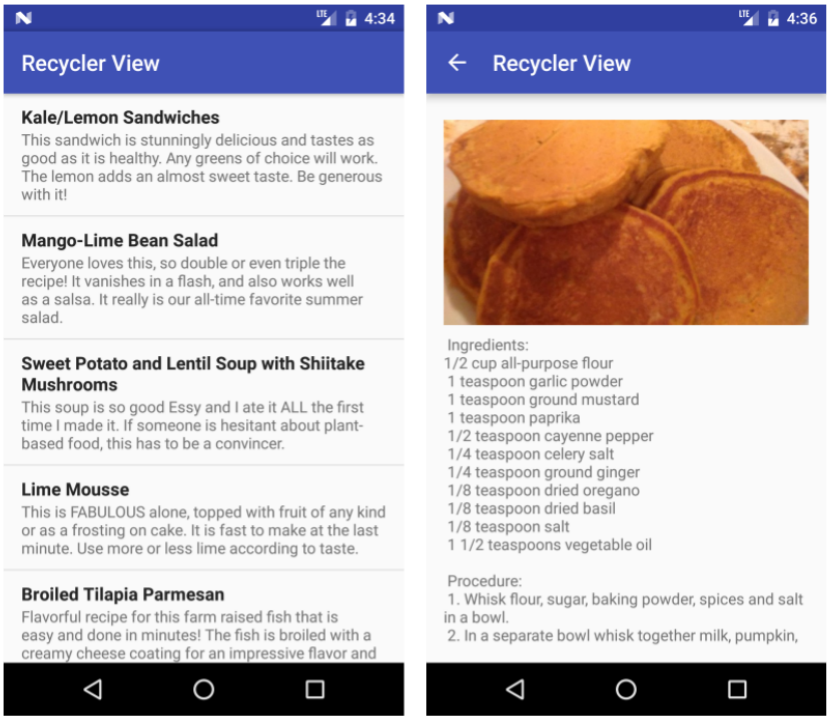
Homework

Build and run an app

Create an app that uses a `RecyclerView` to display a list of recipes. Each list item must show the name of the recipe with a short description. When the user taps a recipe (an item in the list), start an `Activity` that shows the full recipe text.

- Use separate `TextView` elements and styling for the recipe name and description.
- You may use placeholder text for the full recipes.
- As an option, add an image for the finished dish to each recipe.
- Clicking the **Up** button takes the user back to the list of recipes.

The screenshot below shows an example for a simple implementation. Your app can look very different, as long as it has the required functionality.



Answer these questions

Question 1

Which of the following statements about a RecyclerView is *false*? Choose one.

- A `RecyclerView` is a more resource-efficient way to display scrollable lists.
- You need to provide a layout for just one item of the list.
- All list items look the same.
- You don't need a layout manager with a `RecyclerView` to handle the hierarchy and layout of `View` elements.

Question 2

Which of the following is the primary component you need to provide to an adapter a `View` item and its position within a `RecyclerView`? Choose one.

- `RecyclerView`
- `RecyclerView.Adapter`
- `RecyclerView.ViewHolder`
- `AppCompatActivity`

Question 3

Which interface do you need to implement in order to listen and respond to user clicks in a `RecyclerView`? Choose one.

- `View.onClickListener`
- `RecyclerView.Adapter`
- `RecyclerView.ViewHolder`
- `View.OnKeyListener`

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- Implements a `RecyclerView` that shows a scrollable list of recipe titles and short descriptions.
- The code extends or implements `RecyclerView`, `RecyclerView.Adapter`, `RecyclerView.ViewHolder`, and `View.OnClickListener`.
- Clicking a list item starts an `Activity` that shows the full recipe.
- The `AndroidManifest.xml` file defines a parent relationship so that clicking the **Up** button in a recipe view goes back to the list of recipes.
- `ViewHolder` contains a layout with two `TextView` elements; for example, a `LinearLayout` with two `TextView` elements.

Lesson 5.1: Drawables, styles, and themes

Introduction

In this chapter you learn how to apply common styles to your views, use drawable resources, and apply themes to your app. These practices reduce your code and make your code easier to read and maintain.

What you should already know

You should be able to:

- Create and run apps in Android Studio.

- Create and edit UI elements using the layout editor.
- Edit XML layout code, and access elements from your Java code.
- Extract hardcoded strings into string resources.
- Extract hardcoded dimensions into dimension resources.
- Add menu items and icons to the options menu in the app bar.
- Create a click handler for a `Button` click.
- Display a `Toast` message.
- Pass data from one `Activity` to another using an `Intent`.

What you'll learn

You will learn how to:

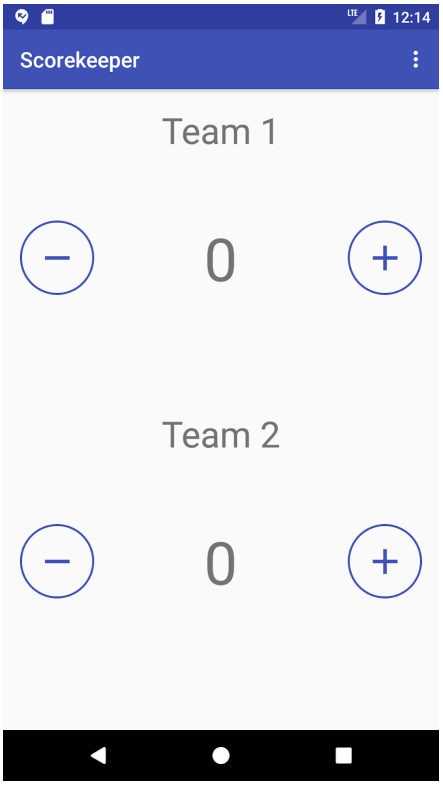
- Define a style resource.
- Apply a style to a `View`.
- Apply a theme to an `Activity` or app in XML and programmatically.
- Use `Drawable` resources.

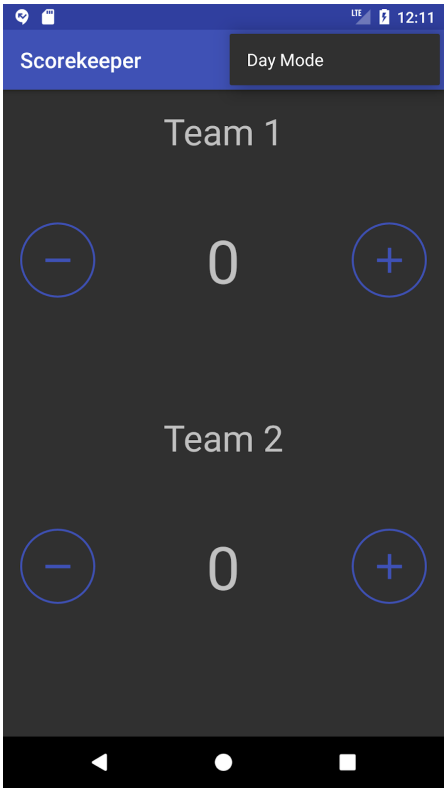
What you'll do

- Create a new app and add `Button` and `TextView` elements to the layout.
- Create `Drawable` resources in XML and use them as backgrounds for your `Button` elements.
- Apply styles to UI elements.
- Add a menu item that changes the theme of the app to a low contrast "night mode."

App overview

The Scorekeeper app consists of two sets of `Button` elements and two `TextView` elements, which are used to keep track of the score for any point-based game with two players.





Task 1: Create the Scorekeeper app

In this section, you will create your Android Studio project, modify the layout, and add the `android:onClick` attribute to its `Button` elements.

1.1 Create the project

1. Start Android Studio and create a new Android Studio project with the name **Scorekeeper**.
2. Accept the default minimum SDK and choose the **Empty Activity** template.
3. Accept the default `Activity` name (`MainActivity`), and make sure the **Generate Layout file** and **Backwards Compatibility (AppCompat)** options are checked.

4. Click **Finish**.

1.2 Create the layout for MainActivity

The first step is to change the layout from `ConstraintLayout` to `LinearLayout`:

1. Open **activity_main.xml** and click the **Text** tab to see the XML code. At the top, or *root*, of the `View` hierarchy is the `ConstraintLayout` `ViewGroup`:

```
android.support.constraint.ConstraintLayout
```

2. Change this `ViewGroup` to `LinearLayout`. The second line of code now looks something like this:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

3. Delete the following line of XML code, which is related to `ConstraintLayout`:

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

The block of XML code at the top should now look like this:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
```

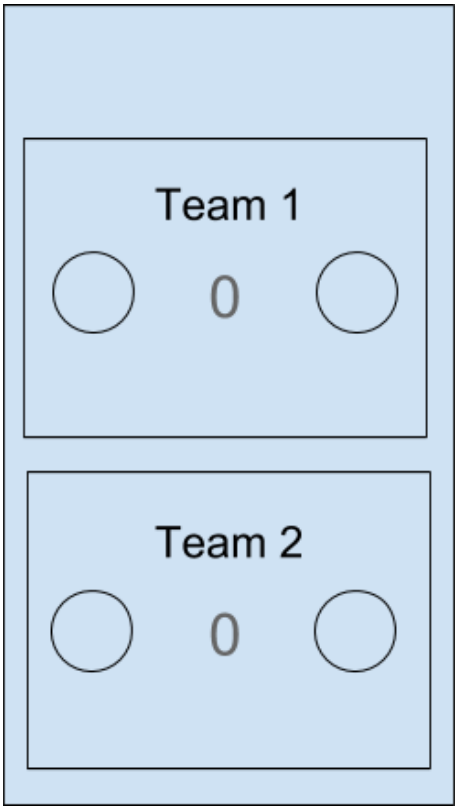
```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.android.scorekeeper.MainActivity">
```

4. Add the following attributes (without removing the existing attributes):

Attribute	**Value**
`android:orientation`	`"vertical"`
`android:padding`	`"16dp"`

1.3 Create the score containers

The layout for this app includes score containers defined by two `RelativeLayout` view group elements—one for each team. The following diagram shows the layout in its simplest form.



1. Inside the `LinearLayout`, add two `RelativeLayout` elements with the following attributes:

RelativeLayout attribute	**Value**
`android:layout_width`	`"match_parent"`
`android:layout_height`	`"0dp"`
`android:layout_weight`	`"1"`

You may be surprised to see that the `layout_height` attribute is set to 0dp. This is because we are using the `layout_weight` attribute to determine how much space these `RelativeLayout` elements take up in the parent `LinearLayout`.

2. Add two `ImageButton` elements to each `RelativeLayout`: one for decreasing the score, and one for increasing the score. Use these attributes:

ImageButton attribute	**Value**
`android:id`	`"@+id/decreaseTeam1"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_alignParentLeft`	`"true"`
`android:layout_alignParentStart`	`"true"`
`android:layout_centerVertical`	`"true"`

Use `increaseTeam1` as the `android:id` for the second `ImageButton` that increases the score. Use `decreaseTeam2` and `increaseTeam2` for the third and fourth `ImageButton` elements.

3. Add a `TextView` to each `RelativeLayout` between the `ImageButton` elements for displaying the score. Use these attributes:

TextView attribute	**Value**
`android:id`	`"@+id/score_1"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_centerHorizontal`	`"true"`
`android:layout_centerVertical`	`"true"`
`android:text`	`"0"`

Use `score_2` as the `android:id` for the second `TextView` between the `ImageButton` elements.

4. Add another `TextView` to each `RelativeLayout` above the score to represent the Team Names. Use these attributes:

TextView attribute	**Value**
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_alignParentTop`	`"true"`
`android:layout_centerHorizontal`	`"true"`
`android:text`	`"Team 1"`

Use `"Team 2"` as the `android:text` for the second `TextView`.

1.4 Add vector assets

You will use material icons in the Vector Asset Studio for the scoring `ImageButton` elements.

1. Select **File > New > Vector Asset** to open the Vector Asset Studio.
2. Click the icon to open a list of material icon files. Select the **Content** category.
3. Choose the **add** icon and click **OK**.
4. Rename the resource file **ic_plus** and check the **Override** checkbox next to size options.
5. Change the size of the icon to **40dp** x **40dp**.
6. Click **Next** and then **Finish**.
7. Repeat this process to add the **remove** icon and name the file **ic_minus**.

You can now add the icons and content descriptions for the `ImageButton` elements. The content description provides a useful and descriptive label that explains the meaning and purpose of the `ImageButton` to users who may require Android accessibility features such as the Google TalkBack screen reader.

8. Add the following attributes to the `ImageButton` elements on the *left* side of the layout:

```
android:src="@drawable/ic_minus"
android:contentDescription="Minus Button"
```

9. Add the following attributes to the `ImageButton` elements on the *right* side of the layout:

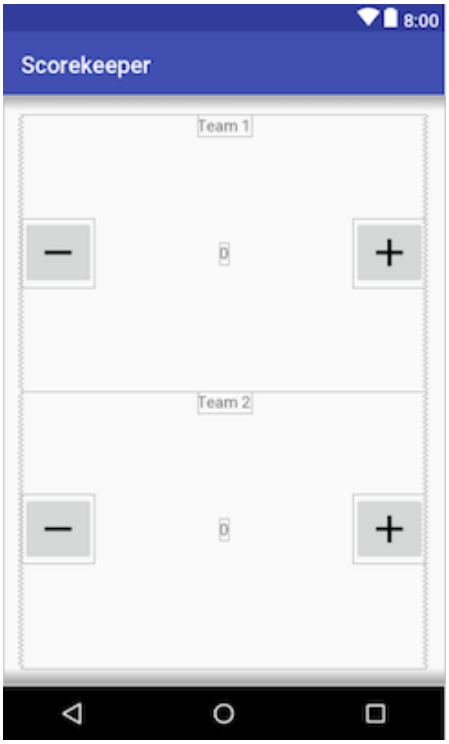
```
android:src="@drawable/ic_plus"
android:contentDescription="Plus Button"
```

10. Extract all of your string resources. This process removes all of your strings from the Java code and puts them in the `strings.xml` file. This allows for your app to be easily localized into different languages.

Solution code for the layout

The following shows the layout for the Scorekeeper app, and the XML code for the layout.

Note: Your code may be a little different, as there are multiple ways to achieve the same layout.



XML code:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context="com.example.android.scorekeeper.MainActivity">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1">

        <TextView
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentTop="true"
            android:layout_centerHorizontal="true"
            android:text="@string/team_1" />

        <ImageButton
            android:id="@+id/decreaseTeam1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentLeft="true"
            android:layout_alignParentStart="true"
            android:layout_centerVertical="true"
            android:src="@drawable/ic_minus"
            android:contentDescription=
                            "@string/minus_button_description" />

        <TextView
            android:id="@+id/score_1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerHorizontal="true"
            android:layout_centerVertical="true"
            android:text="@string/initial_count" />

        <ImageButton
            android:id="@+id/increaseTeam1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentEnd="true"
            android:layout_alignParentRight="true"
            android:layout_centerVertical="true"
            android:src="@drawable/ic_plus"
            android:contentDescription=
                             "@string/plus_button_description" />
    </RelativeLayout>

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1">

        <TextView
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentTop="true"
            android:layout_centerHorizontal="true"
            android:text="@string/team_2" />

        <ImageButton
            android:id="@+id/decreaseTeam2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentLeft="true"
            android:layout_alignParentStart="true"
            android:layout_centerVertical="true"
            android:src="@drawable/ic_minus"
            android:contentDescription=
                                "@string/minus_button_description" />

        <TextView
            android:id="@+id/score_2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerHorizontal="true"
            android:layout_centerVertical="true"
            android:text="@string/initial_count" />

        <ImageButton
            android:id="@+id/increaseTeam2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentEnd="true"
            android:layout_alignParentRight="true"
            android:layout_centerVertical="true"
            android:src="@drawable/ic_plus"
            android:contentDescription=
                                 "@string/plus_button_description" />
    </RelativeLayout>
</LinearLayout>
```

1.5 Initialize the TextView elements and score count variables

In order to keep track of the scores, you will need two things:

- Integer variables to keep track of the scores.
- A reference to each score `TextView` element in `MainActivity` so that you can update the scores.

Follow these steps:

1. Create two integer member variables representing the score of each team.

```
// Member variables for holding the score
private int mScore1;
private int mScore2;
```

2. Create two `TextView` member variables to hold references to the `TextView` elements.

```
// Member variables for holding the score
private int mScore1;
private int mScore2;
```

3. In the `onCreate()` method of `MainActivity`, find your score `TextView` elements by id and assign them to the member variables.

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

        //Find the TextViews by ID
        mScoreText1 = (TextView)findViewById(R.id.score_1);
```

```
        mScoreText2 = (TextView)findViewById(R.id.score_2);
}
```

1.6 Implement the click handlers for the ImageButton elements

You will add the `android:onClick` attribute to each `ImageButton`, and create two click handler methods in `MainActivity`. The left `ImageButton` elements should decrement the score `TextView`, while the right ones should increment it.

1. Open **activity_main.xml** if it is not already open, and add the following `android:onClick` attribute to the first `ImageButton` on the *left* side of the layout:

```
android:onClick="decreaseScore"
```

2. The `decreaseScore` method name is underlined in red. Click the red bulb icon in the left margin, or click the method name and press **Option-Return**, and choose **Create 'decreaseScore(view)' in 'MainActivity'**. Android Studio creates the `decreaseScore()` method stub in `MainActivity`.

3. Add the above `android:onClick` attribute to the second `ImageButton` on the left side of the layout. This time the method name is valid, because the stub has already been created.

4. Add the following `android:onClick` attribute to each `ImageButton` on the *right* side of the layout:

```
android:onClick="increaseScore"
```

5. The `increaseScore` method name is underlined in red. Click the red bulb icon in the left margin, or click the method name and press **Option-Return**, and choose **Create**

'increaseScore(view)' in 'MainActivity'. Android Studio creates the `increaseScore()` method stub in `MainActivity`.

6. Add code to the stub methods to decrease and increase the score as shown below. Both methods use `view.getID()` to get the ID of the team's `ImageButton` that was clicked, so that your code can update the proper team.

Solution code for increaseScore() and decreaseScore()

```
/**
* Method that handles the onClick of both the decrement buttons
* @param view The button view that was clicked
*/
public void decreaseScore(View view) {
   // Get the ID of the button that was clicked
   int viewID = view.getId();
   switch (viewID){
       //If it was on Team 1
       case R.id.decreaseTeam1:
           //Decrement the score and update the TextView
           mScore1--;
           mScoreText1.setText(String.valueOf(mScore1));
           break;
       //If it was Team 2
       case R.id.decreaseTeam2:
           //Decrement the score and update the TextView
           mScore2--;
           mScoreText2.setText(String.valueOf(mScore2));
   }
}

/**
* Method that handles the onClick of both the increment buttons
* @param view The button view that was clicked
*/
public void increaseScore(View view) {
   //Get the ID of the button that was clicked
   int viewID = view.getId();
```

```
    switch (viewID){
        //If it was on Team 1
        case R.id.increaseTeam1:
            //Increment the score and update the TextView
            mScore1++;
            mScoreText1.setText(String.valueOf(mScore1));
            break;
        //If it was Team 2
        case R.id.increaseTeam2:
            //Increment the score and update the TextView
            mScore2++;
            mScoreText2.setText(String.valueOf(mScore2));
    }
}
```

Task 2: Create a drawable resource

You now have a functioning Scorekeeper app! However, the layout is dull and does not communicate the function of the `ImageButton` elements. In order to make it clearer, the standard grey background of the buttons can be changed.

In Android, graphics are often handled by a resource called a `Drawable`. In the following exercise you will learn how to create a certain type of `Drawable` called a `ShapeDrawable`, and apply it to your `ImageButton` elements as a background.

For more information on `Drawables`, see Drawable Resources.

2.1 Create a ShapeDrawable

A `ShapeDrawable` is a primitive geometric shape defined in an XML file by a number of attributes including color, shape, padding and more. It defines a vector graphic, which can scale up and down without losing any definition.

1. In the Project > Android pane, **right-click** (or **Control-click**) on the **drawable** folder in the **res** directory.

2. Choose **New > Drawable resource file**.
3. Name the file **button_background** and click **OK**. (Don't change the **Source set** or **Directory name**, and don't add qualifiers). Android Studio creates the file `button_background.xml` in the `drawable` folder.
4. Open **button_background.xml**, click the **Text** tab to edit the XML code, and remove all of the code except:

```
<?xml version="1.0" encoding="utf-8"?>
```

5. Add the following code which creates an oval shape with an outline:

```
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval">
        <stroke
            android:width="2dp"
            android:color="@color/colorPrimary"/>
</shape>
```

2.2 Apply the ShapeDrawable as a background

1. Open **activity_main.xml**.
2. Add the `Drawable` as the background for all four `ImageButton` elements:

```
android:background="@drawable/button_background"
```

3. Click the **Preview** tab in the layout editor to see that the background automatically scales to fit the size of the `ImageButton`.

4. In order to render each `ImageButton` properly on all devices, you will change the `android:layout_height` and `android:layout_width` attributes for each `ImageButton` to `70dp`, which is a good size on most devices. Change the first attribute for the first `ImageButton` as follows:

```
android:layout_width="70dp"
```

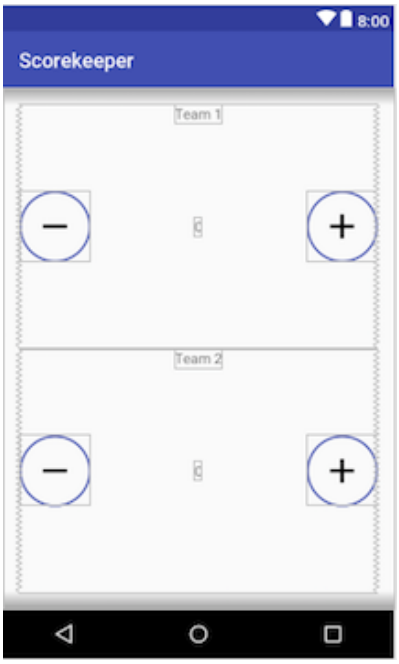
5. Click once on `"70dp"`, press **Alt-Enter** in Windows or **Option-Enter** in macOS, and choose **Extract dimension resource** from the popup menu.
6. Enter **button_size** for the **Resource name**.
7. Click **OK**. This creates a dimension resource in the `dimens.xml` file (in the `values` folder), and the dimension in your code is replaced with a reference to the resource:

```
@dimen/button_size
```

8. Change the `android:layout_height` and `android:layout_width` attributes for each `ImageButton` element using the new dimension resource:

```
android:layout_width="@dimen/button_size"
android:layout_height="@dimen/button_size"
```

9. Click the **Preview** tab in the layout editor to see the layout. The `ShapeDrawable` is now used for the `ImageButton` elements.



Task 3: Style your View elements

As you continue to add `View` elements and attributes to your layout, your code will start to become large and repetitive, especially when you apply the same attributes to many similar elements. A style can specify common properties such as padding, font color, font size, and background color. Attributes that are layout-oriented such as height, width and relative location should remain in the layout resource file.

In the following exercise, you will learn how to create styles and apply them to multiple `View` elements and layouts, allowing common attributes to be updated simultaneously from one location.

Note: Styles are meant for attributes that modify the look of the `View`. Layout parameters such as height, weight, and relative location should stay in the layout file.

3.1 Create Button styles

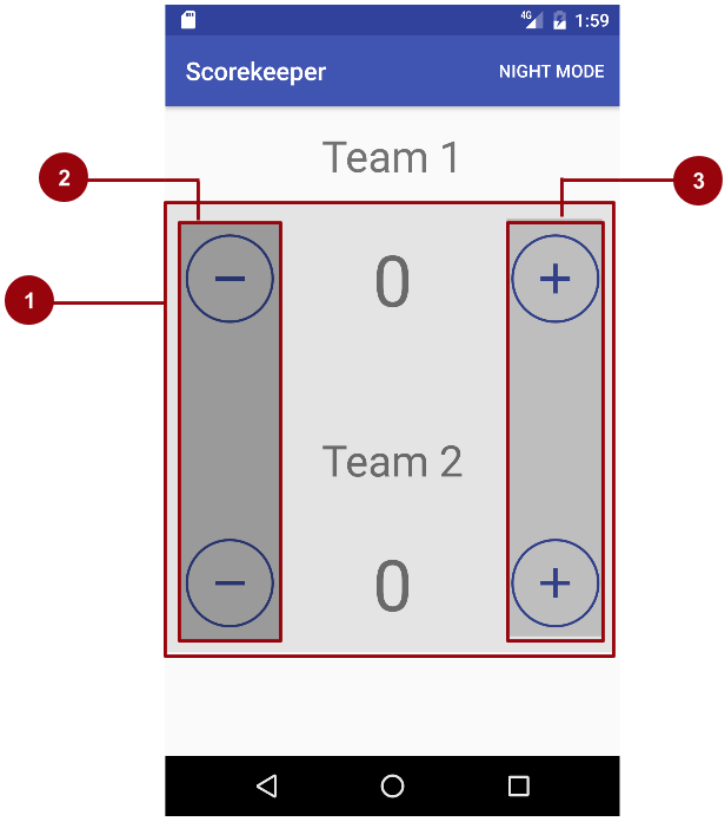
In Android, styles can inherit properties from other styles. You can declare a parent for your style using an optional `parent` parameter and has the following properties:

- Any style attributes defined by the parent style are automatically included in the child style.
- A style attribute defined in both the parent and child style uses the child style's definition (the child overrides the parent).
- A child style can include additional attributes that the parent does not define.

For example, all four `ImageButton` elements in the Scorekeeper app share a common background `Drawable` but with different icons for plus (increase the score) and minus (decrease the score). Furthermore, the two plus `ImageButton` elements share the same icon, as do the two minus `ImageButton` elements. You can therefore create three styles:

1. A style for all of the `ImageButton` elements, which includes the default properties of an `ImageButton` and also the `Drawable` background.
2. A style for the minus `ImageButton` elements. This style inherits the attributes of the `ImageButton` style and includes the minus icon.
3. A style for the plus `ImageButton` elements. This style inherits from the `ImageButton` style and includes the plus icon.

These styles are represented in the figure below.



Do the following:

1. Expand **res > values** in the Project > Android pane, and open the **styles.xml** file.

   This is where all of your style code will be located. The `"AppTheme"` style is always automatically added, and you can see that it extends from `Theme.AppCompat.Light.DarkActionBar`.

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
```

Note the `parent` attribute, which is how you specify your parent style using XML.

The `name` attribute, in this case `AppTheme`, defines the name of the style. The parent attribute, in this case `Theme.AppCompat.Light.DarkActionBar`, declares the parent style attributes that `AppTheme` inherits. In this case it is the Android default theme, with a light background and a dark action bar.

A theme is a style that's applied to an entire `Activity` or app instead of a single `View`. Using a theme creates a consistent style throughout an entire `Activity` or app—for example, a consistent look and feel for the app bar in every part of your app.

2. In between the `<resources>` tags, add a new style with the following attributes to create a common style for all buttons:

```
<style name="ScoreButtons" parent="Widget.AppCompat.Button">
    <item name="android:background">@drawable/button_background</item>
</style>
```

The above snippet sets the parent style to `Widget.AppCompat.Button` to retain the default attributes of a `Button`. It also adds an attribute that changes the background of the `Drawable` to the one you created in the previous task.

1. Create the style for the plus buttons by extending the `ScoreButtons` style:

```
<style name="PlusButtons" parent="ScoreButtons">
    <item name="android:src">@drawable/ic_plus</item>
    <item name=
        "android:contentDescription">@string/plus_button_description</item>
</style>
```

The `contentDescription` attribute is for visually impaired users. It acts as a label that certain accessibility devices use to read out loud to provide some context about the meaning of the UI element.

1. Create the style for the minus buttons:

```
<style name="MinusButtons" parent="ScoreButtons">
    <item name="android:src">@drawable/ic_minus</item>
    <item name=
        "android:contentDescription">@string/minus_button_description</item>
</style>
```

1. You can now use these styles to replace specific style attributes of the `ImageButton` elements. Open the **activity_main.xml** layout file for `MainActivity`, and replace the following attributes for both of the minus `ImageButton` elements:

Delete these attributes	**Add this attribute**
`android:src`	`style="@style/MinusButtons`
`android:contentDescription`	
`android:background`	

Note: The `style` attribute does not use the `android:` namespace because `style` is part of the default XML attributes.

2. Replace the following attributes for both of the plus `ImageButton` elements:

Delete these attributes	**Add this attribute**
`android:src`	`style="@style/PlusButtons`
`android:contentDescription`	

`android:background`	

3.2 Create TextView styles

The team name and score display `TextView` elements can also be styled since they have common colors and fonts. Do the following:

1. Add the following attribute to all `TextView` elements:

```
android:textAppearance="@style/TextAppearance.AppCompat.Headline"
```

2. **Right-click** (or **Control-click**) anywhere in the first score `TextView` attributes and choose **Refactor > Extract > Style...**
3. Name the style **ScoreText** and check the **textAppearance** box (the attribute you just added) as well as the **Launch 'Use Styles Where Possible' refactoring after the style is extracted** checkbox. This will scan the layout file for views with the same attributes and apply the style for you. Do *not* extract the attributes that are related to the layout—click the other checkboxes to turn them off.
4. Choose **OK.**
5. Make sure the scope is set to the **activity_main.xml** layout file and click **OK.**
6. A pane at the bottom of Android Studio will open if the same style is found in other views. Select **Do Refactor** to apply the new style to the views with the same attributes.
7. Run your app. There should be no change except that all of your styling code is now in your resources file and your layout file is shorter.

Layout and style solution code

The following are code snippets for the layout and styles.

styles.xml:

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

   <style name="ScoreButtons" parent="AppTheme">
       <item
         name="android:background">@drawable/button_background</item>
```

```
    </style>

    <style name="PlusButtons" parent="ScoreButtons">
        <item name="android:src">@drawable/ic_plus</item>
        <item name=
          "android:contentDescription">@string/plus_button_description</item>
    </style>

    <style name="MinusButtons" parent="ScoreButtons">
        <item name="android:src">@drawable/ic_minus</item>
        <item name=
           "android:contentDescription">@string/minus_button_description</item>
    </style>

    <style name="ScoreText">
        <item name=
"android:textAppearance">@style/TextAppearance.AppCompat.Headline</item>
    </style>
</resources>
```

activity_main.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context="com.example.android.scorekeeper.MainActivity">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentTop="true"
            android:layout_centerHorizontal="true"
            android:text="@string/team_1"
```

```
            style="@style/ScoreText" />

        <ImageButton
            android:id="@+id/decreaseTeam1"
            android:layout_width="@dimen/button_size"
            android:layout_height="@dimen/button_size"
            android:layout_alignParentLeft="true"
            android:layout_alignParentStart="true"
            android:layout_centerVertical="true"
            style="@style/MinusButtons"
            android:onClick="decreaseScore"/>

        <TextView
            android:id="@+id/score_1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerHorizontal="true"
            android:layout_centerVertical="true"
            android:text="@string/initial_count"
            style="@style/ScoreText" />

        <ImageButton
            android:id="@+id/increaseTeam1"
            android:layout_width="@dimen/button_size"
            android:layout_height="@dimen/button_size"
            android:layout_alignParentEnd="true"
            android:layout_alignParentRight="true"
            android:layout_centerVertical="true"
            style="@style/PlusButtons"
            android:onClick="increaseScore"/>

    </RelativeLayout>

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentTop="true"
            android:layout_centerHorizontal="true"
```

```
            android:text="@string/team_2"
            style="@style/ScoreText" />

        <ImageButton
            android:id="@+id/decreaseTeam2"
            android:layout_width="@dimen/button_size"
            android:layout_height="@dimen/button_size"
            android:layout_alignParentLeft="true"
            android:layout_alignParentStart="true"
            android:layout_centerVertical="true"
            style="@style/MinusButtons"
            android:onClick="decreaseScore"/>

        <TextView
            android:id="@+id/score_2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerHorizontal="true"
            android:layout_centerVertical="true"
            android:text="@string/initial_count"
            style="@style/ScoreText" />

        <ImageButton
            android:id="@+id/increaseTeam2"
            android:layout_width="@dimen/button_size"
            android:layout_height="@dimen/button_size"
            android:layout_alignParentEnd="true"
            android:layout_alignParentRight="true"
            android:layout_centerVertical="true"
            style="@style/PlusButtons"
            android:onClick="increaseScore"/>
    </RelativeLayout>
</LinearLayout>
```

3.3 Updating the styles

The power of using styles becomes apparent when you want to make changes to several elements of the same style. You can make the text bigger, bolder and brighter, and change the icons to the color of the button backgrounds.

1. Open the **styles.xml** file, and add or modify the following attributes for the styles:

Style	Item
`ScoreButtons`	`<item name="android:tint">@color/colorPrimary</item>`
`ScoreText`	`<item name="android:textAppearance">@style/TextAppearance.AppCompat.Display3 </item>`

The `colorPrimary` value is automatically generated by Android Studio when you create the project. You can find it in the **Project > Android** pane in the `colors.xml` file inside the `values` folder. The `TextAppearance.AppCompat.Display3` attribute is a predefined text style supplied by Android.

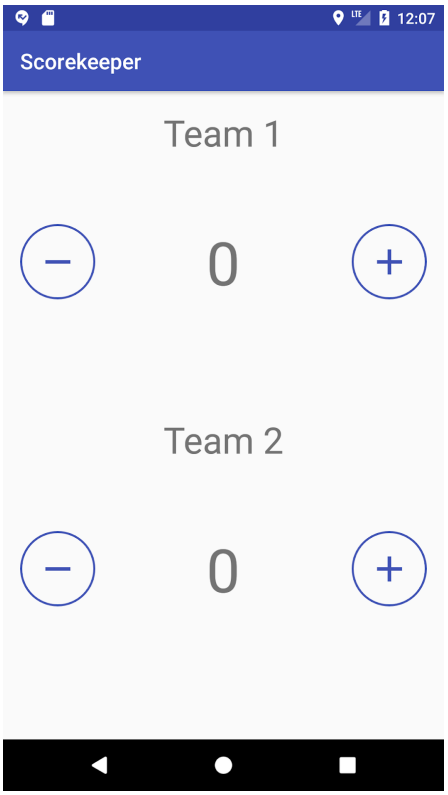
1. Create a new style called **TeamText** with the `textAppearance` attribute set as follows:

```
<style name="TeamText">
   <item name=
    "android:textAppearance">@style/TextAppearance.AppCompat.Display1
   </item>
</style>
```

2. In **activity_main.xml**, change the `style` attribute of the `TextView` team name elements to the newly created `TeamText` style.

```
style="@style/TeamText"
```

3. Run your app. With only these adjustments to the `style.xml` file, all of the views updated to reflect the changes.



Task 4: Themes and final touches

You've seen that `View` elements with similar characteristics can be styled together in the `styles.xml` file. But what if you want to define styles for an entire `Activity`, or the entire app? It's possible to accomplish this by using *themes*. To set a theme for each `Activity`, you need to modify the `AndroidManifest.xml` file.

In this task, you will add the "night mode" theme to your app, which will allow the users to use a low contrast version of your app that is easier on the eyes at night time, as well as make a few polishing touches to the UI.

4.1 Explore themes

1. Open the **AndroidManifest.xml** file, find the `<application>` tag, and change the `android:theme` attribute to:

```
android:theme="@style/Theme.AppCompat.Light.NoActionBar"
```

This is a predefined theme that removes the action bar from your activity.

2. Run your app. The toolbar disappears!
3. Change the theme of the app back to `AppTheme`, which is a child of the `Theme.Appcompat.Light.DarkActionBar` theme as can be seen in `styles.xml`.

To apply a theme to an activity instead of the entire application, place the theme attribute in the `<Activity>` tag instead of the `<application>` tag. For more information on themes and styles, see the Style and Theme Guide.

4.2 Add theme button to the menu

One use for setting a theme for your app is to provide an alternate visual experience for browsing at night. In such conditions, it is often better to have a low contrast, dark layout. The Android framework provides a theme that is designed exactly for this: The `DayNight` theme.

This theme has built-in options that allow you to control the colors in your app programmatically: either by setting it to change automatically by time, or by user command.

In this exercise you will add an options menu item that will toggle the app between the regular theme and a "night-mode" theme.

1. Open **strings.xml** and create two string resources for this options menu item:

```
<string name="night_mode">Night Mode</string>
<string name="day_mode">Day Mode</string>
```

2. **Right-click** (or **Control-click**) on the **res** directory in the **Project > Android** pane, and choose **New > Android resource file**.
3. Name the file **main_menu**, change the **Resource Type** to **Menu**, and click **OK**. The layout editor appears for the `main_menu.xml` file.
4. Click the **Text** tab of the layout editor to show the XML code.
5. Add a menu item with the following attributes:

```
<item
    android:id="@+id/night_mode"
    android:title="@string/night_mode"/>
```

6. Open **MainActivity**, press **Ctrl-O** to open the **Override Method** menu, and select the **onCreateOptionsMenu(menu:Menu):boolean** method located under the **android.app.Activity** category.
7. Click **OK.** Android Studio creates the `onCreateOptionsMenu()` method stub with `return super.onCreateOptionsMenu(menu)` as its only statement.
1. In `onCreateOptionsMenu()` right before the `return.super` statement, add code to inflate the menu:

```
getMenuInflater().inflate(R.menu.main_menu, menu);
```

4.3 Change the theme from the menu

The `DayNight` theme uses the `AppCompatDelegate` class to set the night mode options in your `Activity`. To learn more about this theme, visit this blog post.

1. In your **styles.xml** file, modify the parent of `AppTheme` to **"Theme.AppCompat.DayNight.DarkActionBar"**.
2. Override the `onOptionsItemSelected()` method in `MainActivity`, and add code to check which menu item was clicked:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   //Check if the correct item was clicked
   if(item.getItemId()==R.id.night_mode){}
       // TODO: Get the night mode state of the app.
   return true;
}
```

3. Replace the `TODO:` comment in the above snippet with code that checks if the night mode is enabled. If enabled, the code changes night mode to the disabled state; otherwise, the code enables night mode:

```
if(item.getItemId()==R.id.night_mode){
    // Get the night mode state of the app.
    int nightMode = AppCompatDelegate.getDefaultNightMode();
    //Set the theme mode for the restarted activity
    if (nightMode == AppCompatDelegate.MODE_NIGHT_YES) {
        AppCompatDelegate.setDefaultNightMode
                          (AppCompatDelegate.MODE_NIGHT_NO);
} else {
   AppCompatDelegate.setDefaultNightMode
                         (AppCompatDelegate.MODE_NIGHT_YES);
}
// Recreate the activity for the theme change to take effect.
recreate();
```

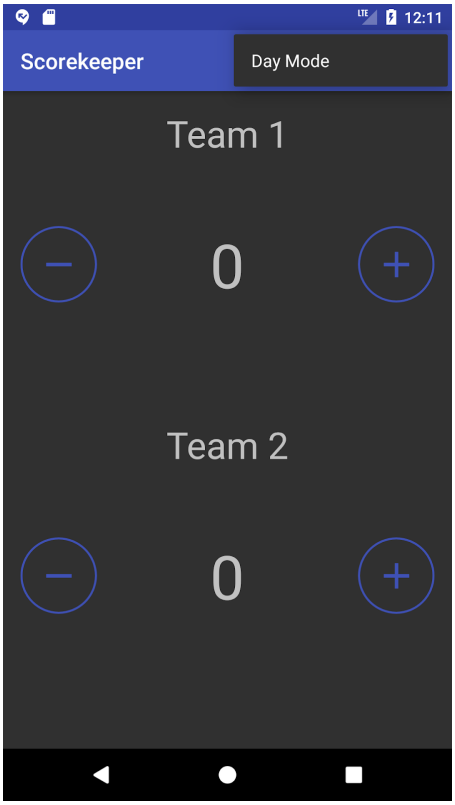
In response to a click on the menu item, the code verifies the current night mode setting by calling `AppCompatDelegate.getDefaultNightMode()`.

The theme can only change while the activity is being created, so the code calls `recreate()` for the theme change to take effect.

4. Run your app. The **Night Mode** menu item should now toggle the theme of your `Activity`.
5. You may notice that the label for your menu item always reads **Night Mode**, which may be confusing to your user if the app is already in the dark theme.
1. Replace the `return super.onCreateOptionsMenu(menu)` statement in the `onCreateOptionsMenu()` method with the following code:

```
// Change the label of the menu based on the state of the app.
int nightMode = AppCompatDelegate.getDefaultNightMode();
if(nightMode == AppCompatDelegate.MODE_NIGHT_YES){
    menu.findItem(R.id.night_mode).setTitle(R.string.day_mode);
} else{
    menu.findItem(R.id.night_mode).setTitle(R.string.night_mode);
}
return true;
```

1. Run your app. The menu item label **Night Mode**, after the user taps it, now changes to **Day Mode** (along with the theme).



4.4 SaveInstanceState

You learned in previous lessons that you must be prepared for your `Activity` to be destroyed and recreated at unexpected times, for example when your screen is rotated. In this app, the `TextView` elements containing the scores are reset to the initial value of 0 when the device is rotated. To fix this, do the following:

1. Open `MainActivity` and add tags under the member variables, which will be used as the keys in `onSaveInstanceState()`:

```
static final String STATE_SCORE_1 = "Team 1 Score";
static final String STATE_SCORE_2 = "Team 2 Score";
```

2. At the end of `MainActivity`, override the `onSaveInstanceState()` method to preserve the values of the two score `TextView` elements:

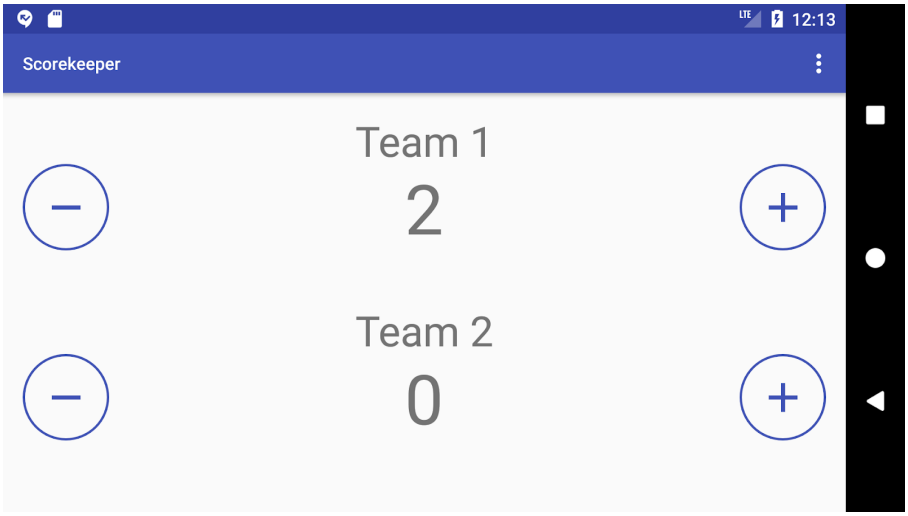
```
@Override
protected void onSaveInstanceState(Bundle outState) {
   // Save the scores.
   outState.putInt(STATE_SCORE_1, mScore1);
   outState.putInt(STATE_SCORE_2, mScore2);
   super.onSaveInstanceState(outState);
}
```

3. At the end of the `onCreate()` method, add code to check if there is a `savedInstanceState`. If there is, restore the scores to the `TextView` elements:

```
if (savedInstanceState != null) {
    mScore1 = savedInstanceState.getInt(STATE_SCORE_1);
    mScore2 = savedInstanceState.getInt(STATE_SCORE_2);

    //Set the score text views
    mScoreText1.setText(String.valueOf(mScore1));
    mScoreText2.setText(String.valueOf(mScore2));
}
```

4. Run the app, and tap the plus `ImageButton` to increase the score. Switch the device or emulator to horizontal orientation to see that the score is retained.



That's it! Congratulations, you now have a styled Scorekeeper app that continues to work if the user changes the device to horizontal or vertical orientation.

Solution code

Android Studio project: Scorekeeper

Coding challenge

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

Challenge: Right now, your buttons do not behave intuitively because they do not change their appearance when they are pressed. Android has another type of `Drawable` called

`StateListDrawable` which allows for a different graphic to be used depending on the state of the object.

For this challenge problem, create a `Drawable` resource that changes the background of the `ImageButton` to the same color as the border when the state of the `ImageButton` is "pressed". You should also set the color of the text inside the `ImageButton` elements to a selector that makes it white when the button is "pressed".

Summary

- `Drawable` elements enhance the look of an app's UI.
- A `ShapeDrawable` is a primitive geometric shape defined in an XML file. The attributes that define a `ShapeDrawable` include color, shape, padding, and more.
- The Android platform supplies a large collection of styles and themes.
- Using styles can reduce the amount of code needed for your UI components.
- A `style` can specify common properties such as height, padding, font color, font size, and background color.
- A `style` should not include layout-related information.
- A `style` can be applied to a `View`, `Activity`, or the entire app. A `style` applied to an `Activity` or the entire app must be defined in the `AndroidManifest.xml` file.
- To inherit a style, a new style identifies a `parent` attribute in the XML.
- When you apply a `style` to a collection of `View` elements in an activity or in your entire app, that is known as a *theme*.
- To apply a theme, you use the `android:theme` attribute.

Related concepts

The related concept documentation is in 5.1: Drawables, styles, and themes.

Learn more

Android Studio documentation:

- Android Studio User Guide
- Add Multi-Density Vector Graphics
- Create App Icons with Image Asset Studio

Android developer documentation:

- Best Practices for User Interface
- Linear Layout
- Drawable Resources
- Styles and Themes
- Buttons
- Layouts
- Support Library
- Screen Compatibility Overview
- Support Different Screen Sizes
- Animate Drawable Graphics
- Loading Large Bitmaps Efficiently
- R.style class of styles and themes
- support.v7.appcompat.R.style class of styles and themes

Material Design:

- Understanding navigation
- App bar

Android Developers Blog: Android Design Support Library

Other:

- Medium: DayNight Theme Guide
- Video: Udacity - Themes and Styles
- Roman Nurik's Android Asset Studio
- Glide documentation

Homework

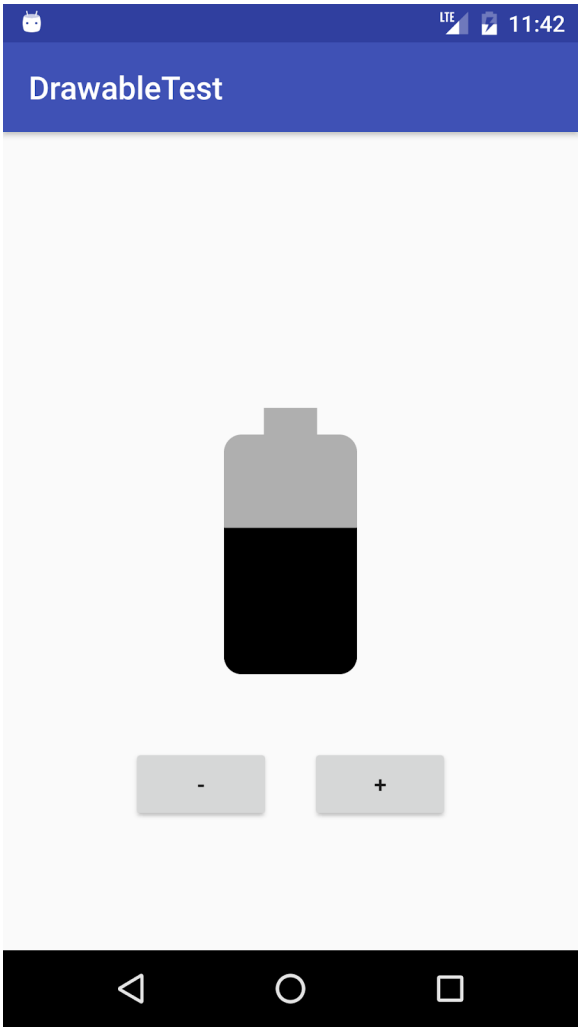
Build and run an app

Create an app that displays an `ImageView` and plus and minus buttons, as shown below. The `ImageView` contains a level list drawable that is a battery level indicator. Tapping the plus or minus button changes the level of the indicator.

Use the battery icons from the Vector Asset Studio to represent 7 different values for the battery level.

The app should have the following properties:

- The plus button increments the level, causing the battery indicator to appear more full.
- The minus button decrements the level, causing the indicator to empty one level.



Answer these questions

Question 1

Which type of `Drawable` do you use to create a `Button` with a background that stretches properly to accommodate the text or image inside the `Button` so that it looks correct for different screen sizes and orientations? Choose one:

- `LevelListDrawable`
- `TransitionDrawable`

- `StateListDrawable`
- `NinePatchDrawable`

Question 2

Which type of `Drawable` do you use to create a `Button` that shows one background when it is pressed and a different background when it is hovered over? Choose one:

- `LevelListDrawable`
- `TransitionDrawable`
- `StateListDrawable`
- `NinePatchDrawable`

Question 3

Suppose you want to create an app that has a white background, dark text, and a dark action bar. Which base style does your application style inherit from? Choose one:

- `Theme.AppCompat.Light`
- `Theme.AppCompat.Dark.NoActionBar`
- `Theme.AppCompat.Light.DarkActionBar`
- `Theme.AppCompat.NoActionBar`
- `Theme.NoActionBar`

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- The buttons increment a count variable. The count variable sets the level on the `ImageView`, using the `setImageLevel()` method.
- The levels in the `LevelListDrawable` go from 0 to 6.

- Before incrementing or decrementing the image level, the `onClick()` methods check to see whether the count variable is within the range of the `LevelListDrawable` (0 to 6). This way, the user can't set a level that doesn't exist.

Lesson 5.2: Cards and colors

Introduction

Google's Material Design guidelines are a series of best practices for creating visually appealing and intuitive applications. In this practical you learn how to add `CardView` and `FloatingActionButton` widgets to your app, how to use images efficiently, and how to employ Material Design best practices to make your user's experience delightful.

What you should already know

You should be able to:

- Create and run apps in Android Studio.
- Create and edit UI elements using the layout editor.
- Edit XML layout code, and access elements from your Java code.
- Create a click handler for a `Button` click.
- Extract text to a string resource and a dimension to a dimension resource.
- Use drawables, styles, and themes.
- Use a `RecyclerView` to display a list.

What you'll learn

- Recommended use of Material Design widgets such as `FloatingActionButton` and `CardView`.

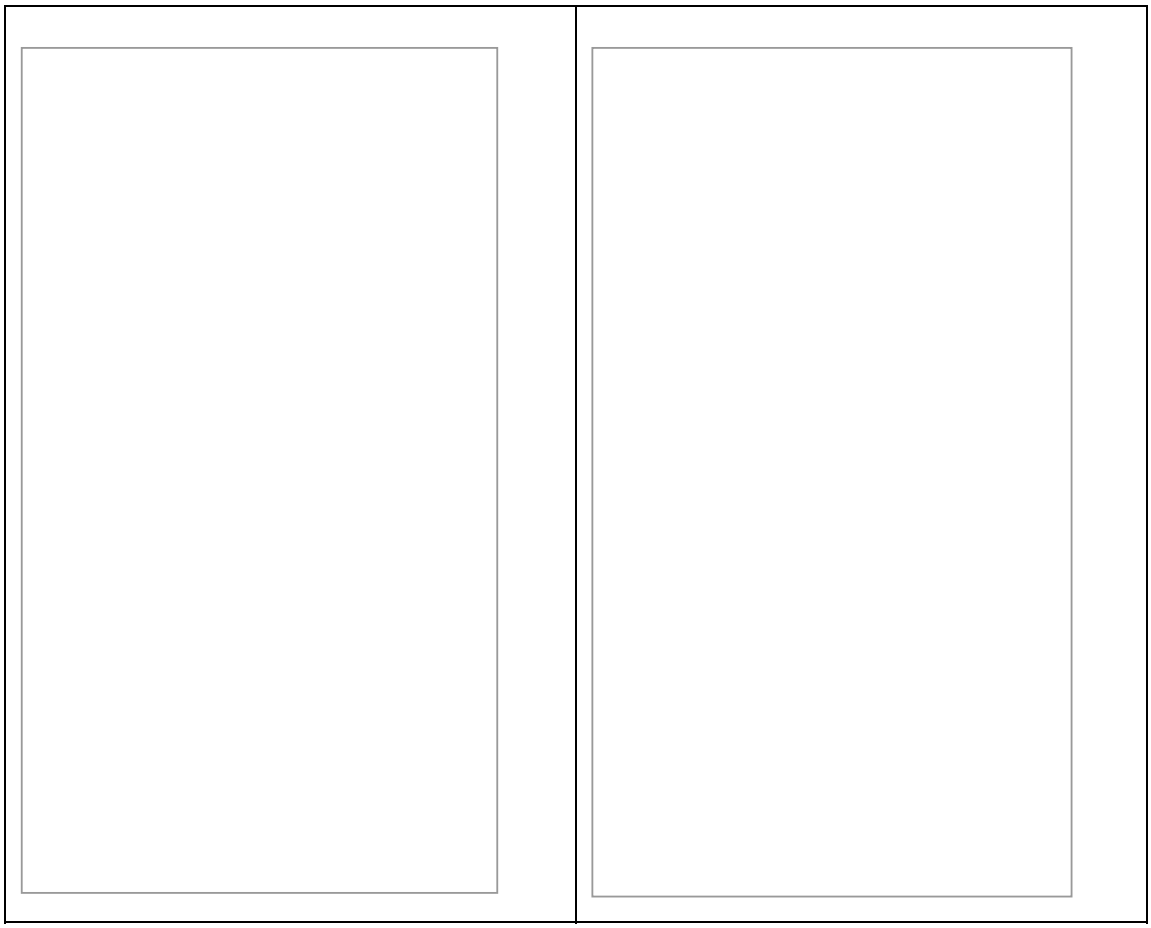
- How to efficiently use images in your app.
- Recommended best practices for designing intuitive layouts using bold colors.

What you'll do

- Modify an app to follow Material Design guidelines.
- Add images and styling to a `RecyclerView` list.
- Implement an `ItemTouchHelper` to add drag-and-drop functionality to your app.

App overview

The MaterialMe app is a mock sports-news app with very poor design implementation. You will fix it up to meet the design guidelines to create a delightful user experience! Below are screenshots of the app before and after the Material Design improvements.



Task 1: Download the starter code

The complete starter app project for this practical is available at MaterialMe-Starter. In this task you will load the project into Android Studio and explore some of the app's key features.

1.1 Open and run the MaterialMe project

1. Download the MaterialMe-Starter code.
2. Open the app in Android Studio.
3. Run the app.

The app shows a list of sports names with some placeholder news text for each sport. The current layout and style of the app makes it nearly unusable: each row of data is not clearly separated and there is no imagery or color to engage the user.

1.2 Explore the app

Before making modifications to the app, explore its current structure. It contains the following elements:

Sport.java

This class represents the data model for each row of data in the `RecyclerView`. Right now it contains a field for the title of the sport and a field for some information about the sport.

SportsAdapter.java

This is the adapter for the `RecyclerView`. It uses an `ArrayList` of Sport objects as its data and populates each row with this data.

MainActivity.java

The `MainActivity` initializes the RecyclerView and adapter, and creates the data from resource files.

strings.xml

This resource file contains all of the data for the app, including the titles and information for each sport.

list_item.xml

This layout file defines each row of the `RecyclerView`. It consists of three `TextView` elements, one for each piece of data (the title and the info for each sport) and one used as a label.

Task 2: Add a CardView and images

One of the fundamental principles of Material Design is the use of bold imagery to enhance the user experience. Adding images to the `RecyclerView` list items is a good start for creating a dynamic and captivating user experience.

When presenting information that has mixed media (like images and text), the Material Design guidelines recommend using a `CardView`, which is a `FrameLayout` with some extra features (such as elevation and rounded corners) that give it a consistent look and feel across many different applications and platforms. `CardView` is a UI component found in the Android Support Libraries.

In this section, you will move each list item into a `CardView` and add an `Image` to make the app comply with Material guidelines.

2.1 Add the CardView

`CardView` is not included in the default Android SDK, so you must add it as a `build.gradle` dependency. Do the following:

1. Open the **build.gradle (Module: app)** file, and add the following line to the `dependencies` section:

```
implementation 'com.android.support:cardview-v7:26.1.0'
```

> The version of the support library may have changed since the writing of this practical. Update the above to the version suggested by Android Studio, and click **Sync** to sync your `build.gradle` files.

2. Open the **list_item.xml** file, and surround the root `LinearLayout` with `android.support.v7.widget.CardView`. Move the schema declaration

(`xmlns:android="http://schemas.android.com/apk/res/android`) to the `CardView`, and add the following attributes:

Attribute	**Value**
`android:layout_width`	`"match_parent"`
`android:layout_height`	`"wrap_content"`
`android:layout_margin`	`"8dp"`

The schema declaration needs to move to the `CardView` because the `CardView` is now the top-level view in your layout file.

3. Choose **Code > Reformat Code** to reformat the XML code, which should now look like this at the beginning and end of the file:

```
<android.support.v7.widget.CardView
     xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">
        <!-- Rest of LinearLayout -->
        <!-- TextView elements -->
    </LinearLayout>
</android.support.v7.widget.CardView>
```

4. Run the app. Now each row item is contained inside a `CardView`, which is elevated above the bottom layer and casts a shadow.

2.2 Download the images

The `CardView` is not intended to be used exclusively with plain text: it is best for displaying a mixture of content. You have a good opportunity to make this app more exciting by adding banner images to every row!

Using images is resource intensive for your app: the Android framework has to load the entire image into memory at full resolution, even if the app only displays a small thumbnail of the image.

In this section you will learn how to use the Glide library to load large images efficiently, without draining your resources or even crashing your app due to 'Out of Memory' exceptions.

1. Download the banner images zip file.
2. Open the **MaterialMe > app > src > main > res** directory in your operating system's file explorer, and create a **drawable** directory, and copy the individual graphics files into the **drawable** directory.
3. You will need an array with the path to each image so that you can include it in the `Sports` object. To do this, define an array that contains all of the paths to the drawables as items in your `string.xml` file. Be sure to that they are in the same order as the `sports_titles` array also defined in the same file:

```
<array name="sports_images">
   <item>@drawable/img_baseball</item>
   <item>@drawable/img_badminton</item>
   <item>@drawable/img_basketball</item>
   <item>@drawable/img_bowling</item>
   <item>@drawable/img_cycling</item>
   <item>@drawable/img_golf</item>
   <item>@drawable/img_running</item>
   <item>@drawable/img_soccer</item>
   <item>@drawable/img_swimming</item>
   <item>@drawable/img_tabletennis</item>
   <item>@drawable/img_tennis</item>
</array>
```

2.3 Modify the Sport object

The `Sport` object will need to include the `Drawable` resource that corresponds to the sport. To achieve that:

1. Add an integer member variable to the `Sport` object that will contain the `Drawable` resource:

```
private final int imageResource;
```

2. Modify the constructor so that it takes an integer as a parameter and assigns it to the member variable:

```
public Sport(String title, String info, int imageResource) {
   this.title = title;
   this.info = info;
   this.imageResource = imageResource;
}
```

3. Create a getter for the resource integer:

```
public int getImageResource() {
   return imageResource;
}
```

2.4 Fix the initializeData() method

In `MainActivity`, the `initializeData()` method is now broken, because the constructor for the `Sport` object demands the image resource as the third parameter.

A convenient data structure to use would be a `TypedArray`. A `TypedArray` allows you to store an array of other XML resources. By using a `TypedArray`, you will be able to obtain the image resources as well as the sports title and information by using indexing in the same loop.

1. In the `initializeData()` method, get the `TypedArray` of resource IDs by calling `getResources().obtainTypedArray()`, passing in the name of the array of `Drawable` resources you defined in your `strings.xml` file:

```
TypedArray sportsImageResources =
        getResources().obtainTypedArray(R.array.sports_images);
```

   You can access an element at index `i` in the `TypedArray` by using the appropriate "get" method, depending on the type of resource in the array. In this specific case, it contains resource IDs, so you use the `getResourceId()` method.

2. Fix the code in the loop that creates the `Sport` objects, adding the appropriate `Drawable` resource ID as the third parameter by calling `getResourceId()` on the `TypedArray`:

```
for(int i=0;i<sportsList.length;i++){
   mSportsData.add(new Sport(sportsList[i],sportsInfo[i],
       sportsImageResources.getResourceId(i,0)));
}
```

3. Clean up the data in the typed array once you have created the `Sport` data `ArrayList`:

```
sportsImageResources.recycle();
```

2.5 Add an ImageView to the list items

1. Change the `LinearLayout` inside the **list_item.xml** file to a `RelativeLayout`, and delete the `android:orientation` attribute.
2. Add an `ImageView` as the first element within the `RelativeLayout` with the following attributes:

Attribute	**Value**
`android:layout_width`	`"match_parent"`
`android:layout_height`	`"wrap_content"`
`android:id`	`"@+id/sportsImage"`
`android:adjustViewBounds`	`"true"`

The `adjustViewBounds` attribute makes the `ImageView` adjust its boundaries to preserve the aspect ratio of the image.

3. Add the following attributes to the `title` `TextView` element:

Attribute	**Value**
`android:layout_alignBottom`	`"@id/sportsImage"`
`android:theme`	`"@style/ThemeOverlay.AppCompat.Dark"`

4. Add the following attributes to the `newsTitle` `TextView` element:

Attribute	**Value**

`android:layout_below`	`"@id/sportsImage"`
`android:textColor`	`"?android:textColorSecondary"`

5. Add the following attributes to the `subTitle` `TextView` element:

Attribute	**Value**
`android:layout_below`	`"@id/newsTitle"`

The question mark in the above `textColor` attribute (`"?android:textColorSecondary"`) means that the framework will apply the value from the currently applied theme. In this case, this attribute is inherited from the `"Theme.AppCompat.Light.DarkActionBar"` theme, which defines it as a light gray color, often used for subheadings.

2.6 Load the images using Glide

After downloading the images and setting up the `ImageView`, the next step is to modify the `SportsAdapter` to load an image into the `ImageView` in `onBindViewHolder()`. If you take this approach, you will find that your app crashes due to "Out of Memory" errors. The Android framework has to load the image into memory each time at full resolution, no matter what the display size of the `ImageView` is.

There are a number of ways to reduce the memory consumption when loading images, but one of the easiest approaches is to use an Image Loading Library like Glide, which you will do in this step. Glide uses background processing, as well some other complex processing, to reduce the memory requirements of loading images. It also includes some useful features like showing placeholder images while the desired images are loaded.

Note: To learn more about reducing memory consumption in your app, see Loading Large Bitmaps Efficiently.

1. Open the **build.gradle (Module: app)** file, and add the following dependency for Glide in the `dependencies` section:

```
implementation 'com.github.bumptech.glide:glide:3.7.0'
```

2. Open **SportsAdapter**, and add a variable in the `ViewHolder` class for the `ImageView`:

```
private ImageView mSportsImage;
```

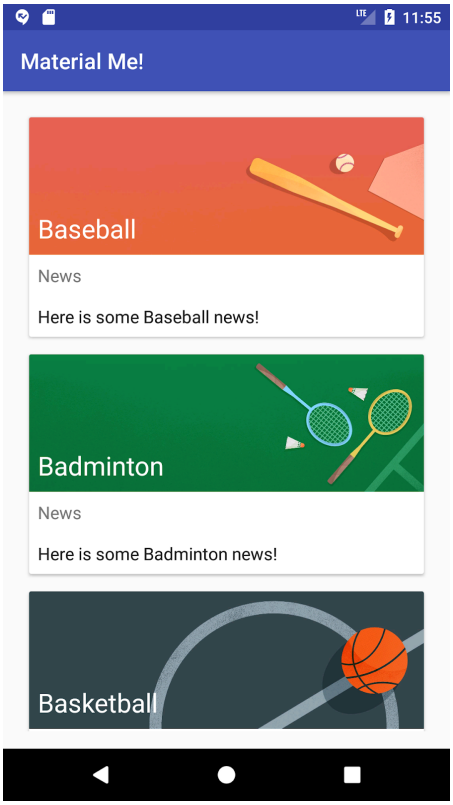
3. Initialize the variable in the `ViewHolder` constructor for the `ViewHolder` class:

```
mSportsImage = itemView.findViewById(R.id.sportsImage);
```

4. Add the following line of code to the `bindTo()` method in the `ViewHolder` class to get the image resource from the `Sport` object and load it into the `ImageView` using Glide:

```
Glide.with(mContext).load(currentSport.getImageResource()).into(mSportsImage
);
```

5. Run the app, your list items should now have bold graphics that make the user experience dynamic and exciting!



That's all takes to load an image with Glide. Glide also has several additional features that let you resize, transform and load images in a variety of ways. Head over to the Glide github page to learn more.

Task 3: Make your CardView swipeable, movable, and clickable

When users see cards in an app, they have expectations about the way the cards behave. The Material Design guidelines say that:

- A card can be dismissed, usually by swiping it away.
- A list of cards can be reordered by holding down and dragging the cards.
- Tapping on card will provide further details.

You will now implement these behaviors in your app.

3.1 Implement swipe to dismiss

The Android SDK includes a class called `ItemTouchHelper` that is used to define what happens to `RecyclerView` list items when the user performs various touch actions, such as swipe, or drag and drop. Some of the common use cases are already implemented in a set of methods in `ItemTouchHelper.SimpleCallback`.

`ItemTouchHelper.SimpleCallback` lets you define which directions are supported for swiping and moving list items, and implement the swiping and moving behavior.

Do the following:

1. Open **MainActivity** and create a new `ItemTouchHelper` object in the `onCreate()` method at the end, below the `initializeData()` method. For its argument, you will create a new instance of `ItemTouchHelper.SimpleCallback`. As you enter **new ItemTouchHelper**, suggestions appear. Select **ItemTouchHelper.SimpleCallback{...}** from the suggestion menu. Android Studio fills in the required methods: `onMove()` and `onSwiped()` as shown below.

```
ItemTouchHelper helper = new ItemTouchHelper(new
                                ItemTouchHelper.SimpleCallback() {
    @Override
    public boolean onMove(RecyclerView recyclerView,
                          RecyclerView.ViewHolder viewHolder,
```

```
                    RecyclerView.ViewHolder target) {
                return false;
            }

    @Override
    public void onSwiped(RecyclerView.ViewHolder viewHolder,
                                                int direction) {

    }
});
```

If the required methods were not automatically added, click on the red bulb in the left margin, and select **Implement methods**.

The `SimpleCallback` constructor will be underlined in red because you have not yet provided the required parameters: the direction that you plan to support for moving and swiping list items, respectively.

2. Because we are only implementing swipe to dismiss at the moment, you should pass in `0` for the supported move directions and `ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT` for the supported swipe directions:

```
ItemTouchHelper helper = new ItemTouchHelper(new ItemTouchHelper
                .SimpleCallback(0, ItemTouchHelper.LEFT |
                ItemTouchHelper.RIGHT) {
```

3. You must now implement the desired behavior in `onSwiped()`. In this case, swiping the card left or right should delete it from the list. Call `remove()` on the data set, passing in the appropriate index by getting the position from the `ViewHolder`:

```
mSportsData.remove(viewHolder.getAdapterPosition());
```

4. To allow the `RecyclerView` to animate the deletion properly, you must also call `notifyItemRemoved()`, again passing in the appropriate index by getting the position from the ViewHolder:

```
mAdapter.notifyItemRemoved(viewHolder.getAdapterPosition());
```

5. Below the new `ItemTouchHelper` object in the `onCreate()` method for `MainActivity`, call `attachToRecyclerView()` on the `ItemTouchHelper` instance to add it to your `RecyclerView`:

```
helper.attachToRecyclerView(mRecyclerView);
```

6. Run your app, you can now swipe list items left and right to delete them!

3.2 Implement drag and drop

You can also implement drag and drop functionality using the same `SimpleCallback`. The first argument of the `SimpleCallback` determines which directions the `ItemTouchHelper` supports for moving the objects around. Do the following:

1. Change the first argument of the SimpleCallback from 0 to include every direction, since we want to be able to drag and drop anywhere:

```
ItemTouchHelper helper = new ItemTouchHelper(new ItemTouchHelper
.SimpleCallback(ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT |
ItemTouchHelper.DOWN | ItemTouchHelper.UP, ItemTouchHelper.LEFT |
ItemTouchHelper.RIGHT) {
```

2. In the `onMove()` method, get the original and target index from the second and third argument passed in (corresponding to the original and target view holders).

```
int from = viewHolder.getAdapterPosition();
int to = target.getAdapterPosition();
```

3. Swap the items in the dataset by calling `Collections.swap()` and pass in the dataset, and the initial and final indexes:

```
Collections.swap(mSportsData, from, to);
```

4. Notify the adapter that the item was moved, passing in the old and new indexes, and change the `return` statement to `true`:

```
mAdapter.notifyItemMoved(from, to);
return true;
```

5. Run your app. You can now delete your list items by swiping them left or right, or reorder them using a long press to activate Drag and Drop mode.

3.3 Implement the DetailActivity layout

According to Material Design guidelines, a card is used to provide an entry point to more detailed information. You may find yourself tapping on the cards to see more information about the sports, because that is how you expect cards to behave.

In this section, you will add a detail `Activity` that will be launched when any list item is pressed. For this practical, the detail `Activity` will contain the name and image of the list item you clicked, but will contain only generic placeholder detail text, so you don't have to create custom detail for each list item.

1. Create a new `Activity` by going to **File > New > Activity > Empty Activity**.
2. Call it **DetailActivity**, and accept all of the defaults.
3. Open the newly created **activity_detail.xml** layout file and change the root `ViewGroup` to `RelativeLayout`, as you've done in previous exercises.
4. Remove the `xmlns:app="http://schemas.android.com/apk/res-auto"` statement from the `RelativeLayout`.
5. Copy all of the `TextView` and `ImageView` elements from the **list_item.xml** file to the **activity_detail.xml** file.
6. Add the word "Detail" to the reference in each `android:id` attribute in order to differentiate it from `list_item.xml` IDs. For example, change the `ImageView` ID from **sportsImage** to **sportsImageDetail**.
7. In all `TextView` and `ImageView` elements, change all references to the IDs for relative placement such `layout_below` to use the "Detail" ID.
8. For the `subTitleDetail` `TextView`, remove the placeholder text string and paste a paragraph of generic text to substitute detail text (For example, a few paragraphs of Lorem Ipsum). Extract the text to a string resource.
9. Change the padding on the `TextView` elements to 16dp.
10. Wrap the entire `RelativeLayout` with a `ScrollView`. Add the required `layout_height` and `layout_width` attributes, and append the `xmlns:android="http://schemas.android.com/apk/res/android"` attribute to the end of the `ScrollView`.
11. Change the `layout_height` attribute of the `RelativeLayout` to `"wrap_content"`.

The first two elements of the `activity_detail.xml` layout should now look as follows:

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

    <RelativeLayout xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        tools:context="com.example.android.materialme.DetailActivity">
```

3.4 Implement the detail view and click listener

Follow these steps to implement the detail view and click listener:

1. Open **SportsAdapter** and change the `ViewHolder` inner class, which already extends `RecyclerView.ViewHolder`, to also implement `View.OnClickListener`, and implement the required method (`onClick()`).

```
class ViewHolder extends RecyclerView.ViewHolder
                                implements View.OnClickListener{
    // Rest of ViewHolder code.
    //
    @Override
    public void onClick(View view) {

    }
}
```

2. Set the `OnClickListener` to the `itemView` in the `ViewHolder` constructor. The entire constructor should now look like this:

```
ViewHolder(View itemView) {
    super(itemView);

    //Initialize the views
    mTitleText = itemView.findViewById(R.id.title);
    mInfoText = itemView.findViewById(R.id.subTitle);
    mSportsImage = itemView.findViewById(R.id.sportsImage);

    // Set the OnClickListener to the entire view.
    itemView.setOnClickListener(this);
}
```

3. In the `onClick()` method, get the `Sport` object for the item that was clicked using `getAdapterPosition()`:

```
Sport currentSport = mSportsData.get(getAdapterPosition());
```

4. In the same method, add an `Intent` that launches `DetailActivity`, put the `title` and `image_resource` as extras in the `Intent`, and call `startActivity()` on the `mContext` variable, passing in the new `Intent`.

```
Intent detailIntent = new Intent(mContext, DetailActivity.class);
detailIntent.putExtra("title", currentSport.getTitle());
detailIntent.putExtra("image_resource",
                            currentSport.getImageResource());
mContext.startActivity(detailIntent);
```

5. Open **DetailActivity** and initialize the `ImageView` and `title` `TextView` in `onCreate()`:

```
TextView sportsTitle = findViewById(R.id.titleDetail);
ImageView sportsImage = findViewById(R.id.sportsImageDetail);
```

6. Get the `title` from the incoming `Intent` and set it to the `TextView`:

```
sportsTitle.setText(getIntent().getStringExtra("title"));
```

7. Use Glide to load the image into the `ImageView`:

```
Glide.with(this).load(getIntent().getIntExtra("image_resource",0))
.into(sportsImage);
```

8. Run the app. Tapping on a list item now launches `DetailActivity`.

Task 4: Add the FAB and choose a Material Design color palette

One of the principles behind Material Design is using consistent elements across applications and platforms so that users recognize patterns and know how to use them. You have already used one such element: the Floating Action Button (FAB). The FAB is a circular button that floats above the rest

of the UI. It is used to promote a particular action to the user, one that is very likely to be used in a given activity. In this task, you will create a FAB that resets the dataset to its original state.

4.1 Add the FAB

The Floating Action Button is part of the Design Support Library.

1. Open the **build.gradle (Module: app)** file and add the following line of code for the design support library in the `dependencies` section:

```
implementation 'com.android.support:design:26.1.0'
```

2. Add an icon for the FAB by right-clicking (or Control-clicking) the res folder in the Project > Android pane, and choosing **New > Vector Asset**. The FAB will reset the contents of the `RecyclerView`, so the `refresh` icon should do:  . Change the name to **ic_reset**, click **Next**, and click **Finish**.

3. Open **activity_main.xml** and add a `FloatingActionButton` with the following attributes:

Attribute	**Value**
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_alignParentBottom`	`"true"`
`android:layout_alignParentRight`	`"true`
`android:layout_alignParentEnd`	`"true`
`android:layout_margin`	`"16dp"`

`android:src`	`"@drawable/ic_reset"`
`android:tint`	`"@android:color/white"`
`android:onClick`	`resetSports`

4. Open **MainActivity** and add the `resetSports()` method with a statement to call `initializeData()` to reset the data.
5. Run the app. You can now reset the data by tapping the FAB.

Because the `Activity` is destroyed and recreated when the configuration changes, rotating the device resets the data in this implementation. In order for the changes to be persistent (as in the case of reordering or removing data), you would have to implement `onSaveInstanceState()` or write the changes to a persistent source (like a database or SharedPreferences, which are described in other lessons).

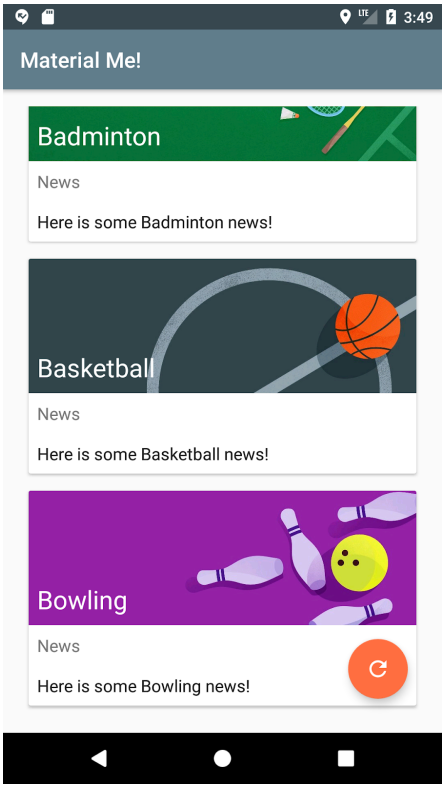
4.2 Choose a Material Design palette

Material Design recommends picking at least these three colors for your app:

- A primary color. This one is automatically used to color your app bar (the bar that contains the title of your app).
- A primary dark color. A darker shade of the same color. This is used for the status bar above the app bar, among other things.
- An accent color. A color that contrasts well with the primary color. This is used for various highlights, but it is also the default color of the FAB.

When you ran your app, you may have noticed that the FAB color and app bar color are already set. In this task you will learn where these colors are set. You can use the Material Color Guide to pick some colors to experiment with.

1. In the **Project > Android** pane, navigate to your **styles.xml** file (located in the **values** directory). The `AppTheme` style defines three colors by default: `colorPrimary`, `colorPrimaryDark`, and `colorAccent`. These styles are defined by values from the `colors.xml` file.
2. Pick a color from the Material Color Guide to use as your primary color, such as `#607D8B` (in the Blue Grey color swatch). It should be within the 300-700 range of the color swatch so that you can still pick a proper accent and dark color.
3. Open the **colors.xml** file, and modify the `colorPrimary` hex value to match the color you picked.
4. Pick a darker shade of the same color to use as your primary dark color, such as `#37474F`. Again, modify the **colors.xml** hex value for `colorPrimaryDark` to match.
5. Pick an accent color for your FAB from the colors whose values start with an A, and whose color contrasts well with the primary color (like Deep Orange A200). Change the `colorAccent` value in **colors.xml** to match.
6. Run the app. The app bar and FAB have now changed to reflect the new color palette!



If you want to change the color of the FAB to something other than theme colors, use the `app:backgroundTint` attribute. Note that this uses the `app:` namespace and Android Studio will prompt you to add a statement to define the namespace.

Solution code

Android Studio project: MaterialMe

Coding challenges

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

Coding challenge 1

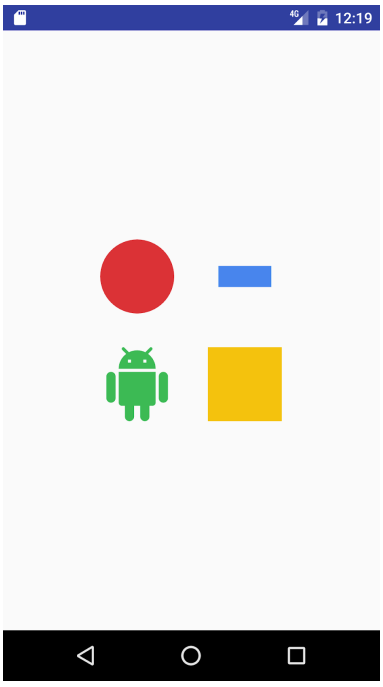
This challenge consists of two small improvements to the MaterialMe app:

- Add real details to the `Sport` object and pass the details to the `DetailActivity`.
- Implement a way to ensure that the state of the app is persistent across orientation changes.

Coding challenge 2

Create an app with four images arranged in a grid in the center of your layout. Make the first three solid colored backgrounds with different shapes (square, circle, line), and the fourth the Android Material Design Icon. Make each of these images respond to clicks as follows:

1. One of the colored blocks relaunches the `Activity` using the `Explode` animation for both the enter and exit transitions.
2. Relaunch the `Activity` from another colored block, this time using the `Fade` transition.
3. Touching the third colored block starts an in-place animation of the `View` (such as a rotation).
4. Finally, touching the Android icon starts a secondary `Activity` with a Shared Element Transition swapping the Android block with one of the other blocks.



Note: You must set your minimum SDK level to 21 or higher in order to implement shared element transitions.

Challenge 2 solution code

Android Studio project: TransitionsandAnimations

Summary

- A `CardView` is a good layout to use for presenting information that has mixed media (such as images and text).
- `CardView` is a UI component found in the Android Support Library.
- `CardView` was *not* designed just for text child `View` elements.
- Loading images directly into an `ImageView` is memory intensive, because images are loaded at full resolution. To efficiently load images into your app, use an image loading library such as Glide.
- The Android SDK has a class called `ItemTouchHelper` that helps your app get information about tap, swipe, and drag-and-drop events in your UI.
- A `FloatingActionButton` (FAB) focuses the user on a particular action and "floats" in your UI.
- Material Design is a set of guiding principles for creating consistent, intuitive, and playful applications.
- According to Material Design, it's good practice to choose three colors for your app: a primary color, a primary dark color, and an accent color.
- Material Design promotes the use of bold imagery and colors to enhance user experience. It also promotes consistent elements across platforms, for example by using `CardView` and FAB widgets.
- Use Material Design to create meaningful, intuitive motion for UI elements such as cards that can be dismissed or rearranged.

Related concept

The related concept documentation is in 5.2: Material Design.

Learn more

Android Studio documentation:

- Android Studio User Guide
- Add multi-density vector graphics
- Create app icons with Image Asset Studio

Android developer documentation:

- User Interface & Navigation
- Linear Layout
- `FloatingActionButton`
- Drawable Resources
- View Animation
- Support Library
- Animate Drawable Graphics
- Loading Large Bitmaps Efficiently

Material Design:

- Material Design for Android
- Material Design palette generator
- Create a List with RecyclerView
- App Bar
- Defining Custom Animations

Android Developers Blog: Android Design Support Library

Other:

- Glide documentation
- Glide github page

Homework

Build and run an app

Open the MaterialMe app.

1. Create a shared element transition between the `MainActivity` and the `DetailActivity`, with the banner image for the sport as the shared element.
2. Clicking on a list item in the MaterialMe app triggers the transition. The banner image from the card moves to the top of the screen in the detail view.

Answer these questions

Question 1

Which color attribute in your style defines the color of the status bar above the app bar? Choose one:

- `colorPrimary`
- `colorPrimaryDark`
- `colorAccent`
- `colorAccentDark`

Question 2

Which support library does the `FloatingActionButton` belong to? Choose one:

- `v4 Support Library`
- `v7 Support Library`
- `Design Support Library`
- `Custom Button Support Library`

Question 3

In the Material Design color palette, which shade of a color should you use as the primary color for your *brand* in your app? Choose one:

- Any color shade that starts with `A`.
- Any color shade labeled `200`.
- Any color shade labeled `500`.
- Any color shade labeled `900`.

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- Window-content transitions are enabled in the app theme.
- A shared element transition is specified in the app style.
- The transition is defined as an XML resource.
- A common name is assigned to the shared elements in both layouts with the `android:transitionName` attribute.
- The code uses the `ActivityOptions.makeSceneTransitionAnimation()` method.

Lesson 5.3: Adaptive layouts

Introduction

The MaterialMe app that you created in a previous chapter doesn't properly handle device-orientation changes from portrait (vertical) mode to landscape (horizontal) mode. On a tablet, the font sizes are too small, and the space is not used efficiently.

The Android framework has a way to solve both issues. *Resource qualifiers* allow the Android runtime to use alternate XML resource files depending on the device configuration—the orientation, the locale, and other qualifiers. For a full list of available qualifiers, see Providing alternative resources.

In this practical you optimize the use of space in the MaterialMe app so that the app works well in landscape mode and on tablets. In another practical on using the layout editor, you learned how to create layout variants for horizontal orientation and tablets. In this practical you use an *adaptive* layout, which is a layout that works well for different screen sizes and orientations, different devices, different locales and languages, and different versions of Android.

What you should already know

You should be able to:

- Create and run apps in Android Studio.
- Create and edit UI elements using the layout editor.
- Edit XML layout code, and access elements from your Java code.
- Create a click handler for a `Button` click.
- Use drawables, styles, and themes.
- Extract text to a string resource and a dimension to a dimension resource.

What you'll learn

You will learn how to:

- Create alternate resources for devices in landscape mode.
- Create alternate resources for tablets.

- Create alternate resources for different locales.

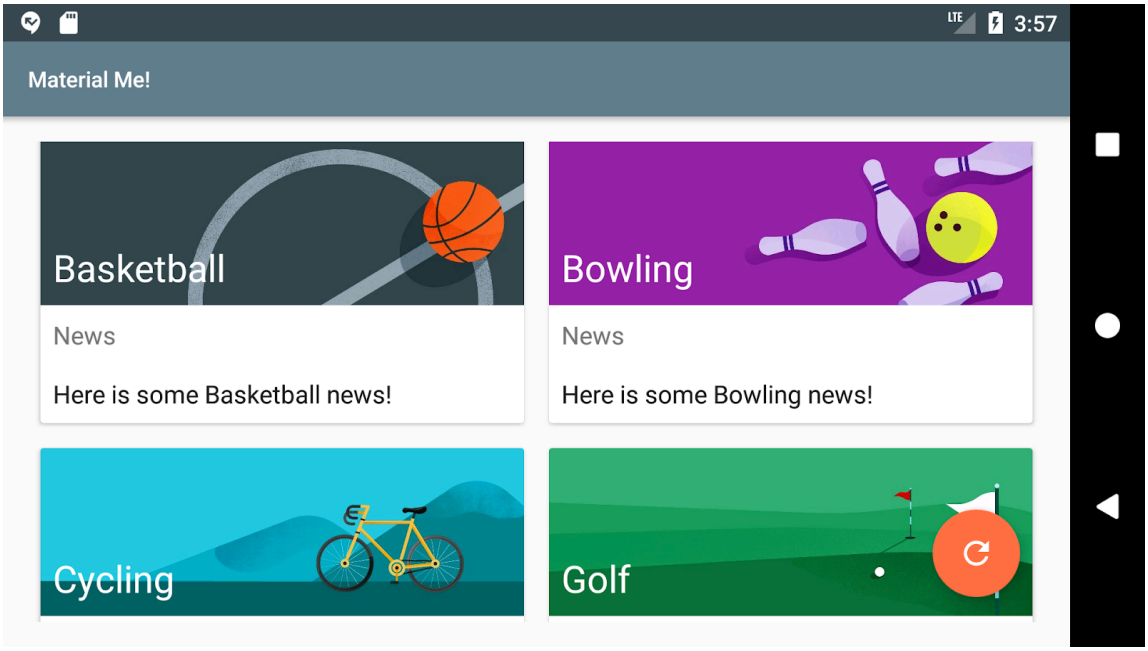
What you'll do

- Update the MaterialMe app for better use of space in landscape mode.
- Add an alternative layout for tablets.
- Localize the content of your app.

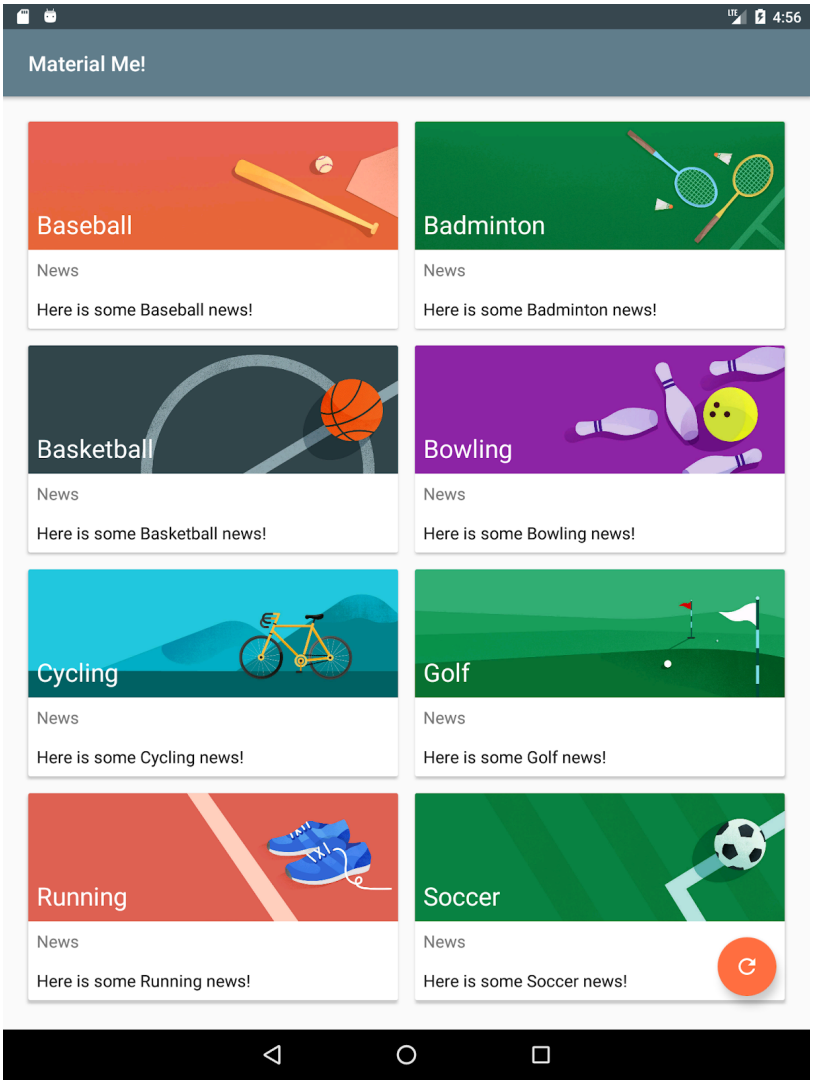
App overview

The updated MaterialMe app will include an improved layout for landscape mode on phones. It will also include improved layouts for portrait and landscape modes on tablets, and it will offer localized content for users outside the United States.

The screenshot below shows a phone running the updated MaterialMe app in landscape orientation:



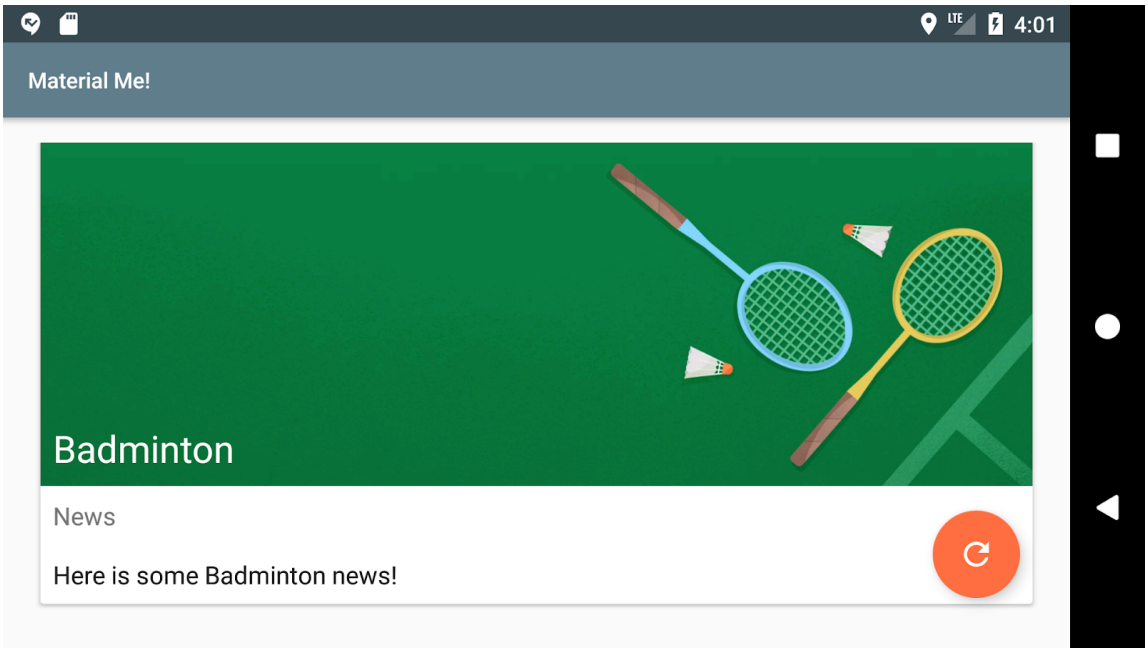
The screenshot below shows a tablet running the updated MaterialMe app, with qualifiers to show two columns when running in portrait orientation:



Task 1: Support landscape orientation

You may recall that when the user changes the orientation of the device, the Android framework destroys and recreates the current activity. The new orientation often has different layout requirements than the original one. For example, the MaterialMe app looks good in portrait mode,

but does not make optimal use of the screen in landscape mode. With the larger width in landscape mode, the image in each list item overwhelms the text providing a poor user experience.



In this task, you will create an alternative resource file that will change the appearance of the app when it is used in landscape orientation.

1.1 Change to a GridLayoutManager

Layouts that contain list items often look unbalanced in landscape mode when the list items include full-width images. One good solution is to use a grid instead of a linear list when displaying `CardView` elements in landscape mode.

Recall that the items in a `RecyclerView` list are placed using a `LayoutManager`; until now, you have been using the `LinearLayoutManager` which lays out each item in a vertical or horizontal scrolling list. `GridLayoutManager` is another layout manager that displays items in a grid, rather than a list.

When you create a `GridLayoutManager`, you supply two parameters: the app `context`, and an integer representing the number of columns. You can change the number of columns programmatically, which gives you flexibility in designing adaptive layouts. In this case, the number of columns integer should be 1 in portrait orientation (single column) and 2 when in landscape mode. Notice that when the number of columns is 1, a `GridLayoutManager` behaves similar to a `LinearLayoutManager`.

This practical builds on the MaterialMe app from the previous practical.

1. Continue developing your version of the MaterialMe app, or download MaterialMe. If you decide to make a copy of the MaterialMe project to preserve the version from the previous practical, rename the copied version **MaterialMe-Resource**.
2. Create a new resources file called `integers.xml`. To do this, open the **res** folder in the **Project > Android** pane, **right-click** (or **Control-click**) on the **values** folder, and select **New > Values resource file**.
1. Name the file **integers.xml** and click **OK**.
2. Create an integer constant between the `<resources>` tags called `grid_column_count` and set it equal to 1:

```
<integer name="grid_column_count">1</integer>
```

3. Create another values resource file, again called **integers.xml**; however, the name will be modified as you add resource qualifiers from the **Available qualifiers** pane. The resource qualifiers are used to label resource configurations for various situations.
4. Select **Orientation** in the **Available qualifiers** pane, and press the **>>** symbol in the middle of the dialog to assign this qualifier.
5. Change the **Screen orientation** menu to **Landscape**, and notice how the directory name `values-land` appears. This is the essence of resource qualifiers: the directory name tells Android when to use that specific layout file. In this case, that is when the phone is rotated to landscape mode.
6. Click **OK** to generate the new layout file.
7. Copy the integer constant you created into this new resource file, but change the value to 2.

You should now have two individual `integers.xml` files grouped into an `integers.xml` folder in the Project > Android pane. The second file is labeled with the qualifier you selected, which is `land` in this case. The qualifier appears in parentheses: `integers.xml (land)`.

1.2 Modify MainActivity

1. Open **MainActivity**, and add code to `onCreate()` to get the integer from the `integers.xml` resource file:

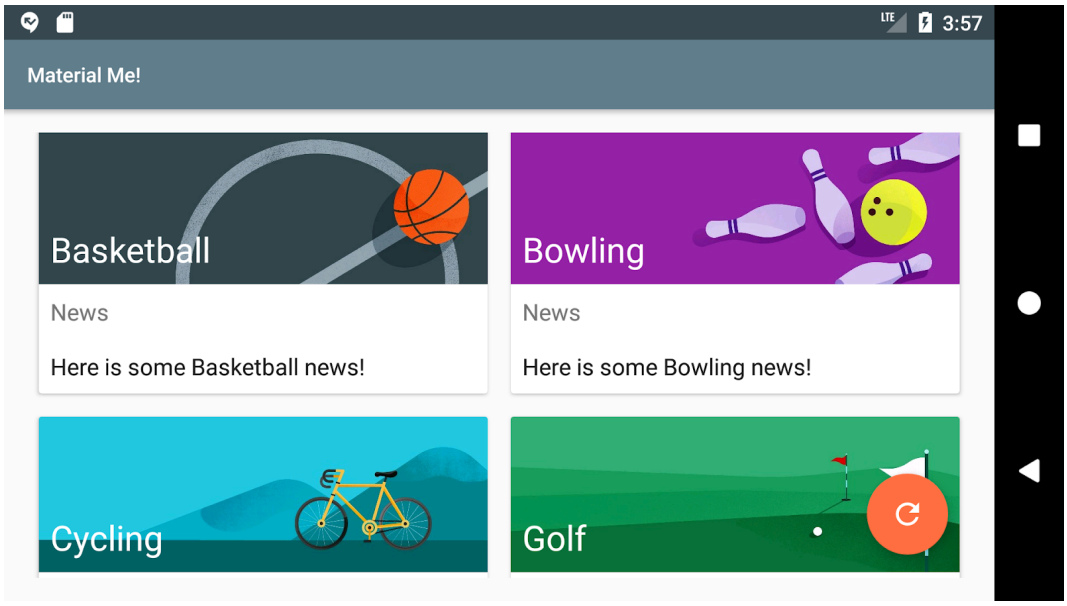
```
int gridColumnCount =
        getResources().getInteger(R.integer.grid_column_count);
```

The Android runtime will take care of deciding which `integers.xml` file to use, depending on the state of the device.

2. Change the `LinearLayoutManager` for the `RecyclerView` to a `GridLayoutManager`, passing in the context and the newly created integer:

```
mRecyclerView.setLayoutManager(new
                    GridLayoutManager(this, gridColumnCount));
```

3. Run the app and rotate the device. The number of columns changes automatically with the orientation of the device.



When using the app in landscape mode, you will notice that the swipe to dismiss functionality is no longer intuitive, since the items are now in a grid rather than a single column. In the next steps, you will turn off the swipe action if there is more than one column.

4. Use the `gridColumnCount` variable to disable the swipe action (set `swipeDirs` to zero) when there is more than one column:

```
int swipeDirs;
if(gridColumnCount > 1){
   swipeDirs = 0;
} else {
   swipeDirs = ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT;
}
```

5. Use `swipeDirs` in place of the swipe direction arguments (`ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT`) for `ItemTouchHelper.SimpleCallback()`:

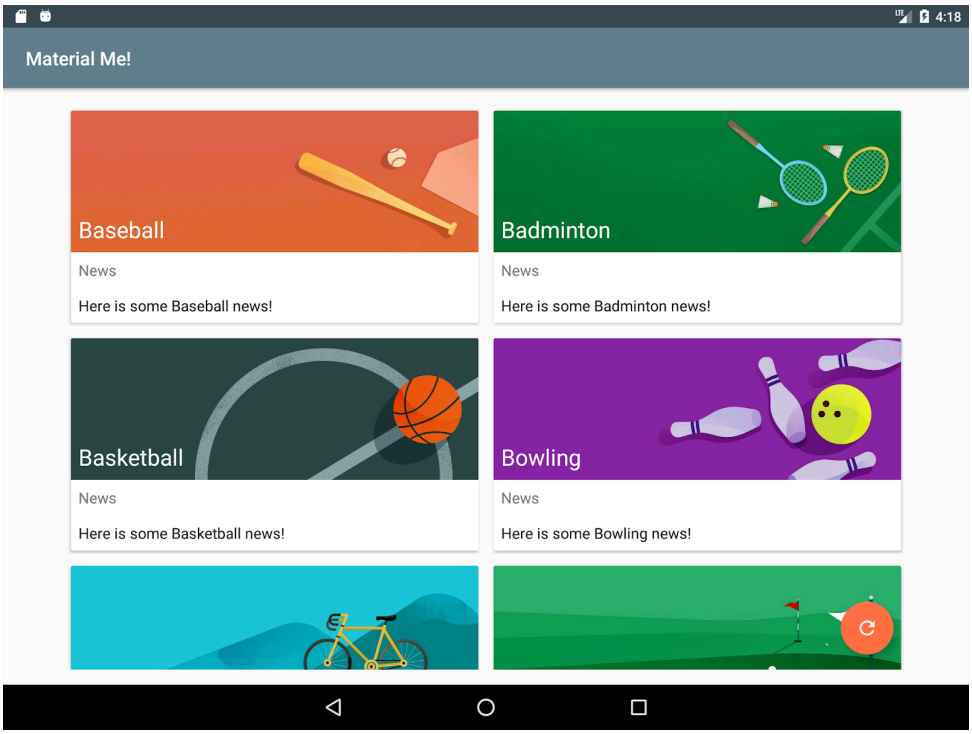
```
ItemTouchHelper helper = new ItemTouchHelper(new
          ItemTouchHelper.SimpleCallback(ItemTouchHelper.LEFT |
                ItemTouchHelper.RIGHT |
                ItemTouchHelper.DOWN | ItemTouchHelper.UP,
                swipeDirs) {
```

6. Run the app and rotate the device. In landscape (horizontal) orientation, the user can no longer swipe to delete a card.

Task 2 : Support tablets

Although you have modified the app to look better in landscape mode, running it on a tablet with physically larger dimensions results in all the text appearing too small. Also when the device is in landscape orientation, the screen is not used efficiently; three columns would be more appropriate for a tablet-sized screen in landscape mode.

In this task, you will add additional resource qualifiers to change the appearance of the app when used on tablets.



2.1 Adapt the layout to tablets

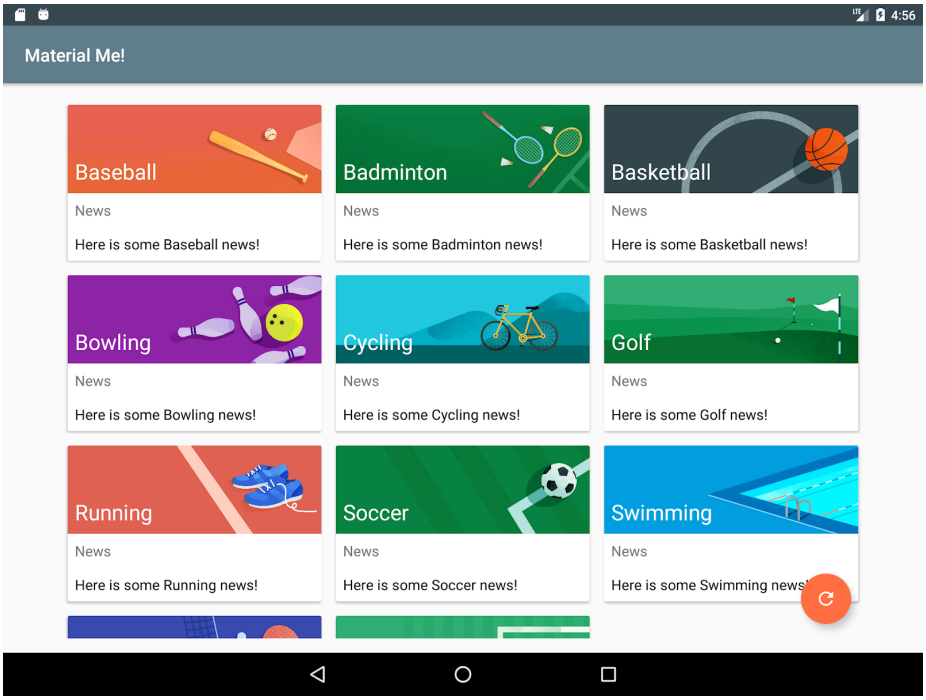
In this step, you create different resource qualifiers to maximize screen use for tablet-sized devices, increasing the column count to 2 for portrait (vertical) orientation and 3 for landscape (horizontal) orientation.

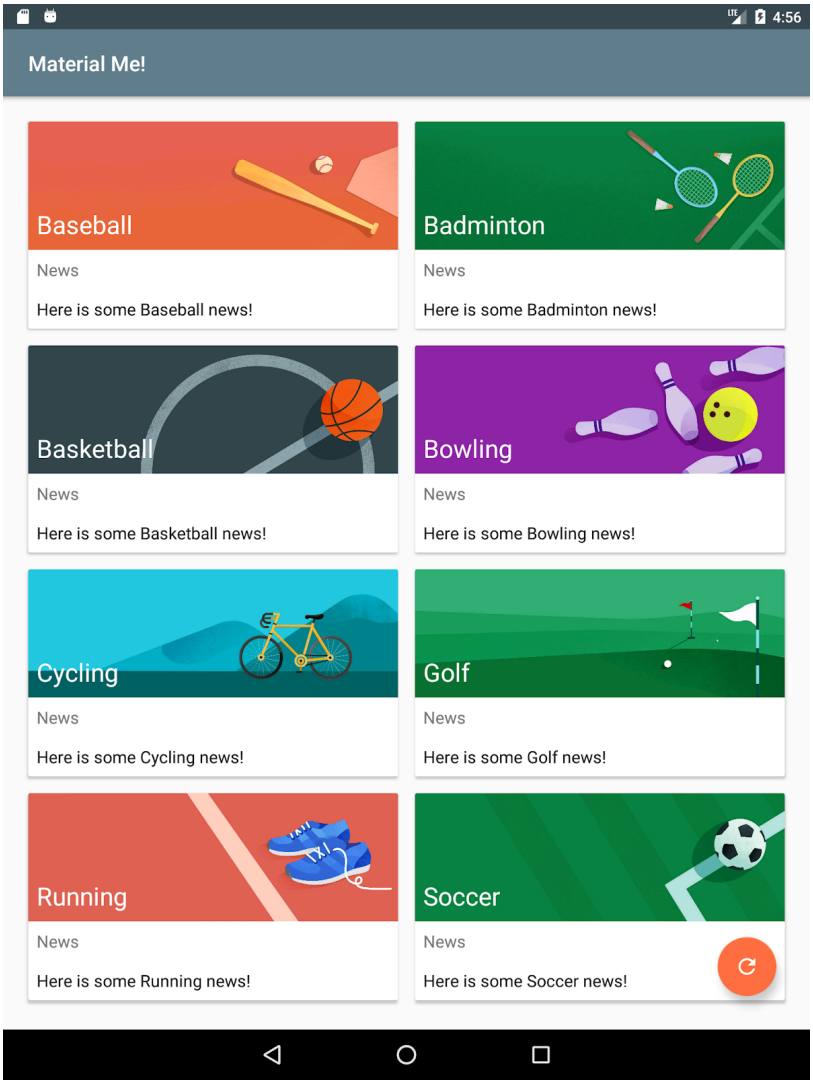
The resource qualifier you need depends on your specific requirements. When creating a new resource file, there are several qualifiers in the **Available qualifiers** pane that you can use to select the correct conditions:

- **Smallest Screen Width**: This qualifier is used most frequently to select for tablets. It is defined by the *smallest* width of the device (regardless of orientation), which removes the ambiguity when talking about "height" and "width" since some devices are traditionally held in landscape mode, and others in portrait. Anything with a smallest width of at least 600dp is considered a tablet.
- **Screen Width**: The screen width is the *effective* width of the device, regardless of the orientation. The width changes when the device is rotated, since the effective height and width of the device are switched.
- **Screen Height**: Same as **Screen Width**, except it uses the effective height instead of the effective width.

To start this task:

1. Create an **integers.xml** resource file which uses the **Smallest Screen Width** qualifier with the value set to **600**. Android uses this file whenever the app runs on a tablet.
2. Copy the code from the **integers.xml (land)** file (it has a grid count of 2) and paste it in the new **integers.xml (sw600dp)** file.
3. Create another **integers.xml** file that includes both the **Smallest Screen Width** qualifier set to **600**, *and* the **Orientation** qualifier set to **Landscape**. Android uses the resulting `integers.xml (sw600dp-land)` file when the app runs on a tablet in landscape mode.
4. Copy the code from the **integers.xml (land)** file and paste it in the new **integers.xml (sw600dp-land)** file.
5. Change the `grid_column_count` variable to 3 in the **integers.xml (sw600dp-land)** file.
6. Run the app on a tablet or tablet emulator, and rotate it to landscape mode. The app should show three columns of cards, as shown in the first figure below. Rotate it to portrait mode, and the app should show two columns of cards, as shown in the second figure below. With these resource qualifier files, the app uses the screen real estate much more effectively.





Tip: If your app uses multiple resource files, Android will use the resource file with the most specific resource qualifier first. For example, if a value is defined in the `integers.xml` file for both **Smallest Screen Width** qualifier and with the **Orientation** set to **Landscape**, Android will use the value for Smallest Screen Width. The precedence for resource qualifiers and resource files is described by Table 2 in the App resources overview.

2.2 Update the tablet list item styles

At this point, your app changes the number of columns in a `GridLayoutManager` to fit the orientation of the device and maximize the use of the screen. However, the `TextView` elements that appeared correctly-sized on a phone's screen now appear too small for the larger screen of a tablet.

To fix this, you will extract the `TextAppearance` styles from the layout resource files into the `styles.xml` resource file. You will also create additional `styles.xml` files for tablets using resource qualifiers.

Note: You could also create alternative layout files with the proper resource qualifiers, and change the styles of the `TextView` elements in those. However, this would require more code duplication, because most of the layout information is the same no matter what device you use, so you will only extract the attributes that will change.

Follow these steps to add the `TextAppearance` styles:

1. Open **styles.xml** and add the following styles:

```
<style name="SportsDetailText"
                   parent="TextAppearance.AppCompat.Subhead"/>
<style name="SportsTitle"
                   parent="TextAppearance.AppCompat.Headline"/>
```

2. Create a new `values` resource file called **styles.xml** that uses the **Smallest Screen Width** qualifier with a value of **600** for tablets.
3. Copy *all* styles from the original **styles.xml** file into the new **styles.xml (sw600dp)** file.
4. In **styles.xml (sw600dp)**, change the `parent` of the `SportsTitle` style to **"TextAppearance.AppCompat.Display1"**:

```
<style name="SportsTitle"
```

```
                    parent="TextAppearance.AppCompat.Display1"/>
```

5. The Android predefined `Display1` style uses the `textColorSecondary` value from the current theme (`ThemeOverlay.AppCompat.Dark`), which in this case is a light gray color. The light gray color does not show up well on the banner images in your app. To correct this add an "`android:textColor`" attribute to the `SportsTitle` style and set it to **"?android:textColorPrimary"**:

```
<style name="SportsTitle"
                  parent="TextAppearance.AppCompat.Display1">
    <item name=
          "android:textColor">?android:textColorPrimary</item>
</style>
```

The question mark tells Android runtime to find the value in the theme applied to the `View`. In this example the theme is `ThemeOverlay.AppCompat.Dark` in which the `textColorPrimary` attribute is white.

6. Change the `parent` of `SportsDetailText` style to **"TextAppearance.AppCompat.Headline"**.
7. To update the style of the `TextView` elements, open **list_item.xml**, and change the `style` attribute of the `title` `TextView` to **@style/SportsTitle**:

```
style="@style/SportsTitle"
```

8. Change the `style` attribute of the `newsTitle` and `subTitle` `TextView` elements to **`@style/SportsDetailText`**.
9. Run your app on a tablet or tablet emulator. Each list item now has a larger text size on the tablet.

2.3 Update the tablet sports detail styles

You have now fixed the display for the `MainActivity`, which lists all the `Sports CardView` elements. The `DetailActivity` still has the same font sizes on tablets and phones.

1. Add the following `style` in the **styles.xml** file for the detail title:

```
<style name="SportsDetailTitle"
                   parent="TextAppearance.AppCompat.Headline"/>
```

2. Add the following `style` in the **styles.xml (sw600dp)** file for the detail title:

```
<style name="SportsDetailTitle"
                   parent="TextAppearance.AppCompat.Display3"/>
```

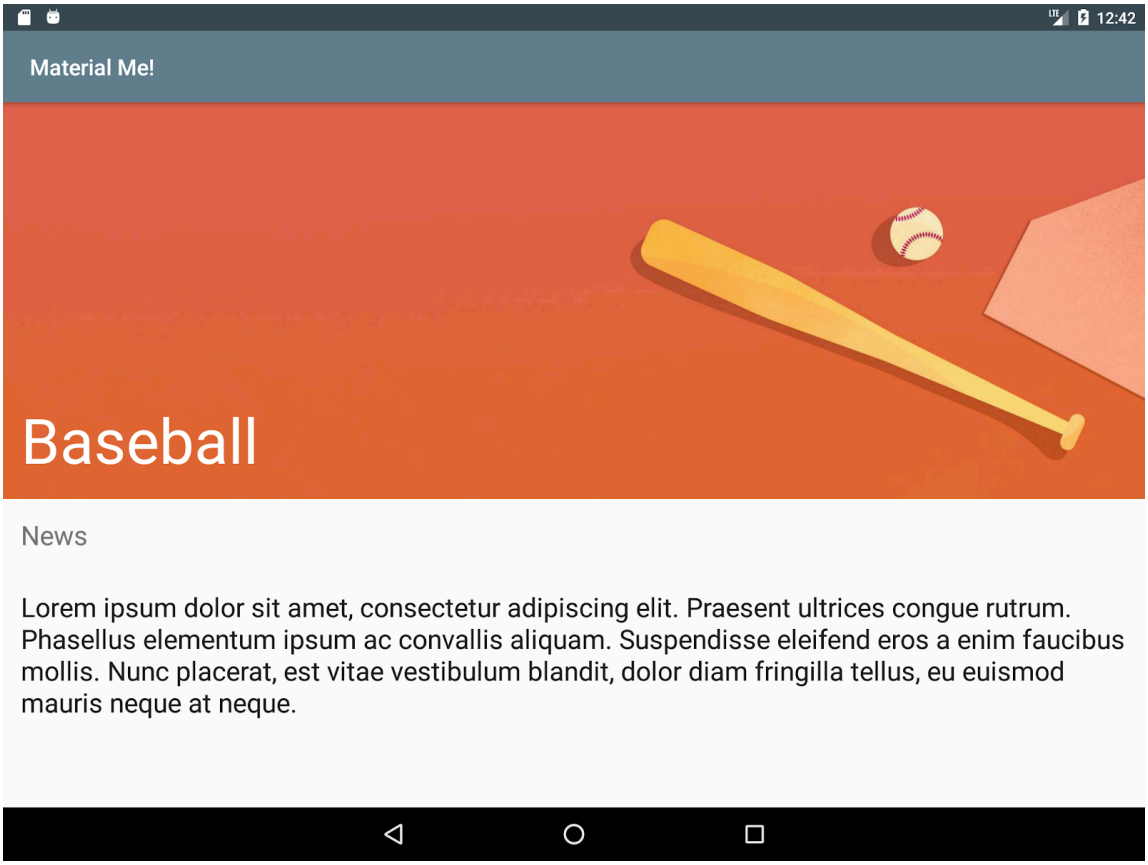
3. Open **activity_detail.xml**, and change the `style` attribute of both the `newsTitleDetail` and `subTitleDetail` TextView elements to the new `SportsDetailText` style you created in a previous step:

```
style="@style/SportsDetailText"
```

4. In **activity_detail.xml**, change the `style` attribute of the `titleDetail TextView` element to the new `SportsDetailTitle` style you created:

```
style="@style/SportsDetailTitle"
```

5. Run your app. All of the text is now larger on the tablet, which greatly improves the user experience of your application.



Task 3: Localize your app

A "locale" represents a specific geographic, political or cultural region of the world. Resource qualifiers can be used to provide alternate resources based on the users' locale. Just as for

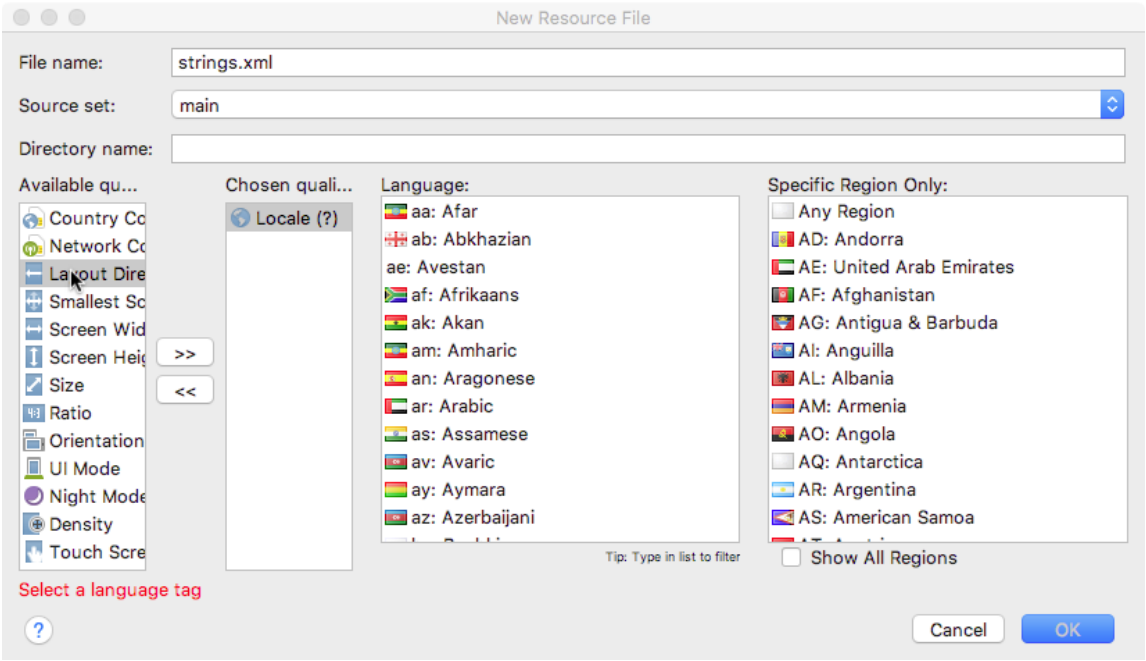
orientation and screen width, Android provides the ability to include separate resource files for different locales. In this step, you will modify your `strings.xml` file to be a little more international.

3.1 Add a localized strings.xml file

You may have noticed that the sports information contained in this app is designed for users from the U.S. The app uses the term "soccer" to represent a sport known as "football" everywhere else in the world.

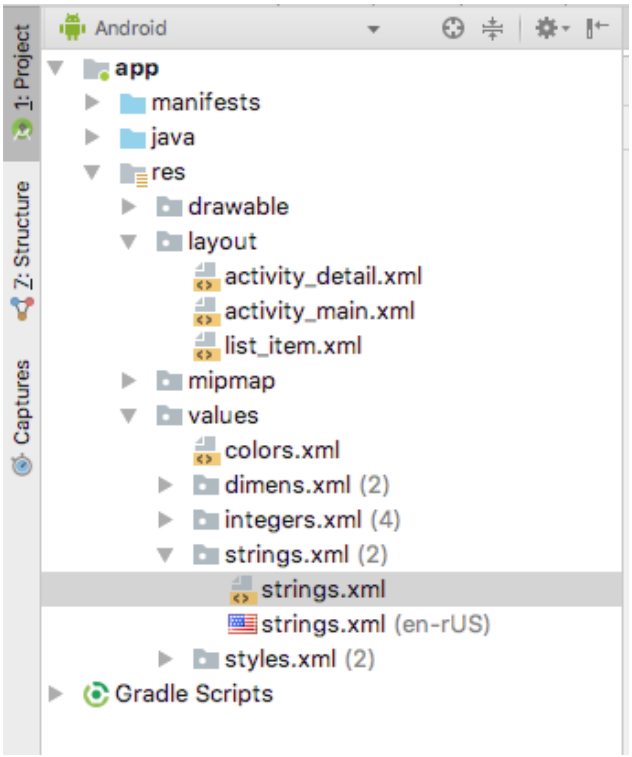
To make your app more internationalized, you can provide a locale-specific `strings.xml` file. This alternative-resource file will show the word "soccer" to users in the U.S. The generic `strings.xml` file will show the word "football" to users in all other locales.

1. Create a new `values` resource file.
2. Call the file **strings.xml** and select **Locale** from the list of available qualifiers. The Language and Specific Region Only panes appear.



3. In the **Language** pane, select **en: English**.

4. In the **Specific Region Only** pane, select **US: United States** and click **OK**. Android Studio creates a specific `values` directory in your project directories for the U.S. locale, called `values-en-rUS`. In the **Project > Android** pane, the `strings.xml` file in this directory appears as **strings.xml (en-rUS)** within the newly created **strings.xml** folder (with a U.S. flag icon).



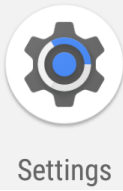
5. Copy all string resources of the generic **strings.xml** file (now located in the **strings.xml** folder) to **strings.xml (en-rUS)**.
6. In the generic **strings.xml** file, change the **Soccer** item in the `sports_titles array` to **Football**, and change the **Soccer news** text in the `sports_info` array to **Football news**.

3.2 Run the app in different locales

In order to see the locale-specific differences, you can start your device or emulator, and change its language and locale to U.S. English (if not already set). In U.S. English, you should see "Soccer". You

can then switch to any language and locale *other than* U.S. English, and run the app again. You should then see "Football".

1. 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:



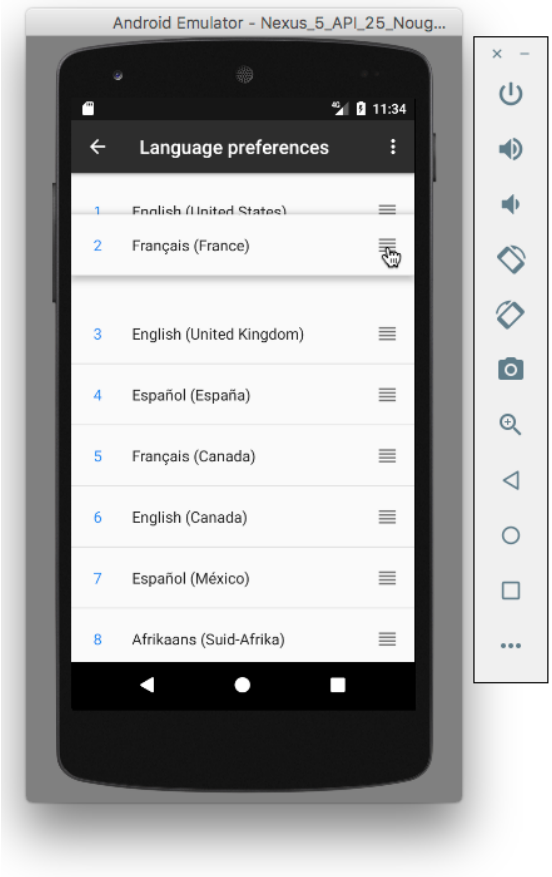
2. Find the **Languages & input** settings in the Settings app, and choose **Languages**.

   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.

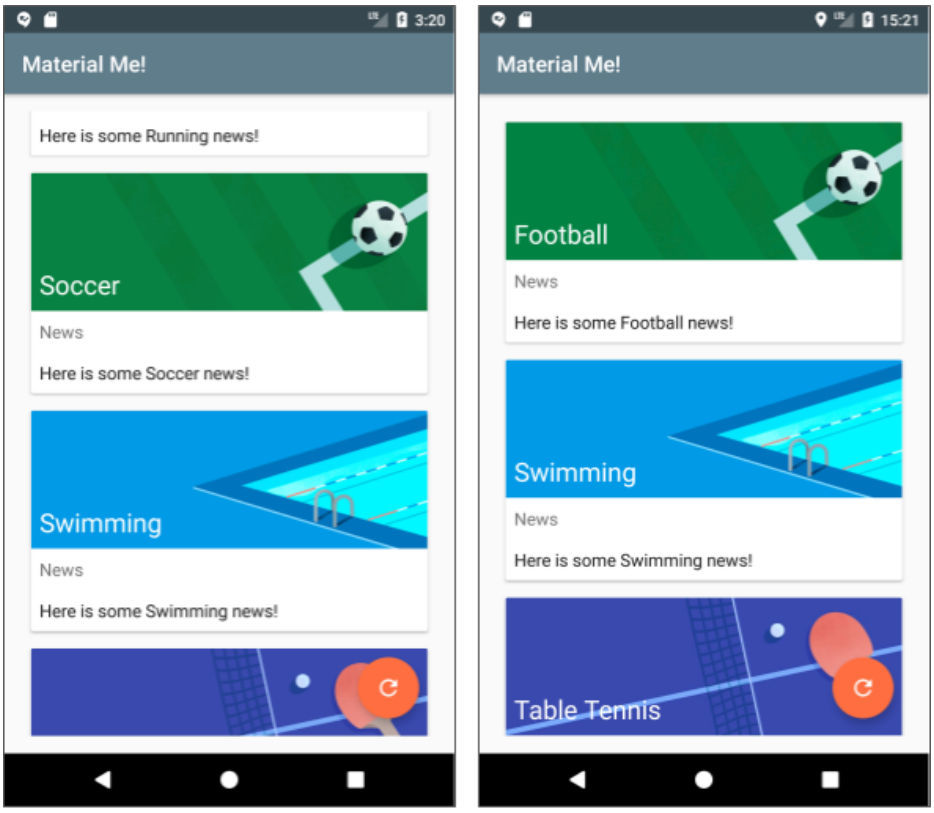
3. For devices and emulators running a version of Android previous to Android 7, choose **Language** on the **Languages & input** screen, select a language and locale such as **Français (France)**, and skip the following steps.

   (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.)

4. For devices and emulators running Android 7 or newer, choose **Languages** on the **Languages & input** screen, select a language such as **Français (France)**, and use the move icon on the right side of the **Language preferences** screen to drag **Français (France)** to the top of the list.



5. Run the app with your device or emulator. In U.S. English, you should see "Soccer".
6. Switch to any language and locale *other than* U.S. English, and run the app again. You should then see "Football".



This example does not show a translated word for "Football" depending on the language. For a lesson in localizing an app with translations, see the Advanced Android Development — Practicals.

Solution code

Android Studio project: MaterialMe-Resource

Coding challenge

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

Challenge 1: It turns out that several countries other than the U.S. use "soccer" instead of "football". Research these countries and add localized strings resources for them.

Challenge 2: Use the localization techniques you learned in Task 3 in combination with Google translate to translate all of the strings in your app into a different language.

Summary

- `GridLayoutManager` is a layout manager that handles two-dimensional scrolling lists.
- You can dynamically change the number of columns in a `GridLayoutManager`.
- The Android runtime uses alternative configuration files, depending on the runtime environment of the device running your app. For example, the runtime might use alternative configuration files for different device layouts, screen dimensions, locale, countries, or keyboard types.
- In your code, you create these alternative resources for the Android runtime to use. The resources are located in files that have resource qualifiers as part of their names.
- The format for a directory holding alternative resource files is *`<resource_name>-<qualifier>`*.
- You can qualify any file in your `res` directory in this way.

Related concepts

The related concept is in 5.3: Resources for adaptive layouts.

Learn more

Android Studio documentation: Android Studio User Guide

Android developer documentation:

- Resources Overview
- Providing Resources
- Localizing with Resources
- `LinearLayoutManager`
- `GridLayoutManager`
- Supporting Multiple Screens

Material Design:

- Material Design for Android
- Material Design Guidelines

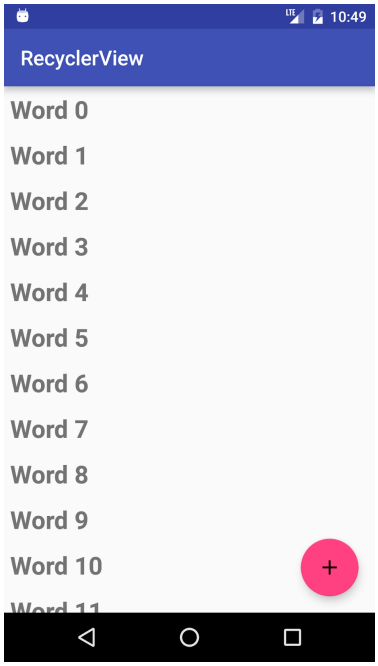
Homework

Build and run an app

Modify the RecyclerView app to use a `GridLayoutManager` with the following column counts:

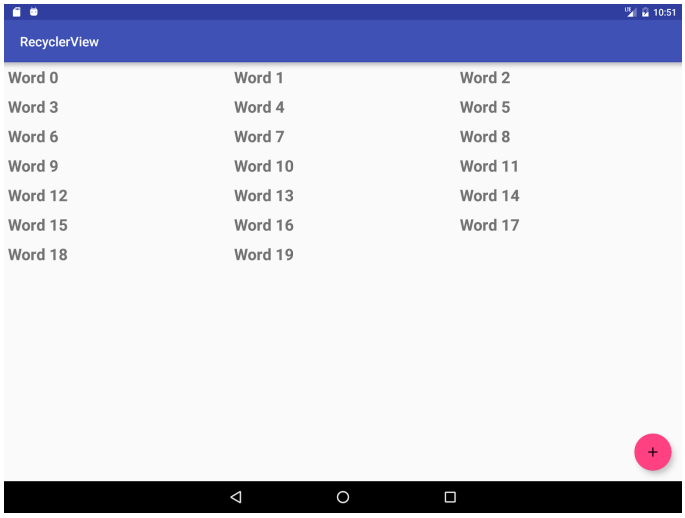
- For a phone: 1 column in portrait orientation, 2 columns in landscape orientation
- For a tablet: 2 columns in portrait orientation, 3 columns in landscape orientation

The screenshot below shows a resource-qualified `RecyclerView` on a phone in portrait orientation:



The screenshot below shows a resource-qualified `RecyclerView` on a phone in landscape orientation:

The screenshot below shows a resource-qualified `RecyclerView` on a tablet in landscape orientation:



Answer these questions

Question 1

Which resource qualifier is used most frequently to select for tablets? Choose one:

- Orientation
- Screen width
- Screen height
- Smallest screen width

Question 2

Which folder would hold the `strings.xml` file for translation into French for Canada? Choose one:

- `res/values-fr-rFR/`
- `res/values-ca-rFR/`
- `res/values-fr-rCA/`
- `res/values-en-rFR/`

Question 3

Which folder is for XML files that contain strings, integers, and colors? Choose one:

- `res/layout`
- `res/mipmap`
- `res/raw`
- `res/values`

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- For phones and tablets in both landscape and portrait modes, the code includes resource-qualified `values` files that contain the integer for the column count.
- The app uses `getResources().getInteger()` to retrieve a value from a resource file, then uses the value as the column count for grid layout.

Lesson 6.1: Espresso for UI testing

Introduction

As a developer, it's important that you test user interactions within your app to make sure that your users don't encounter unexpected results or have a poor experience with your app.

You can test an app's user interface (UI) manually by running the app and trying the UI. But for a complex app, you couldn't cover all the permutations of user interactions within all the app's

functionality. You would also have to repeat these manual tests for different device configurations in an emulator, and on different hardware devices.

When you automate tests of UI interactions, you save time, and your testing is systematic. You can use suites of automated tests to perform all the UI interactions automatically, which makes it easier to run tests for different device configurations. To verify that your app's UI functions correctly, it's a good idea to get into the habit of creating UI tests as you code.

Espresso is a testing framework for Android that makes it easy to write reliable UI tests for an app. The framework, which is part of the Android Support Repository, provides APIs for writing UI tests to simulate user interactions within the app—everything from clicking buttons and navigating views to selecting menu items and entering data.

What you should already know

You should be able to:

- Create and run apps in Android Studio.
- Check for the Android Support Repository and install it if necessary.
- Create and edit UI elements using the layout editor and XML.
- Access UI elements from your code.
- Add a click handler to a `Button`.

What you'll learn

You will learn how to:

- Set up Espresso in your app project.
- Write Espresso tests.
- Test for user input and check for the correct output.
- Find a spinner, click one of its items, and check for the correct output.
- Use the **Record Espresso Test** function in Android Studio.

What you'll do

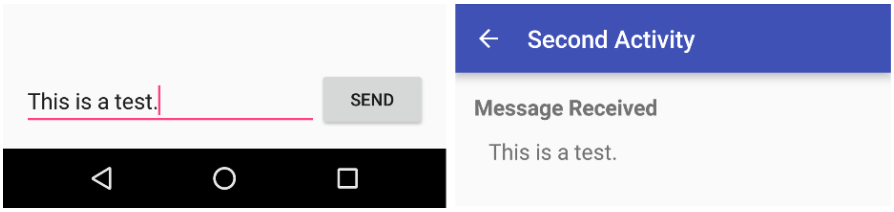
- Modify a project to create Espresso tests.
- Test the app's text input and output.
- Test clicking a spinner item and check its output.
- Record an Espresso test of a `RecyclerView`.

App overview

Tip: For an introduction to testing Android apps, see Test your app.

In this practical you modify the TwoActivities project from a previous lesson. You set up Espresso in the project for testing, and then you test the app's functionality.

The TwoActivities app lets a user enter text in a text field and tap the **Send** button, as shown on the left side of the figure below. In the second `Activity`, the user views the text they entered, as shown on the right side of the figure below.



Task 1: Set up Espresso in your project

To use Espresso, you must already have the Android Support Repository installed with Android Studio. You may also need to configure Espresso in your project.

In this task you check to see if the repository is installed. If it is not, you will install it.

1.1 Check for the Android Support Repository

1. Download the TwoActivities project from the previous lesson on creating and using an `Activity`.
2. Open the project in Android Studio, and choose **Tools > Android > SDK Manager**.

   The Android SDK **Default Preferences** pane appears.

3. Click the **SDK Tools** tab and expand **Support Repository**.
4. Look for **Android Support Repository** in the list.

   If **Installed** appears in the Status column, you're all set. Click **Cancel**.

   If **Not installed** or **Update Available** appears, click the checkbox next to **Android Support Repository**. A download icon should appear next to the checkbox. Click **OK.**

5. Click **OK** again, and then **Finish** when the support repository has been installed.

1.2 Configure Espresso for the project

When you start a project for the phone and tablet form factor using **API 15: Android 4.0.3 (Ice Cream Sandwich)** as the minimum SDK, Android Studio automatically includes the dependencies you need to use Espresso. To execute tests, Espresso and UI Automator use JUnit as their testing framework. JUnit is the most popular and widely-used unit testing framework for Java. Your test class using Espresso or UI Automator should be written as a JUnit 4 test class.

Note: The most current JUnit revision is JUnit 5. However, for the purposes of using Espresso or UI Automator, version 4.12 is recommended.

If you have created your project in a previous version of Android Studio, you may have to add the dependencies and instrumentation yourself. To add the dependencies yourself, follow these steps:

1. Open the TwoActivities project, or if you prefer, make a copy of the project first and then open the copy.

2. Open the **build.gradle (Module: app)** file.
3. Check if the following is included (along with other dependencies) in the `dependencies` section:

```
testImplementation 'junit:junit:4.12'
androidTestImplementation 'com.android.support.test:runner:1.0.1'
androidTestImplementation
           'com.android.support.test.espresso:espresso-core:3.0.1'
```

If the file doesn't include the above dependency statements, enter them into the `dependencies` section.

4. Android Studio also adds the following instrumentation statement to the end of the `defaultConfig` section of a new project:

```
testInstrumentationRunner
                "android.support.test.runner.AndroidJUnitRunner"
```

If the file doesn't include the above instrumentation statement, enter it at the end of the `defaultConfig` section.

Instrumentation is a set of control methods, or hooks, in the Android system. These hooks control an Android component independently of the component's normal lifecycle. They also control how Android loads apps. Using instrumentation makes it possible for tests to invoke methods in the app, and modify and examine fields in the app, independently of the app's normal lifecycle.

5. If you modified the **build.gradle (Module: app)** file, click the **Sync Now** link in the notification about Gradle files in top right corner of the window.

1.3 Turn off animations on your test device

To let Android Studio communicate with your device, you must first turn on USB Debugging on your device, as described in the lesson on installing and running apps.

Android phones and tablets display animations when moving between apps and screens. The animations are attractive when using the device, but they slow down performance, and may also cause unexpected results or may lead your test to fail. So it's a good idea to turn off animations on your physical device. To turn off animations on your test device, tap on the Settings icon on your physical device. Look for **Developer Options.** Now look for the **Drawing** section. In this section, turn off the following options:

- Window animation scale
- Transition animation scale
- Animator duration scale

Tip: Instrumenting a system, for example executing unit tests, can alter the timing of some functions. For this reason, keep unit testing and debugging separate:

> Unit testing uses an API based Espresso framework with hooks for instrumentation. Debugging uses breakpoints and other methods in the coding statements within your app's code, as described in the lesson on debugging.

Task 2: Test for Activity switching and text input

You write Espresso tests based on what a user might do while interacting with your app. The Espresso tests are classes that are separate from your app's code. You can create as many tests as you need in order to interact with the `View` elements in your UI that you want to test.

The Espresso test is like a robot that must be told what to do. It must *find* the `View` you want it to find on the screen, and it must *interact* with it, such as clicking the `View`, and checking the contents of the `View`. If it fails to do any of these things properly, or if the result is not what you expected, the test fails.

With Espresso, you create what is essentially a script of actions to take on each `View` and check against expected results. The key concepts are *locating* and then *interacting* with UI elements. These are the basic steps:

1. **Match a View:** Find a `View`.
2. **Perform an action:** Perform a click or other action that triggers an event with the `View`.
3. **Assert and verify the result:** Check the state of the `View` to see if it reflects the expected state or behavior defined by the assertion.

Hamcrest (an anagram of "matchers") is a framework that assists writing software tests in Java. To create a test, you create a method within the test class that uses Hamcrest expressions.

Tip: For more information about the Hamcrest matchers, see The Hamcrest Tutorial.

With Espresso you use the following types of Hamcrest expressions to help find `View` elements and interact with them:

- *ViewMatchers:* Hamcrest matcher expressions in the `ViewMatchers` class that lets you find a `View` in the current `View` hierarchy so that you can examine something or perform some action.
- *ViewActions:* Hamcrest action expressions in the `ViewActions` class that lets you perform an action on a `View` found by a `ViewMatcher`.
- *ViewAssertions:* Hamcrest assertion expressions in the `ViewAssertions` class that lets you assert or check the state of a `View` found by a `ViewMatcher`.

The following shows how all three expressions work together:

1. Use a `ViewMatcher` to find a `View`: `onView(withId(R.id.my_view))`
2. Use a `ViewAction` to perform an action: `.perform(click())`
3. Use a `ViewAssertion` to check if the result of the action matches an assertion: `.check(matches(isDisplayed()))`

The following shows how the above expressions are used together in a statement:

```
onView(withId(R.id.my_view))
        .perform(click())
        .check(matches(isDisplayed()));
```

2.1 Run the example test

Android Studio creates a blank Espresso test class when you create the project.

1. In the **Project > Android** pane, open **java > com.example.android.twoactivities (androidTest)**, and open **ExampleInstrumentedTest**.

   The project is supplied with this example test. You can create as many tests as you wish in this folder. In the next step you will edit the example test.

2. In the **Project > Android** pane, open **java > com.example.android.twoactivities (androidTest)**, and open **ExampleInstrumentedTest**.

3. To run the test, **right-click** (or **Control-click**) **ExampleInstrumentedTest** and choose **Run ExampleInstrumentedTest** from the pop-up menu. You can then choose to run the test on the emulator or on your device.

The Run pane at the bottom of Android Studio shows the progress of the test, and when finishes, it displays "Tests ran to completion." In the left column Android Studio displays "All Tests Passed".

2.2 Define a class for a test and set up the Activity

You will edit the example test rather than add a new one. To make the example test more understandable, you will rename the class from `ExampleInstrumentedTest` to `ActivityInputOutputTest`. Follow these steps:

1. **Right-click** (or **Control-click**) **ExampleInstrumentedTest** in the **Project > Android** pane, and choose **Refactor > Rename**.
2. Change the class name to **ActivityInputOutputTest**, and leave all options checked. Click **Refactor**.
3. Add the following to the top of the `ActivityInputOutputTest` class, before the first `@Test` annotation:

```
@Rule
public ActivityTestRule mActivityRule = new ActivityTestRule<>(
                    MainActivity.class);
}
```

The class definition now includes several annotations:

- `@RunWith`: To create an instrumented JUnit 4 test class, add the `@RunWith(AndroidJUnit4.class)` annotation at the beginning of your test class definition.
- `@Rule`: The `@Rule` annotation lets you add or redefine the behavior of each test method in a reusable way, using one of the test rule classes that the Android Testing Support Library provides, such as `ActivityTestRule` or `ServiceTestRule`. The rule above uses an `ActivityTestRule` object, which provides functional testing of a single `Activity`—in this case, `MainActivity.class`. During the duration of the test you will be able to manipulate your `Activity` directly, using `ViewMatchers`, `ViewActions`, and `ViewAssertions`.
- `@Test`: The `@Test` annotation tells JUnit that the `public void` method to which it is attached can be run as a test case. A test method begins with the `@Test` annotation and contains the code to exercise and verify a single function in the component that you want to test.

In the above statement, `ActivityTestRule` may turn red at first, but then Android Studio adds the following `import` statement automatically:

```
import android.support.test.rule.ActivityTestRule;
```

2.3 Test switching from one Activity to another

The TwoActivities app includes:

- `MainActivity`: Includes the `button_main` button for switching to `SecondActivity` and the `text_header_reply` view that serves as a text heading.
- `SecondActivity`: Includes the `button_second` button for switching to `MainActivity` and the `text_header` view that serves as a text heading.

When you have an app that switches from one `Activity` to another, you should test that capability. The TwoActivities app provides a text entry field and a **Send** button (the `button_main` id). Clicking **Send** launches `SecondActivity` with the entered text shown in the `text_header` view of `SecondActivity`.

But what happens if no text is entered? Will `SecondActivity` still appear?

The `ActivityInputOutputTest` class of tests will show that the `View` elements in `SecondActivity` appear regardless of whether text is entered.

Note: When you enter `ViewMatchers` and `ViewActions`, *don't* copy them from the text and paste them into the Android Studio editing window. Instead, enter the expressions directly, so that Android Studio properly imports the appropriate `ViewMatchers` and `ViewActions`.

Follow these steps to add your tests to `ActivityInputOutputTest`:

1. Add the beginning of the `activityLaunch()` method to `ActivityInputOutputTest`. This method will test whether the `SecondActivity` `View` elements appear when clicking the `Button`, and include the `@Test` notation on a line immediately above the method:

```
@Test
public void activityLaunch() {
```

2. Add a combined `ViewMatcher` and `ViewAction` expression to the `activityLaunch()` method to locate the `View` containing the `button_main` `Button`, and include a `ViewAction` expression to perform a click.

```
onView(withId(R.id.button_main)).perform(click());
```

The `onView()` method lets you use `ViewMatcher` arguments to find `View` elements. It searches the `View` hierarchy to locate a corresponding `View` instance that meets some given criteria—in this case, the `button_main` `View`. The `.perform(click())` expression is a `ViewAction` expression that performs a click on the `View`.

In the above `onView` statement, `onView`, `withID`, and `click` may turn red at first, but then Android Studio adds import statements for them.

3. Add another `ViewMatcher` expression to the `activityLaunch()` method to find the `text_header` `View` (which is in `SecondActivity`), and a `ViewAction` expression to perform a check to see if the `View` is displayed:

```
onView(withId(R.id.text_header)).check(matches(isDisplayed()));
```

This statement uses the `onView()` method to locate the `text_header` `View` for `SecondActivity` and then check to see if it is displayed after clicking the `button_main` `Button`. The `check()` method may turn red at first, but then Android Studio adds an `import` statement for it.

4. Add similar statements to test whether clicking the `button_second` `Button` in `SecondActivity` switches to `MainActivity`:

```
onView(withId(R.id.button_second)).perform(click());
onView(withId(R.id.text_header_reply)).check(matches(isDisplayed()));
```

5. Review the `activityLaunch()` method you just created in the `ActivityInputOutputTest` class. It should look like this:

```
@Test
public void activityLaunch() {
    onView(withId(R.id.button_main)).perform(click());
    onView(withId(R.id.text_header)).check(matches(isDisplayed()));
    onView(withId(R.id.button_second)).perform(click());
    onView(withId(R.id.text_header_reply)).check(matches(isDisplayed()));
}
```

6. To run the test, **right-click** (or **Control-click**) **ActivityInputOutputTest** and choose **Run ActivityInputOutputTest** from the pop-up menu. You can then choose to run the test on the emulator or on your device.

As the test runs, watch the test automatically start the app and click the `Button`. The `SecondActivity` `View` elements appear. The test then clicks the `Button` in `SecondActivity`, and `MainActivity` `View` elements appear.

The Run window (the bottom pane of Android Studio) shows the progress of the test, and when finishes, it displays "Tests ran to completion." In the left column Android Studio displays "All Tests Passed".

The drop-down menu next to the **Run** icon  in the Android Studio toolbar now shows the name of the test class (`ActivityInputOutputTest`). You can click the **Run** icon to run the test, or switch to **app** in the dropdown menu and then click the **Run** icon to run the app.

2.4 Test text input and output

In this step you will write a test for text input and output. The TwoActivities app uses the `editText_main` `EditText` for input, the `button_main` `Button` for sending the input to `SecondActivity`, and the `TextView` in `SecondActivity` that shows the output in the field with the id `text_message`.

1. Add another `@Test` annotation and a new `textInputOutput()` method to test text input and output:

```
@Test
public void textInputOutput() {
   onView(withId(R.id.editText_main)).perform(typeText("This is a test."));
   onView(withId(R.id.button_main)).perform(click());
}
```

The above method uses a `ViewMatcher` to locate the `View` containing the `editText_main` `EditText`, and a `ViewAction` to enter the text `"This is a test."` It then uses another `ViewMatcher` to find the `View` with the `button_main` `Button`, and another `ViewAction` to click the `Button`.

2. Add a `ViewMatcher` to the method to locate the `text_message` `TextView` in `SecondActivity`, and a `ViewAssertion` to see if the output matches the input to test that the message was correctly sent—it should match `"This is a test."` (Be sure to include the period at the end.)

```
onView(withId(R.id.text_message))
                        .check(matches(withText("This is a test.")));
```

3. Run the test.

As the test runs, the app starts and the text is automatically entered as input; the `Button` is clicked, and `SecondActivity` appears with the text.

The bottom pane of Android Studio shows the progress of the test, and when finished, it displays "Tests ran to completion." In the left column Android Studio displays "All Tests Passed". You have successfully tested the `EditText` input, the **Send** `Button`, and the `TextView` output.

2.5 Introduce an error to show a test failing

Introduce an error in the test to see what a failed test looks like.

1. Change the `matches` check on the `text_message` view from `"This is a test."` to `"This is a failing test."`:

```
onView(withId(R.id.text_message)).check(matches(withText("This is a failing
test.")));
```

2. Run the test again. This time you will see the message in red, "1 test failed", above the bottom pane, and a red exclamation point next to `textInputOutput` in the left column. Scroll the bottom pane to the message "Test running started" and see that all of the results after that point are in red. The very next statement after "Test running started" is:

```
android.support.test.espresso.base.DefaultFailureHandler$AssertionFailedWith
CauseError: 'with text: is "This is a failing test."' doesn't match the
selected view.
Expected: with text: is "This is a failing test."
Got: "AppCompatTextView{id=2131165307, res-name=text_message ...
```

Other fatal error messages appear after the above, due to the cascading effect of a failure leading to other failures. You can safely ignore them and fix the test itself.

Task 2 solution code

See `ActivityInputOutputTest.java` in the Android Studio project: TwoActivitiesEspresso

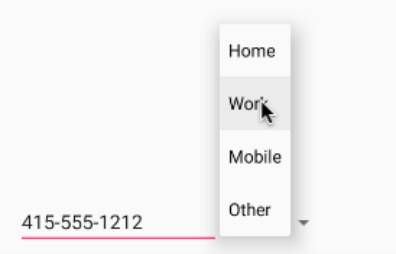
Task 3: Test the display of spinner selections

The Espresso `onView()` method finds a `View` that you can test. This method will find a `View` in the current `View` hierarchy. But you need to be careful—in an `AdapterView` such as a `Spinner`, the `View` is typically dynamically populated with *child* `View` elements at runtime. This means there is a possibility that the `View` you want to test may not be in the `View` hierarchy at that time.

The Espresso API handles this problem by providing a separate `onData()` entry point, which is able to first load the adapter item and bring it into focus prior to locating and performing actions on any of its children.

PhoneNumberSpinner is a starter app you can use to test a `Spinner`. It shows a `Spinner`, with the id `label_spinner`, for choosing the label of a phone number (**Home**, **Work**, **Mobile**, and **Other**). It then displays the phone number and `Spinner` choice in a `TextView`.

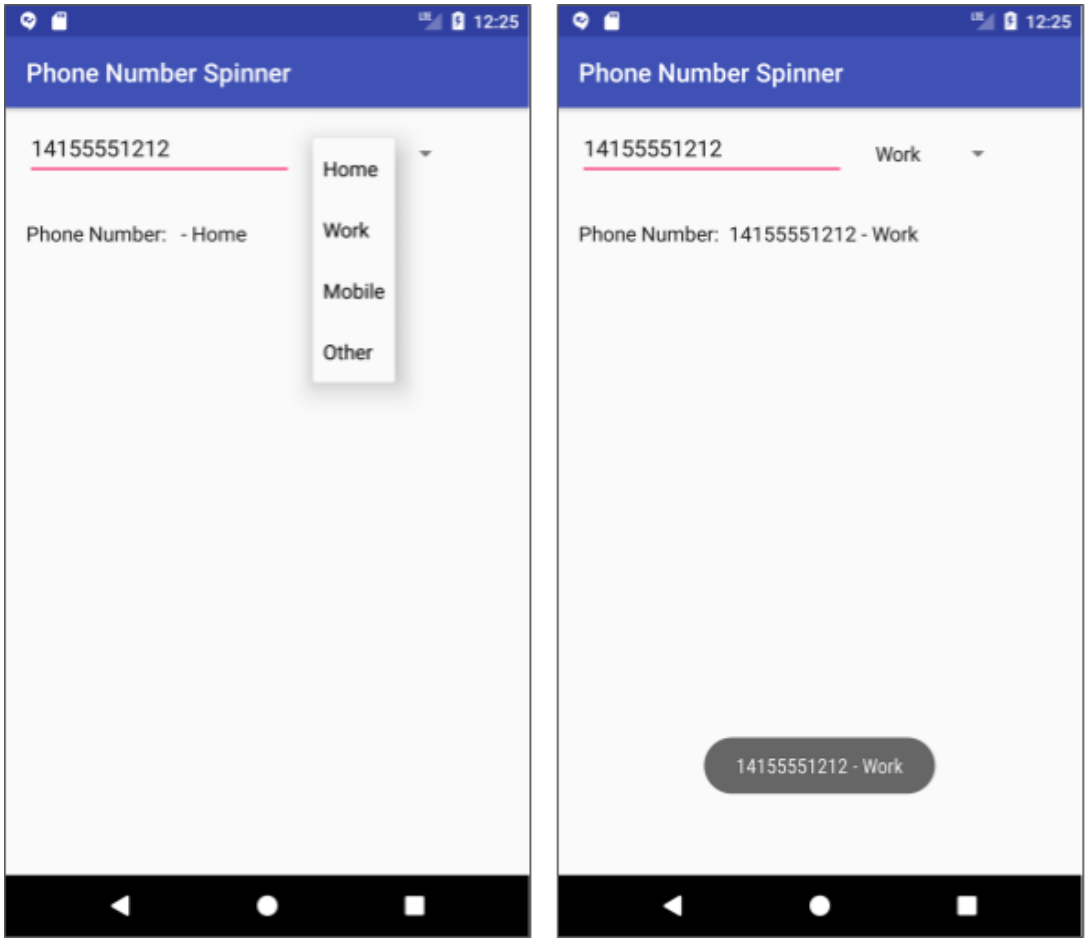
The goal of this test is to open the `Spinner`, click each item, and then verify that the `TextView` `text_phonelabel` contains the item. The test demonstrates that the code retrieving the `Spinner` selection is working properly, and the code displaying the text of the `Spinner` item is also working properly. You will write the test using string resources and iterate through the `Spinner` items so that the test works no matter how many items are in the `Spinner`, or how those items are worded; for example, the words could be in a different language.



3.1 Create the test method

1. Download the PhoneNumberSpinnerEspresso project and then open the project in Android Studio.

2. Run the app. Enter a phone number, and choose a label from the `Spinner` as shown on the left side of the figure below. The result should appear in the `TextView` and in a `Toast` message, as shown on the right side of the figure.



3. Expand **com.example.android.phonenumberspinner (androidTest)**, and open **ExampleInstrumentedTest**.

4. Rename `ExampleInstrumentedTest` to `SpinnerSelectionTest` in the class definition, and add the following:

```
@RunWith(AndroidJUnit4.class)
public class SpinnerSelectionTest {
   @Rule
```

```
   public ActivityTestRule mActivityRule = new ActivityTestRule<>(
                    MainActivity.class);
```

3.2 Access the array used for the Spinner items

You want the test to click each item in the `Spinner` based on the number of elements in the array. But how do you access the array?

1. Create the `iterateSpinnerItems()` method as `public` returning `void`, and assign the array used for the `Spinner` items to a new array to use within the `iterateSpinnerItems()` method:

```
@Test
public void iterateSpinnerItems() {
   String[] myArray =
         mActivityRule.getActivity().getResources()
         .getStringArray(R.array.labels_array);
}
```

   In the statement above, the test accesses the array (with the id `labels_array`) by establishing the context with the `getActivity()` method of the `ActivityTestRule` class, and getting a resources instance using `getResources()`.

2. Assign the length of the array to `size`, and start the beginning of a `for` loop using the `size` as the maximum number for a counter.

```
int size = myArray.length;
    for (int i=0; i<size; i++) {
```

3.3 Locate Spinner items and click on them

1. Add an `onView()` statement within the `for` loop to find the `Spinner` and click on it:

```
// Find the spinner and click on it.
onView(withId(R.id.label_spinner)).perform(click());
```

A user must click the `Spinner` itself in order to click any item in the `Spinner`, so your test must click the `Spinner` first before clicking the item.

2. Write an `onData()` statement to find and click a `Spinner` item:

```
// Find the spinner item and click on it.
onData(is(myArray[i])).perform(click());
```

The above statement matches if the object is a specific item in the `Spinner`, as specified by the `myArray[i]` array element.

If `onData` appears in red, click the word, and then click the red light bulb icon that appears in the left margin. Choose the following in the pop-up menu: **Static import method 'android.support.test.espresso.Espresso.onData'**

If `is` appears in red, click the word, and then click the red light bulb icon that appears in the left margin. Choose the following in the pop-up menu: **Static import method...> Matchers.is (org.hamcrest)**

1. Add another `onView()` statement to the `for` loop to check to see if the resulting `text_phonelabel` matches the `Spinner` item specified by `myArray[i]`.

```
// Find the text view and check that the spinner item
```

```
// is part of the string.
onView(withId(R.id.text_phonelabel))
                    .check(matches(withText(containsString(myArray[i]))));
```

If `containsString` appears in red, click the word, and then click the red light bulb icon that appears in the left margin. Choose the following in the pop-up menu: **Static import method...> Matchers.containsString (org.hamcrest)**

2. Run the test.

The test runs the app, clicks the `Spinner`, and "exercises" the `Spinner`—it clicks each `Spinner` item from top to bottom, checking to see if the item appears in the `text_phonelabel` `TextView`. It doesn't matter how many `Spinner` items are defined in the array, or what language is used for the `Spinner` items—the test performs all of them and checks their output against the array.

The bottom pane of Android Studio shows the progress of the test, and when finished, it displays "Tests ran to completion." In the left column Android Studio displays "All Tests Passed".

Task 3 solution code

See `SpinnerSelectionTest.java` in the Android Studio project: PhoneNumberSpinnerEspresso

Task 4: Record an Espresso test

Android Studio lets you *record* an Espresso test, which is useful for generating tests quickly. While recording a test, you use your app as a normal user—as you click through the app UI, editable test code is generated for you. You can also add assertions to check if a `View` holds a certain value.

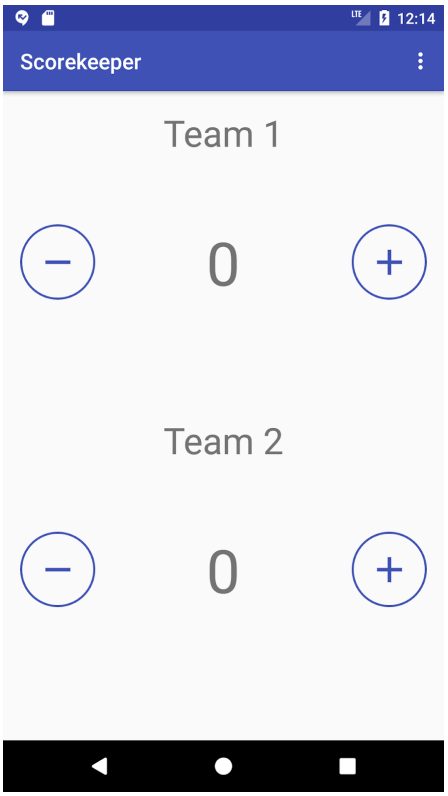
Recording Espresso tests, rather than coding the tests by hand, ensures that your app gets UI test coverage on areas that might take too much time or be too difficult to code by hand.

To demonstrate test recording, you will record a test of the Scorekeeper app created in the practical on using drawables, styles, and themes.

4.1 Open and run the app

1. Download the Scorekeeper project that you created in Android fundamentals 5.1: Drawables, styles, and themes.
2. Open the Scorekeeper project in Android Studio.
3. Run the app to ensure that it runs properly.

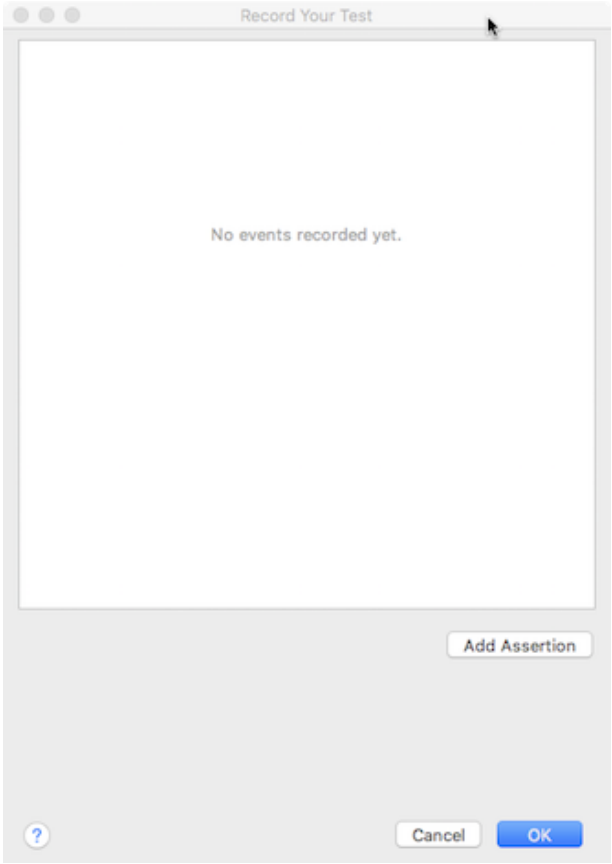
The Scorekeeper app consists of two sets of `Button` elements and two `TextView` elements, which are used to keep track of the score for any point-based game with two players.



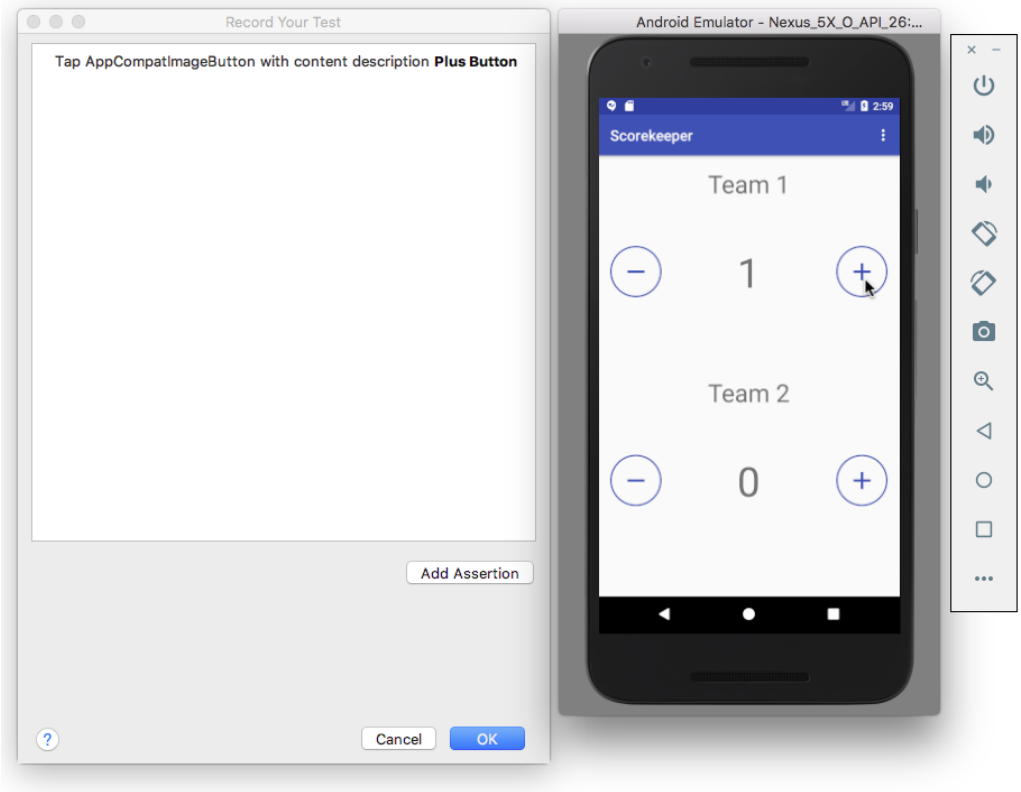
4.2 Record the test

1. Choose **Run > Record Espresso Test**, select your deployment target (an emulator or a device), and click **OK**.

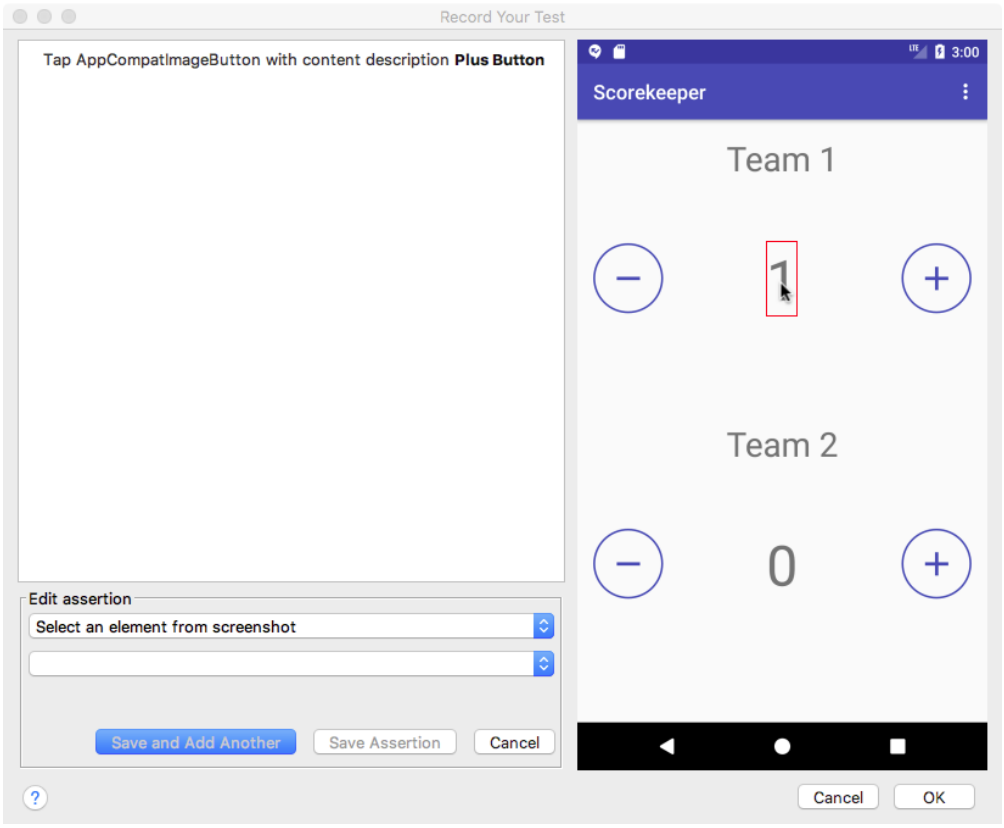
   The Record Your Test dialog appears, and the Debugger pane appears at the bottom of the Android Studio window. If you are using an emulator, the emulator also appears.



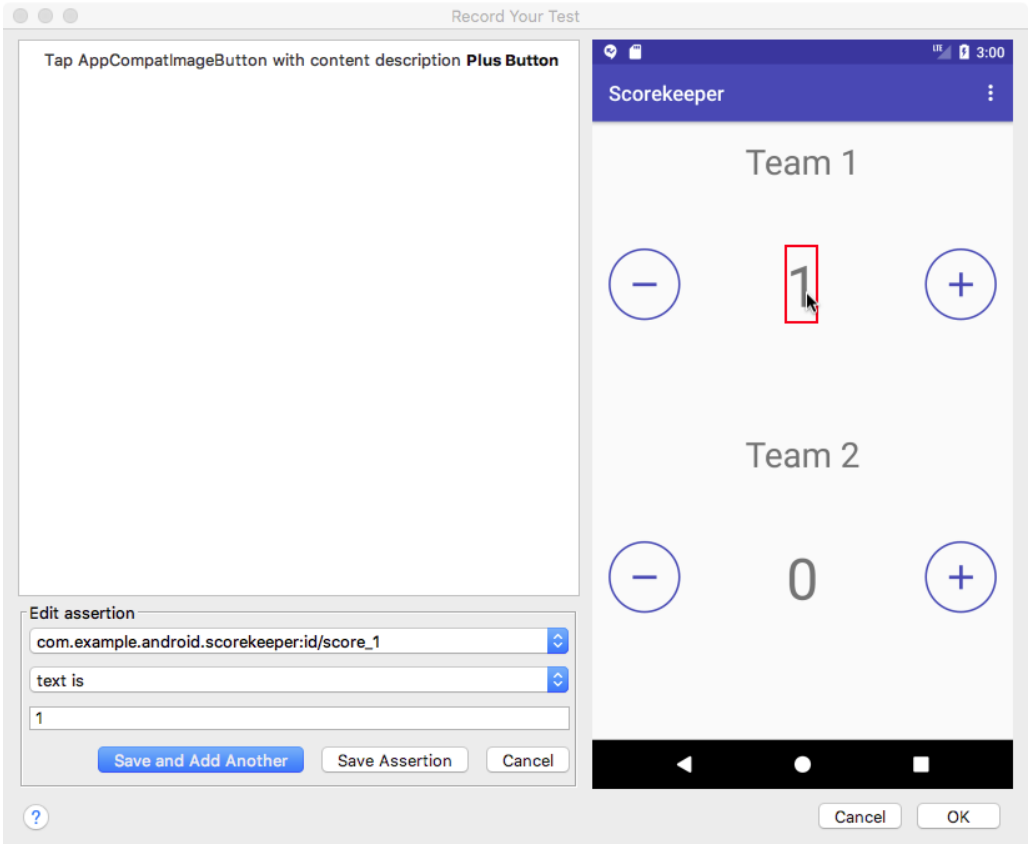
2. On the emulator or device, tap the plus (**+**) `ImageButton` for Team 1 in the app. The Record Your Test window shows the action that was recorded ("Tap AppCompatImageButton with the content description **Plus Button**").



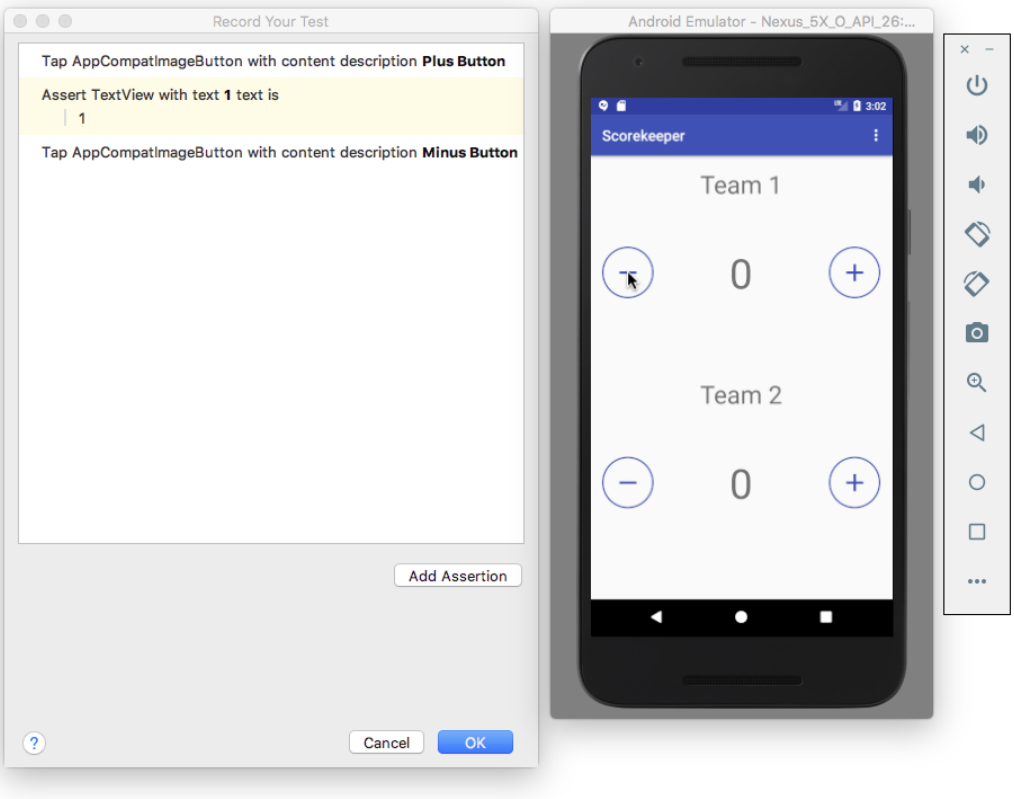
1. Click **Add Assertion** in the Record Your Test window. A screenshot of the app's UI appears in a pane on the right side of the window, and the **Select an element from screenshot** option appears in the dropdown menu. Select the score (**1**) in the screenshot as the UI element you want to check, as shown in the figure below.



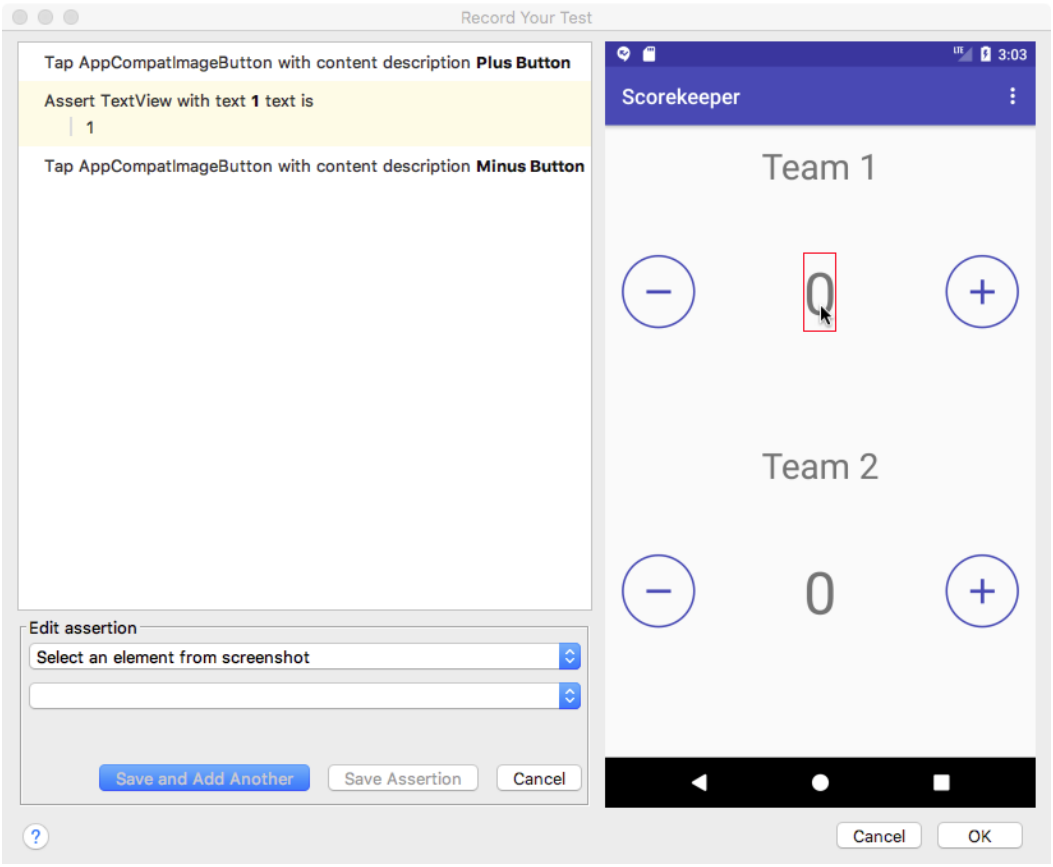
2. Choose **text is** from the second dropdown menu, as shown in the figure below. The text you expect to see (**1**) is already entered in the field below the dropdown menu.



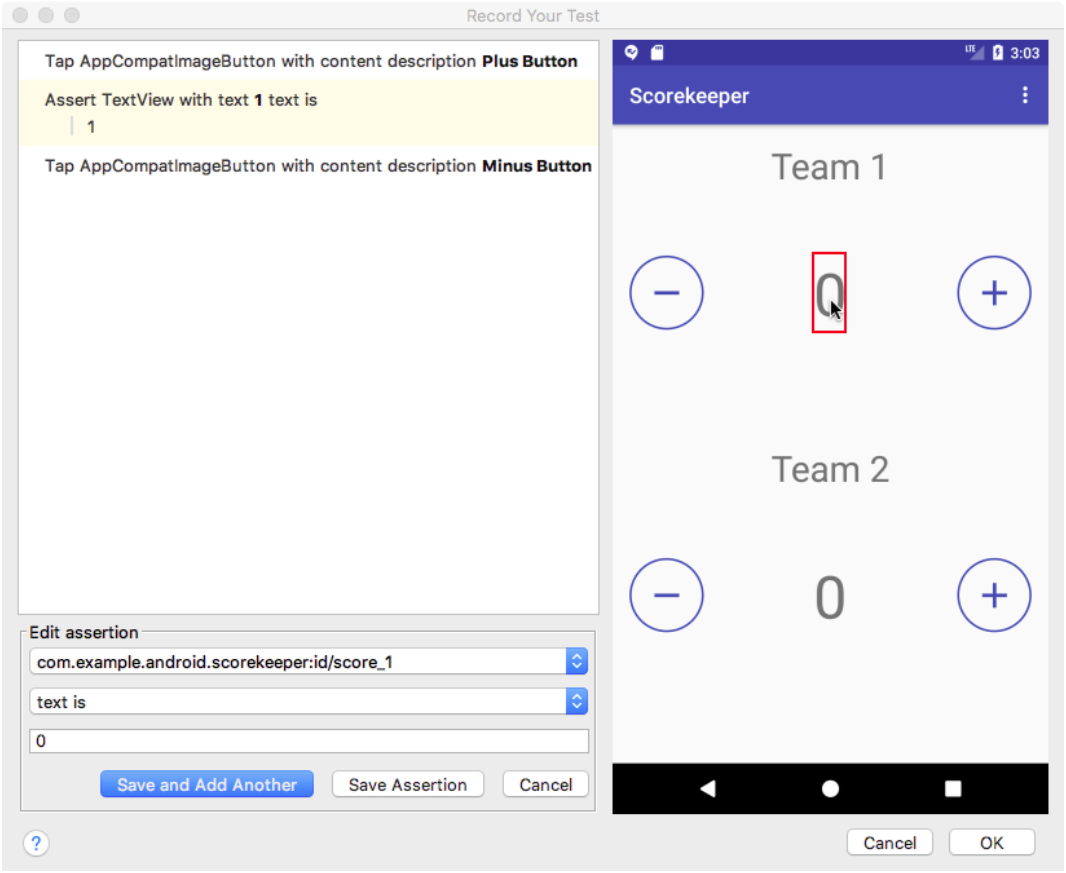
3. Click **Save Assertion**.
4. To record another click, on your emulator or device tap the minus (**–**) `ImageButton` for Team 1 in the app. The Record Your Test window shows the action that was recorded ("Tap AppCompatImageButton with the content description **Minus Button**").



5. Click **Add Assertion** in the Record Your Test window. The app's UI appears in the right-side pane as before. Select the score (**0**) in the screenshot as the UI element you want to check.



6. Choose **text is** from the second dropdown menu, as shown in the figure below. The text you expect to see (**0**) is already entered in the field below the dropdown menu.



7. Click **Save Assertion**, and then click **OK**.

8. In the dialog that appears, edit the name of the test to **ScorePlusMinusTest** so that it is easy for others to understand the purpose of the test.

9. Android Studio may display a request to add more dependencies to your Gradle Build file. Click **Yes** to add the dependencies. Android Studio adds the following to the dependencies section of the `build.gradle (Module: app)` file:

```
androidTestCompile
        'com.android.support.test.espresso:espresso-contrib:2.2.2', {
   exclude group: 'com.android.support', module: 'support-annotations'
   exclude group: 'com.android.support', module: 'support-v4'
```

```
    exclude group: 'com.android.support', module: 'design'
    exclude group: 'com.android.support', module: 'recyclerview-v7'
}
```

10. Expand **com.example.android.scorekeeper (androidTest)** to see the test, and run the test. It should pass. Run it again, and it should pass again.

The following is the test, as recorded in the `ScorePlusMinusTest.java` file:

```
// ... Package and import statements
@LargeTest
@RunWith(AndroidJUnit4.class)
public class ScorePlusMinusTest {

    @Rule
    public ActivityTestRule<MainActivity> mActivityTestRule
                        = new ActivityTestRule<>(MainActivity.class);

    @Test
    public void scorePlusMinusTest() {
        ViewInteraction appCompatImageButton = onView(
                allOf(withId(R.id.increaseTeam1),
                withContentDescription("Plus Button"),
                        childAtPosition(
                                childAtPosition(
                withClassName(is("android.widget.LinearLayout")),
                                        0),
                                3),
                        isDisplayed()));
        appCompatImageButton.perform(click());

        ViewInteraction textView = onView(
                allOf(withId(R.id.score_1), withText("1"),
                        childAtPosition(
                                childAtPosition(IsInstanceOf
               .<View>instanceOf(android.widget.LinearLayout.class),
                                        0),
                                2),
                        isDisplayed()));
        textView.check(matches(withText("1")));

        ViewInteraction appCompatImageButton2 = onView(
                allOf(withId(R.id.decreaseTeam1),
                        withContentDescription("Minus Button"),
```

```
                        childAtPosition(
                                childAtPosition(
                withClassName(is("android.widget.LinearLayout")),
                                        0),
                                1),
                        isDisplayed()));
        appCompatImageButton2.perform(click());

        ViewInteraction textView2 = onView(
                allOf(withId(R.id.score_1), withText("0"),
                        childAtPosition(
                                childAtPosition(IsInstanceOf
                .<View>instanceOf(android.widget.LinearLayout.class),
                                        0),
                                2),
                        isDisplayed()));
        textView2.check(matches(withText("0")));

    }

    private static Matcher<View> childAtPosition(
            final Matcher<View> parentMatcher, final int position) {

        return new TypeSafeMatcher<View>() {
            @Override
            public void describeTo(Description description) {
                description.appendText("Child at position "
                + position + " in parent ");
                parentMatcher.describeTo(description);
            }

            @Override
            public boolean matchesSafely(View view) {
                ViewParent parent = view.getParent();
                return parent instanceof ViewGroup
                        && parentMatcher.matches(parent)
                        && view.equals(((ViewGroup) parent)
                        .getChildAt(position));
            }
        };
    }
}
```

The test uses the `ViewInteraction` class, which is the primary interface for performing actions or assertions on `View` elements, providing both `check()` and `perform()` methods. Examine the test code to see how it works:

- **Perform**: The code below uses a method called `childAtPosition()`, which is defined as a custom `Matcher`, and the `perform()` method to click an `ImageButton`:

```
ViewInteraction appCompatImageButton = onView(
      allOf(withId(R.id.increaseTeam1),
              withContentDescription("Plus Button"),
                        childAtPosition(
                                childAtPosition(
              withClassName(is("android.widget.LinearLayout")),
                                        0),
                                3),
                        isDisplayed()));
appCompatImageButton.perform(click());
```

- **Check whether it matches the assertion**: The code below also uses the `childAtPosition()` custom `Matcher`, and checks to see if the clicked item matches the assertion that it should be "`1`":

```
ViewInteraction textView = onView(
                allOf(withId(R.id.score_1), withText("1"),
                        childAtPosition(
                                childAtPosition(
                                        IsInstanceOf
          .<View>instanceOf(android.widget.LinearLayout.class),
                                        0),
                                2),
                        isDisplayed()));
textView.check(matches(withText("1")));
```

- Custom `Matcher`: The code above uses `childAtPosition()`, which is defined as a custom `Matcher`:

```
private static Matcher<View> childAtPosition(
       final Matcher<View> parentMatcher, final int position) {

        return new TypeSafeMatcher<View>() {
            @Override
            public void describeTo(Description description) {
                description.appendText("Child at position "
                        + position + " in parent ");
                parentMatcher.describeTo(description);
            }

            @Override
            public boolean matchesSafely(View view) {
                ViewParent parent = view.getParent();
                return parent instanceof ViewGroup
                        && parentMatcher.matches(parent)
                        && view.equals(((ViewGroup) parent)
                        .getChildAt(position));
            }
        };
}
```

The custom `Matcher` in the above code extends the abstract `TypeSafeMatcher` class.

You can record multiple interactions with the UI in one recording session. You can also record multiple tests, and edit the tests to perform more actions, using the recorded code as a snippet to copy, paste, and edit.

Task 4 solution code

See `ScorePlusMinusTest.java` in the Android Studio project: ScorekeeperEspresso

Coding challenge

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

You learned how to create a `RecyclerView` in another practical. The app lets you scroll a list of words from "Word 1" to "Word 19". When you tap the `FloatingActionButton`, a new word appears in the list ("+ Word 20").

Like an `AdapterView` (such as a `Spinner`), a `RecyclerView` dynamically populates child `View` elements at runtime. But a `RecyclerView` is not an `AdapterView`, so you can't use `onData()` to interact with list items as you did in the previous task with a `Spinner`. What makes a `RecyclerView` complicated from the point of view of Espresso is that `onView()` can't find the child `View` if it is off the screen.

Challenge: Fortunately, you can record an Espresso test of using the `RecyclerView`. Record a test that taps the `FloatingActionButton`, and check to see if a new word appears in the list ("+ Word 20").

Tip: To add another test that clicks a `RecyclerView` item, don't record the test. Instead, use a class called `RecyclerViewActions`, which exposes a small API that you can use to operate on a `RecyclerView`.

Challenge solution code

See `RecyclerViewTest.java` in the Android Studio project: RecyclerViewEspresso

Summary

To set up Espresso to test an Android Studio project:

- In Android Studio, check for and install the Android Support Repository.

- Add `dependencies` to the **build.gradle (Module: app)** file:

```
testImplementation 'junit:junit:4.12'
androidTestImplementation 'com.android.support.test:runner:1.0.1'
androidTestImplementation
           'com.android.support.test.espresso:espresso-core:3.0.1'
```

- Add the following instrumentation statement to the end of the `defaultConfig` section:

```
testInstrumentationRunner
                "android.support.test.runner.AndroidJUnitRunner"
```

Instrumentation is a set of control methods, or hooks, in the Android system.

- On your test device, turn off animations. To do this, turn on USB Debugging. Then in the Settings app, select **Developer Options > Drawing**. Turn off window animation scale, transition animation scale, and animator duration scale.

To test annotations:

- `@RunWith(AndroidJUnit4.class)`: Create an instrumented JUnit 4 test class.
- `@Rule`: Add or redefine the behavior of each test method in a reusable way, using one of the test rule classes that the Android Testing Support Library provides, such as `ActivityTestRule` or `ServiceTestRule`.
- `@Test`: The `@Test` annotation tells JUnit that the `public void` method to which it is attached can be run as a test case.

To test code:

- `ViewMatchers` class lets you find a view in the current `View` hierarchy so that you can examine something or perform an action.

- `ViewActions` class lets you perform an action on a view found by a `ViewMatcher`.
- `ViewAssertions` class lets you assert or check the state of a view found by a `ViewMatcher`.

To test a spinner:

- Use `onData()` with a `View` that is dynamically populated by an adapter at runtime.
- Get items from an app's array by establishing the context with `getActivity()` and getting a resources instance using `getResources()`.
- Use `onData()` to find and click each spinner item.
- Use `onView()` with a `ViewAction` and `ViewAssertion` to check if the output matches the selected spinner item.

Related concept

The related concept documentation is in 6.1: UI testing.

Learn more

Android Studio Documentation:

- Test Your App
- Espresso basics
- Espresso cheat sheet

Android Developer Documentation:

- Best Practices for Testing
- Getting Started with Testing
- Testing UI for a Single App

- Building Instrumented Unit Tests
- Espresso Advanced Samples
- The Hamcrest Tutorial
- Hamcrest API and Utility Classes
- Test Support APIs

Android Testing Support Library:

- Espresso documentation
- Espresso Samples

Videos

- Android Testing Support - Android Testing Patterns #1 (introduction)
- Android Testing Support - Android Testing Patterns #2 (onView view matching)
- Android Testing Support - Android Testing Patterns #3 (onData and adapter views)

Other:

- Google Testing Blog: Android UI Automated Testing
- Atomic Object: "Espresso – Testing RecyclerViews at Specific Positions"
- Stack Overflow: "How to assert inside a RecyclerView in Espresso?"
- GitHub: Android Testing Samples
- Google Codelabs: Android Testing Codelab

Homework

Build and run an app

Record another Espresso test for the ScorekeeperEspresso app. This test should tap the **Night Mode** option in the options menu and determine whether the **Day Mode** option appears in its place. The test should then tap the **Day Mode** option and determine whether the **Night Mode** option appears in its place.

Answer these questions

Question 1

Which steps do you perform to test a `View` interaction, and in what order? Choose one:

- Match a `View`, assert and verify the result, and perform an action.
- Match a `View`, perform an action, and assert and verify the result.
- Perform an action, match a view, and assert and verify the result.
- Perform an action, and assert and verify the result.

Question 2

Which of the following annotations enables an instrumented JUnit 4 test class? Choose one:

- `@RunWith`
- `@Rule`
- `@Test`
- `@RunWith` and `@Test`

Question 3

Which method would you use to find a child `View` in an `AdapterView`? Choose one:

- `onData()` to load the adapter and enable the child `View` to appear on the screen.
- `onView()` to load the `View` from the current `View` hierarchy.
- `onView().check()` to check the current `View`.
- `onView().perform()` to perform a click on the current `View`.

Submit your app for grading

Guidance for graders

Check that the test meets the following criteria:

- The test appears in the `com.example.android.scorekeeper (androidTest)` folder.
- The test automatically switches the app from Day Mode to Night Mode, and then back to Day Mode.
- The test passes more than once.