Unit 2: User experience

Lesson 4: User interaction

4.1: Buttons and clickable images

Contents:

- Designing for interactivity
- Designing buttons
- Responding to button-click events
- Using clickable images
- Using a floating action button
- Recognizing gestures
- Related practical
- Learn more

Designing for interactivity

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. Users tap these elements on a touchscreen or click them using a pointing device. Any element that users tap or click to perform an action is called a *clickable* element.
For an Android app, user interaction typically involves tapping, typing, using gestures, or talking. The Android framework provides corresponding user interface (UI) elements such as buttons, clickable images, menus, keyboards, text entry fields, and a microphone.

When designing an interactive app, make sure your app is intuitive; that is, your app should perform as your users expect it to perform. For example, when you rent a car, you expect the steering wheel, gear shift, headlights, and indicators to be in a certain place. Another example is that when you first enter a room, you expect the light switch to be in a certain place. Similarly, when a user starts an app, the user expects buttons and images to be clickable. Don't violate established expectations, or you'll make it harder for your users to use your app.

Note: Android users expect UI elements to act in certain ways, so it's important that your app be consistent with other Android apps. To satisfy your users, create a layout that gives users predictable choices.

In this chapter you learn how to create buttons and clickable images for triggering actions.

Designing buttons

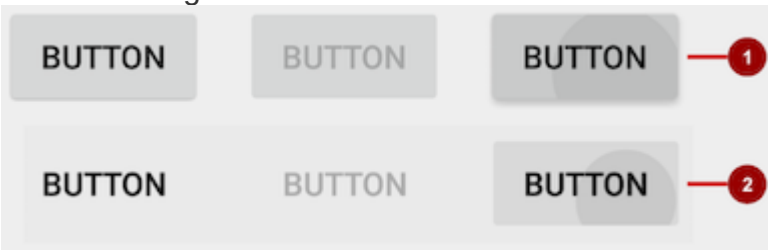
People like to press buttons. Show someone a big red button with a message that says "Do not press" and the person will probably press the button, just for the pleasure of pressing a big red button. (That the button is forbidden is also a factor.)

You use the `Button` class to make a button for an Android app. Buttons can have the following design:

- Text only, as shown on the left side of the figure below.
- Icon only, as shown in the center of the figure below.
- Both text and an icon, as shown on the right side of the figure below.

When the user touches or clicks a button, the button performs an action. The button's text or icon should provide a hint about what that action will be. (Buttons are sometimes called "push-buttons" in Android documentation.)

A button is usually a rectangle or rounded rectangle with a descriptive caption or icon in its center. Android `Button` elements follow the guidelines in the Android Material Design specification. (You learn more about Material Design in another lesson.) Android offers several types of `Button` elements, including raised buttons and flat buttons as shown in the figure below. Each button has three states: normal, disabled, and pressed.



In the figure above:

1. Raised button in three states: normal, disabled, and pressed
2. Flat button in three states: normal, disabled, and pressed

Designing raised buttons

A *raised button* is an outlined rectangle or rounded rectangle that appears lifted from the screen—the shading around it indicates that it is possible to tap or click it. The raised button can show text, an icon, or both.

To use raised buttons that conform to the Material Design specification, follow these steps:

1. If your **build.gradle (Module: app)** file doesn't include the `android.support:appcompat-v7` library, add it to the `dependencies` section:

```
compile 'com.android.support:appcompat-v7:26.1.0.'
```

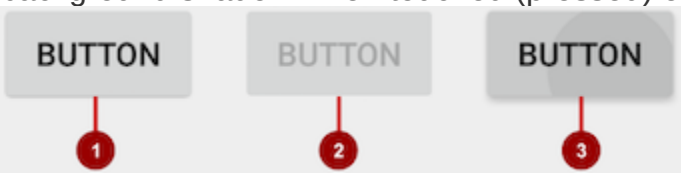
   In the snippet above, `26.1.0` is the version number. If the version number you specified is lower than the currently available library version number, Android Studio will warn you ("a newer version is available"). Update the version number to the one Android Studio tells you to use.
3. Make your `Activity` extend `android.support.v7.app.AppCompatActivity`:

```
public class MainActivity extends AppCompatActivity {
    // ...
}
```

7. Use the `Button` element in the layout file. A raised `Button` is the default style.

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    <!-- more attributes ... -->
/>
```

Use raised `Button` elements to give more prominence to actions in layouts with a lot of varying content. A raised `Button` adds dimension to a flat layout—it shows a background shadow when touched (pressed) or clicked, as shown below.



In the figure above:

1. Normal state: A raised `Button`.
2. Disabled state: When disabled, the `Button` is dimmed out and not active in the app's context. In most cases you would hide an inactive `Button`, but there may be times when you would want to show it as disabled.
3. Pressed state: The pressed state, with a larger background shadow, indicates that the `Button` is being touched or clicked. When you attach a callback to the `Button` (such as the `android:onClick` attribute), the callback is called when the `Button` is in this state.

Creating a raised button with text

Some raised `Button` elements are best designed as text, without an icon, such as a **Save** button, because an icon by itself might not convey an obvious meaning. The `Button` class extends the `TextView` class. To use it, add it to the XML layout:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    <!-- more attributes ... -->
/>
```

The best practice with a text `Button` is to define a very short word as a string resource (`button_text` in the example above), so that the string can be translated. For example, **Save** could be translated into French as **Enregistrer** without changing any of the code.

Creating a raised button with an icon and text

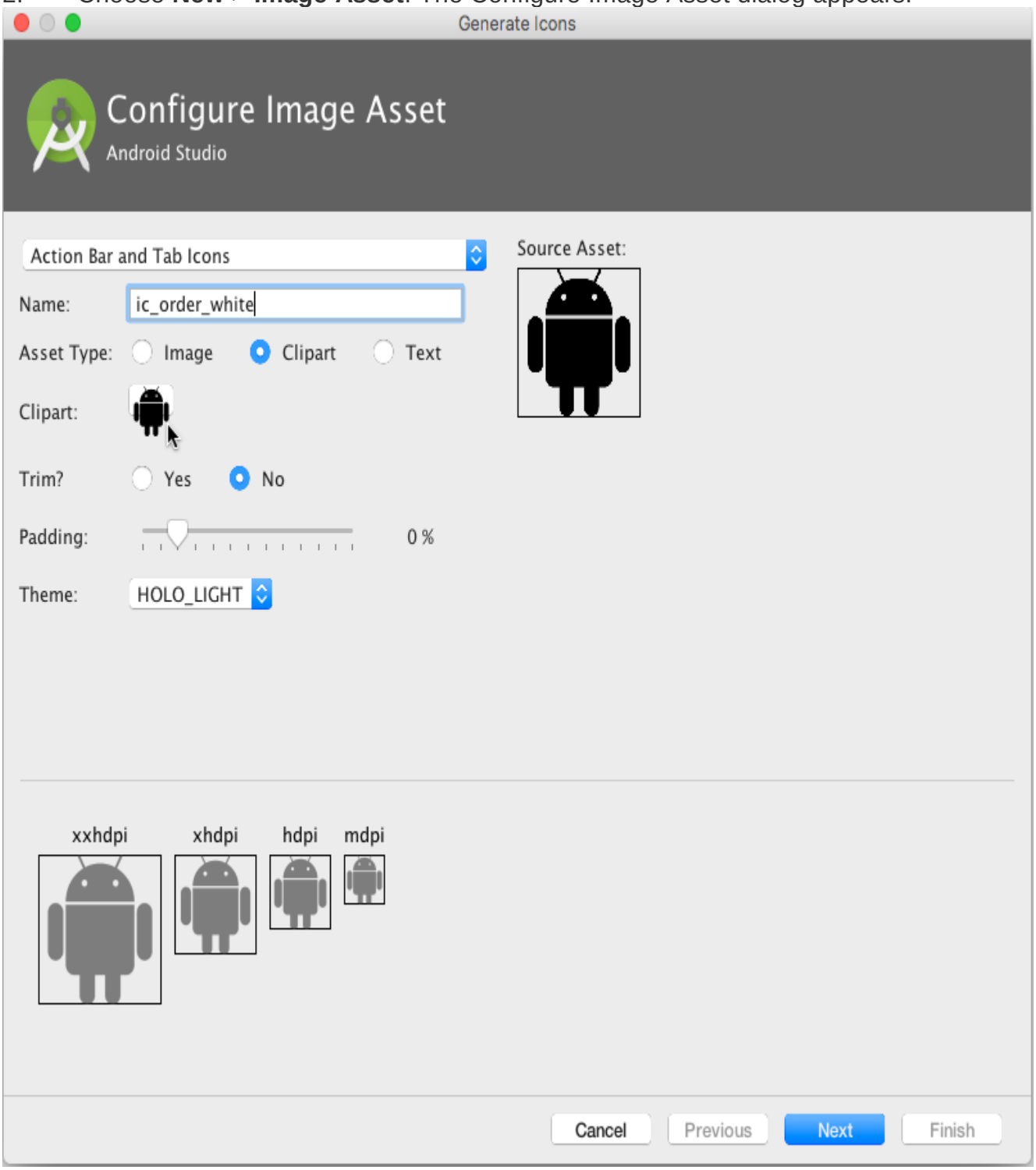
While a `Button` usually displays text that tells the user what the action is, a raised `Button` can also display an icon along with text.

Choosing an icon

To choose images of a standard icon that are resized for different displays, follow these steps:

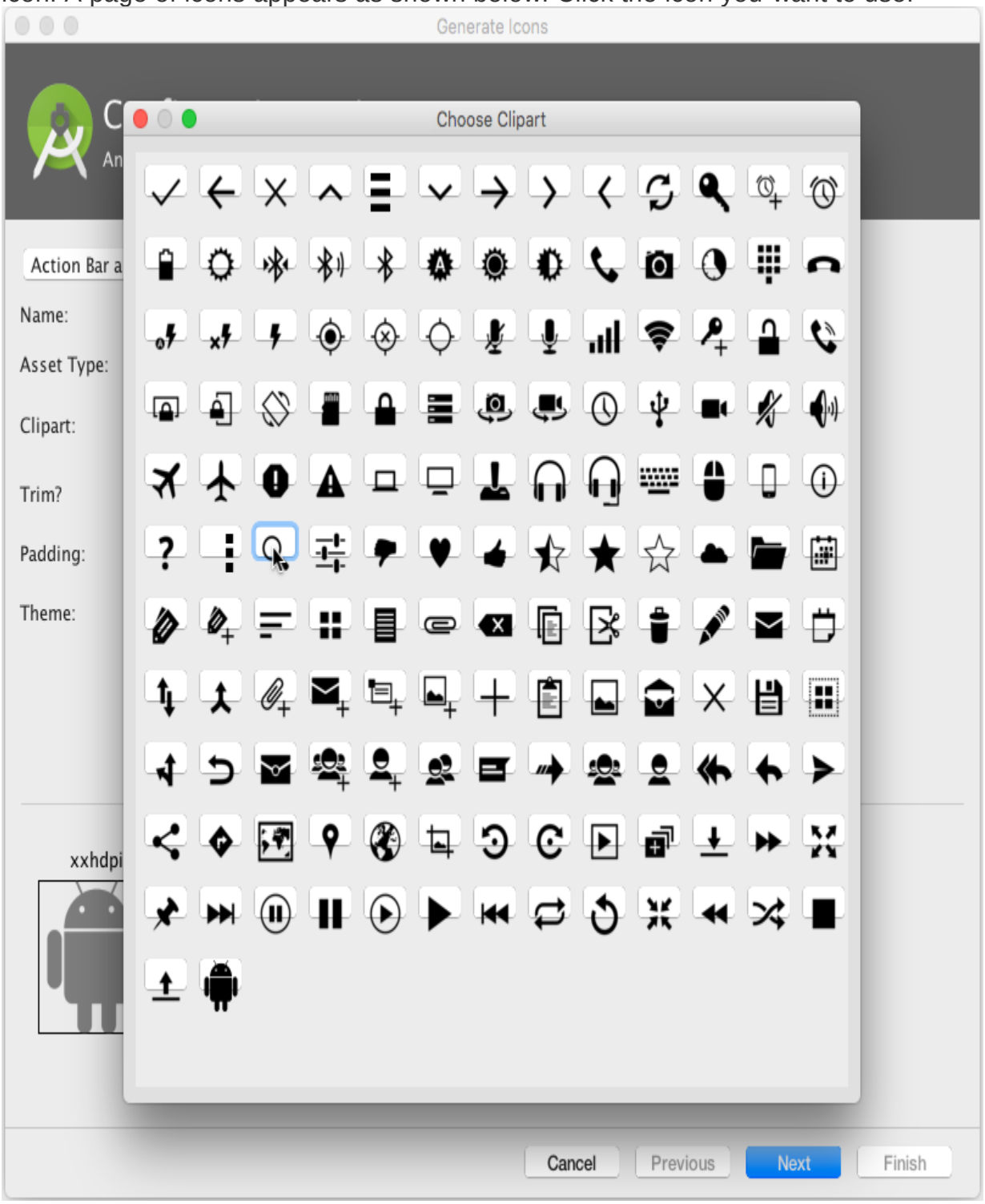
1. Expand **app > res** in the **Project > Android** pane, and right-click (or Command-click) the **drawable** folder.

2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.



3. Choose **Action Bar and Tab Icons** in the drop-down menu. (For a complete description of this dialog, see Create app icons with Image Asset Studio.)

4. Click the **Clipart:** image (the Android logo) to select a clip art image as the icon. A page of icons appears as shown below. Click the icon you want to use.



5. Optional: Choose **HOLO_DARK** from the **Theme** drop-down menu to set the icon to be white against a dark-colored or black background.

6. Optional: Depending on the shape of the icon, you may want to add padding to the icon so that the icon doesn't crowd the text. Drag the Padding slider to the right to add more padding.
7. Click **Next**, and then click **Finish** in the Confirm Icon Path dialog. The icon name should now appear in the **app > res > drawable** folder.

Vector images of a standard icon are automatically resized for different sizes of device displays. To choose vector images, follow these steps:

1. Expand **app > res** in the **Project > Android** pane, and right-click (or Command-click) the **drawable** folder.
2. Choose **New > Vector Asset** for an icon that automatically resizes itself for each display.
3. The Vector Asset Studio dialog appears for a vector asset. Click the **Material Icon** radio button, and then click the **Choose** button to choose an icon from the Material Design specification. (For a complete description of this dialog, see Add Multi-Density Vector Graphics.)
4. Click **Next** after choosing an icon, and click **Finish** to finish. The icon name should now appear in the **app > res > drawable** folder.

Adding the button with text and icon to the layout

To create a button with text and an icon as shown in the figure below, use a `Button` in your XML layout. Add the `android:drawableLeft` attribute to draw the icon to the left of the button's text, as shown in the figure below:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    android:drawableLeft="@drawable/button_icon"
    <!-- more attributes ... -->
/>
```

Creating a raised button with only an icon

If the icon is universally understood, you may want to use it instead of text.

To create a raised button with just an icon or image (no text), use the `ImageButton` class, which extends the `ImageView` class. You can add an `ImageButton` to your XML layout as follows:

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/button_icon"
    <!-- more attributes ... -->
/>
```

Changing the style and appearance of raised buttons

The simplest way to show a more prominent raised button is to use a different background color for the button. You can specify the `android:background` attribute with a drawable or color resource:

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

The appearance of your button—the background color and font—may vary from one device to another, because devices by different manufacturers often have different default styles for input controls. You can control exactly how your buttons and other input controls are styled using a *theme* that you apply to your entire app.

For instance, to ensure that all devices that can run the `Holo` theme will use the `Holo` theme for your app, declare the following in the `<application>` element of the `AndroidManifest.xml` file:

```
android:theme="@android:style/Theme.Holo"
```

After adding the declaration above, the app will be displayed using the theme.

Apps designed for Android 4.0 and higher can also use the `DeviceDefault` public theme family. `DeviceDefault` themes are aliases for the device's native look and feel. The `DeviceDefault` theme family and widget style family offer ways for developers to target the device's native theme with all customizations intact.
For Android apps running on 4.0 and newer, you have the following options:

- Use a theme, such as one of the `Holo` themes, so that your app has the exact same look across all Android-powered devices running 4.0 or newer. In this case, the app's look does not change when running on a device with a different default skin or custom skin.
- Use one of the `DeviceDefault` themes so that your app takes on the look of the device's default skin.
- Don't use a theme, but you may have unpredictable results on some devices.

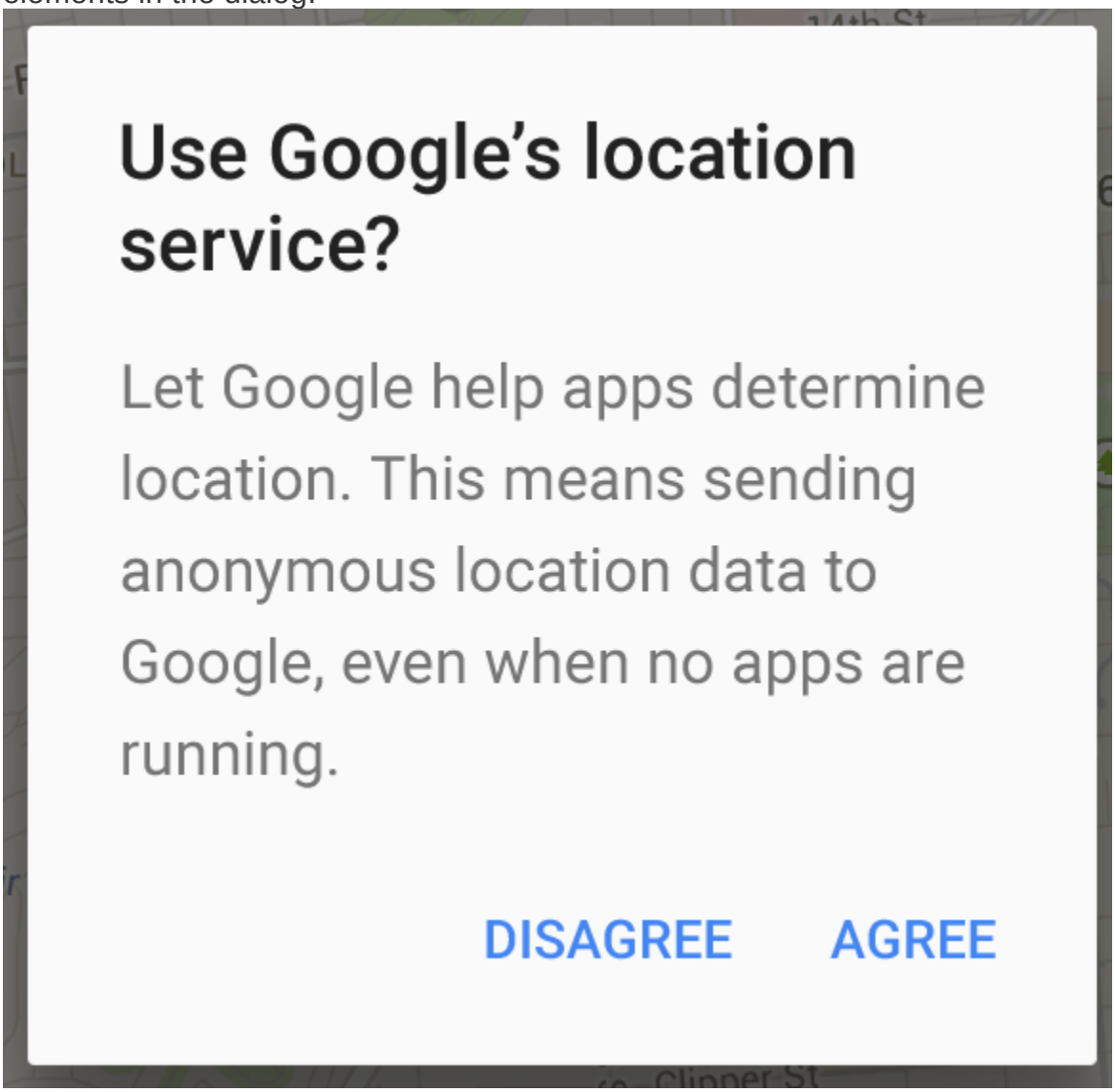
Recommended reading:

- If you're not familiar with Android's style and theme system, you should read Styles and themes.
- For information about using the Holo theme while supporting older devices, see the blog post Holo Everywhere.
- For a guide on styling and customizing buttons using XML, see Buttons in the Android developer documentation.
- For a comprehensive guide to designing buttons, see Buttons in the Material Design specification.

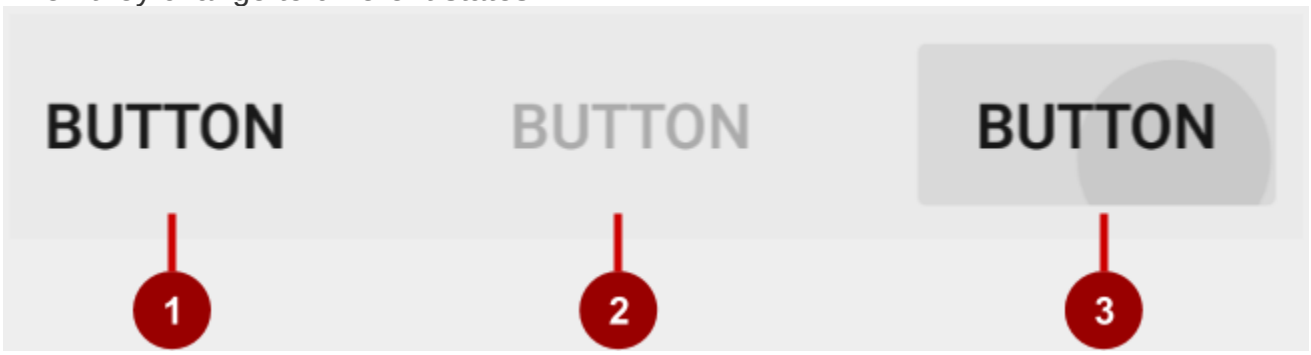
Designing flat buttons

A *flat button*, also known as a *text button* or *borderless button*, is a text-only button that looks flat and doesn't have a shadow. The major benefit of flat buttons is simplicity: a flat button doesn't distract the user from the main content as much as a raised button does. Flat buttons are useful for dialogs that require user interaction, as shown in the figure below. In this case, you want the button to use the same font and style as the surrounding text to keep the look and feel consistent across all the

elements in the dialog.



Flat buttons have no borders or background, but they do change their appearance when they change to different states.



In the figure above:

1. Normal state: In its normal state, the button looks just like ordinary text.
2. Disabled state: When the text is dimmed out, the button is not active in the app's context.
3. Pressed state: A background shadow indicates that the button is being tapped or clicked. When you attach a callback (such as the `android:onClick` attribute) to the button, the callback is called when the button is in this state.

Note: If you use a flat button within a layout, be sure to use padding to set it off from the surrounding text, so that the user can easily see it.

To create a flat button, use the `Button` class. Add a `Button` to your XML layout, and apply `"?android:attr/borderlessButtonStyle"` as the `style` attribute:

```
<Button
    android:id="@+id/button_send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage"
    style="?android:attr/borderlessButtonStyle" />
```

Responding to button-click events

An *event listener* is an interface in the `View` class that contains a single callback method. The Android system calls the method when the user triggers the `View` to which the listener is registered.
To respond to a user tapping or clicking a button, use the event listener called `OnClickListener`, which contains one method, `onClick()`. To provide functionality when the user clicks, you implement this `onClick()` method.
For more about event listeners and other UI events, see Input events overview in the Android developer documentation.

Adding onClick() to the layout element

A quick way to set up an `OnClickListener` for a clickable element in your `Activity` code and assign a callback method is to add the `android:onClick` attribute to the element in the XML layout.
For example, a `Button` in the layout would include the `android:onClick` attribute:

```
<Button
    android:id="@+id/button_send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage" />
```

When a user clicks the `Button`, the Android framework calls the `sendMessage()` method in the `Activity`:

```
public void sendMessage(View view) {
    // Do something in response to button click
}
```

The callback method for the `android:onClick` attribute must be `public`, return `void`, and define a `View` as its only parameter (this is the `View` that was tapped). Use the method to perform a task or call other methods as a response to the `Button` tap.

Using the button-listener design pattern

You can also handle the click event in your Java code using the button-listener design pattern, shown in the figure below. For more information on the "listener" design pattern, see Creating Custom Listeners.

Use the event listener `View.OnClickListener`, which is an interface in the `View` class that contains a single callback method, `onClick()`. The method is called by the Android framework when the view is triggered by user interaction.
The event listener must already be registered to the `View` in order to be called for the event. Follow these steps to register the listener and use it (refer to the figure below the steps):

1. Use the `findViewById()` method of the `View` class to find the `Button` in the XML layout file:

```
Button button = findViewById(R.id.button_send);
```

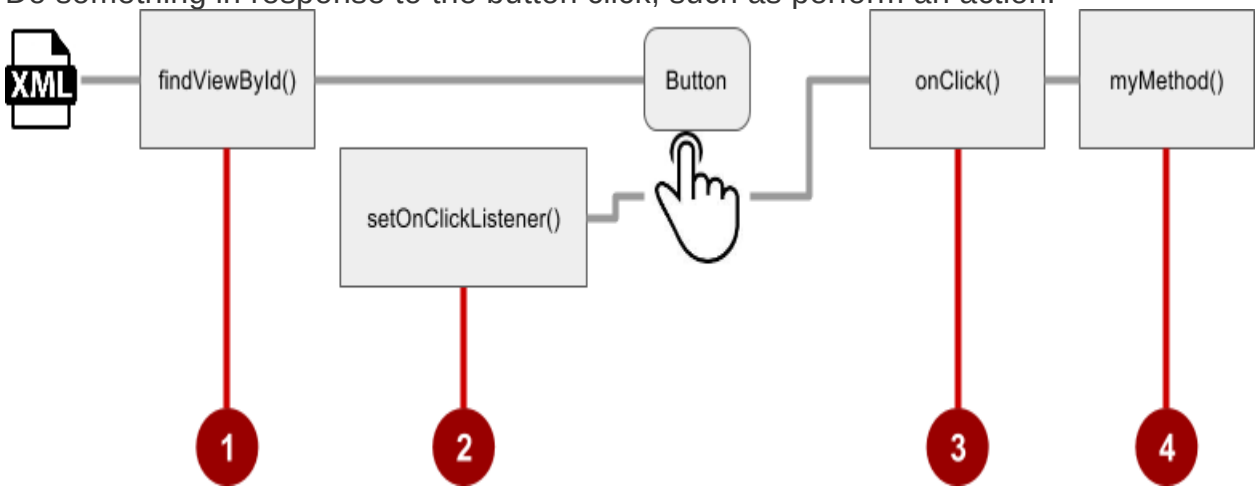
3. Get a new `View.OnClickListener` and register it to the `Button` by calling the `setOnClickListener()` method. The argument to `setOnClickListener()` takes an object that implements the `View.OnClickListener` interface, which has one method: `onClick()`.

```
button.setOnClickListener(new View.OnClickListener() {
    // ... The onClick method goes here.
}
```

7. Override the `onClick()` method:

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // Do something in response to button click
    }
}
```

14. Do something in response to the button click, such as perform an action.



Using the event listener interface for other events

Other events can occur with UI elements, and you can use the callback methods already defined in the event listener interfaces to handle them. The methods are called by the Android framework when the view—to which the listener has been registered—is triggered by user interaction. You therefore must set the appropriate listener to use the method. The following are some of the listeners available in the Android framework and the callback methods associated with each one:

- `onClick()` from `View.OnClickListener`: Handles a click event in which the user touches and then releases an area of the device display occupied by a `View`. The `onClick()` callback has no return value.
- `onLongClick()` from `View.OnLongClickListener`: Handles an event in which the user maintains touch on a `View` for an extended period. This method returns a `boolean` to indicate whether you have consumed the event, and the event should not be carried further. That is, return `true` to indicate that you have handled the event and the event should stop here. Return `false` if you have not handled the event, or if the event should continue to any other listeners.
- `onTouch()` from `View.OnTouchListener`: Handles any form of touch contact with the screen including individual or multiple touches and gesture motions, including a press, a release, or any movement gesture on the screen (within the bounds of the UI element). A `MotionEvent` is passed as an argument, which includes directional information, and it returns a `boolean` to indicate whether your listener consumes this event.
- `onFocusChange()` from `View.OnFocusChangeListener`: Handles when focus moves away from the current `View` as the result of interaction with a trackball or navigation key.
- `onKey()` from `View.OnKeyListener`: Handles when a key on a hardware device is pressed while a `View` has focus.

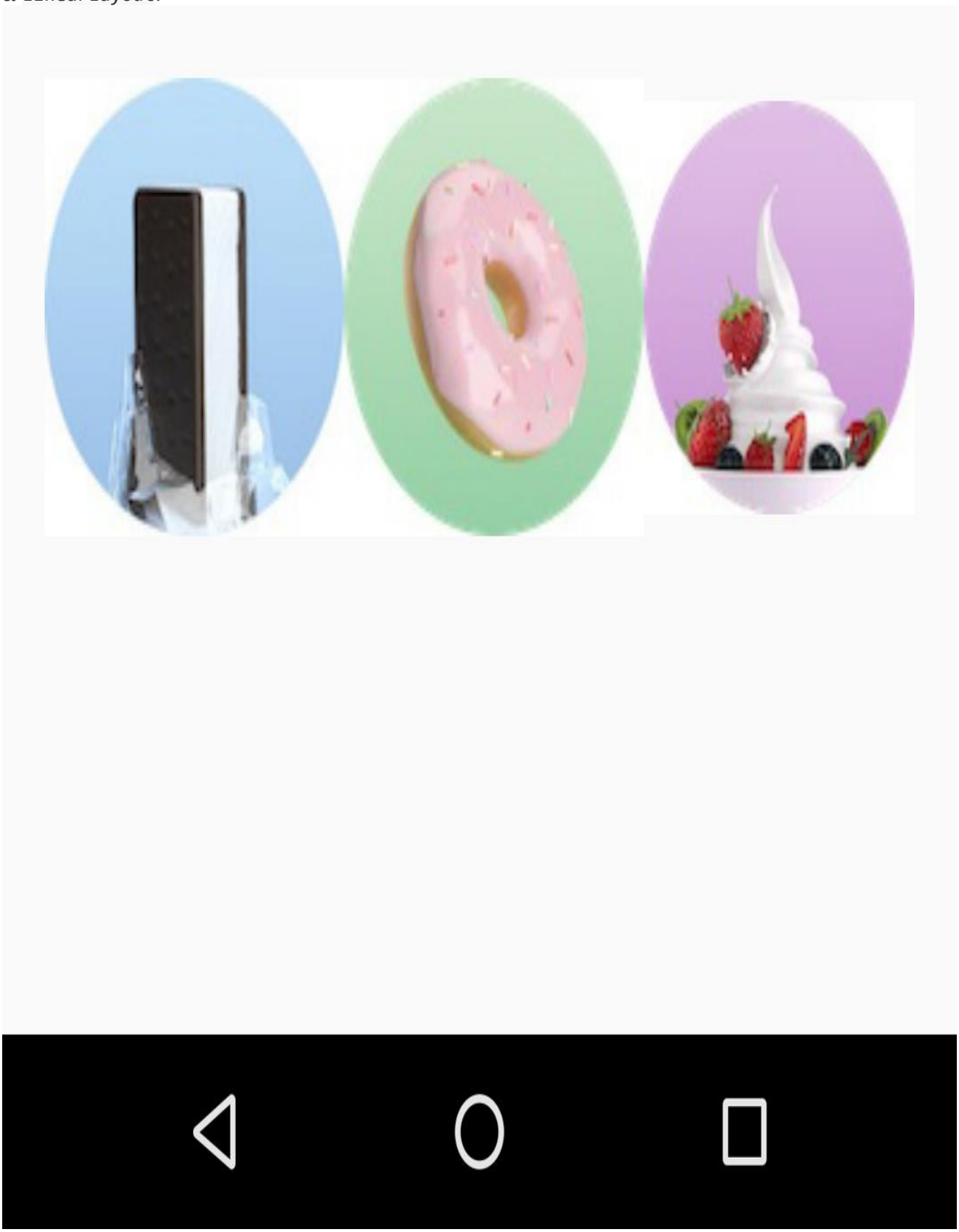
Using clickable images

You can turn any `View`, such as an `ImageView`, into a button by adding the `android:onClick` attribute in the XML layout. The image for the `ImageView` must already be stored in the **drawable** folder of your project.
Note: To bring images into your Android Studio project, create or save the image in PNG or JPEG format, and copy the image file into the **app > src > main > res > drawable** folder of your project. For more information about drawable resources, see Drawable resources in the Android developer documentation.

For example, the following images in the `drawable` folder (icecream_circle.jpg, donut_circle.jpg, and froyo_circle.jpg) are defined for `ImageView` elements arranged in

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

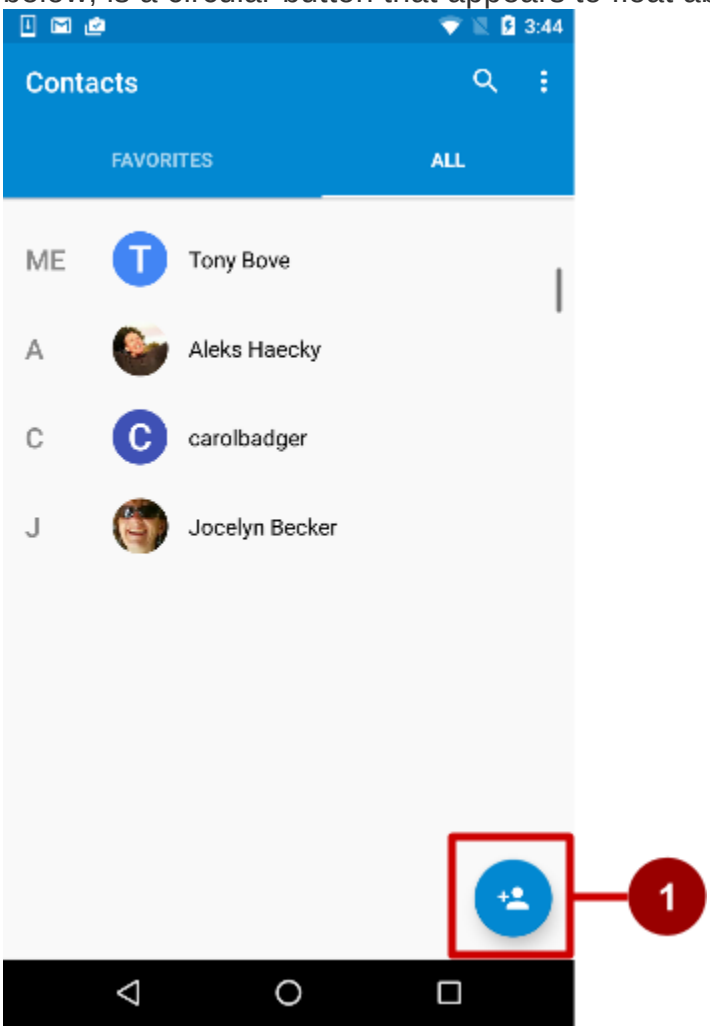
a `LinearLayout`:



```
<LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal"
        android:layout_marginTop="260dp">
        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/icecream_circle"
            android:onClick="orderIcecream"/>
        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/donut_circle"
            android:onClick="orderDonut"/>
        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/froyo_circle"
            android:onClick="orderFroyo"/>
</LinearLayout>
```

Using a floating action button

A floating action button (`FloatingActionButton`), shown below as #1 in the figure below, is a circular button that appears to float above the layout.



You should use a floating action button only to represent the primary action for a screen. For example, the primary action for the Contacts app main screen is adding a contact, as shown in the figure above. A floating action button is the right choice if your app requires an action to be persistent and readily available on a screen. Only one floating action button is recommended per screen.

The floating action button uses the same type of icons that you would use for a button with an icon, or for actions in the app bar at the top of the screen. You can add an icon as described previously in "Choosing an icon for the button".

If you start your project or `Activity` with the Basic Activity template, Android Studio adds a floating action button to the layout file for the `Activity`. To create a floating action button yourself, use the `FloatingActionButton` class, which extends the `ImageButton` class. You can add a floating action button to your XML layout as follows:

```
<android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        android:src="@drawable/ic_fab_chat_button_white" />
```

- Floating action buttons, by default, are 56 x 56 dp in size. It is best to use the default size unless you need the smaller version to create visual continuity with other screen elements.
- You can set the *mini* size (30 x 40 dp) with the `app:fabSize` attribute: `app:fabSize="mini"`
- To set it back to the default size (56 x 56 dp): `app:fabSize="normal"`

For more design instructions involving floating action buttons, see Components–Buttons: Floating Action Button in the Material Design Spec.

Recognizing gestures

A *touch gesture* occurs when a user places one or more fingers on the touchscreen, and your app interprets that pattern of touches as a particular gesture, such as a tap, touch & hold, double-tap, fling, or scroll.

Android provides a variety of classes and methods to help you create and detect gestures. Although your app should not depend on touch gestures for basic behaviors (because the gestures may not be available to all users in all contexts), adding touch-based interaction to your app can greatly increase its usefulness and appeal.

To provide users with a consistent, intuitive experience, your app should follow the accepted Android conventions for touch gestures. The Gestures design guide shows you how to design common gestures in Android apps. For more code samples and details, see Using touch gestures in the Android developer documentation.

Detecting common gestures

If your app uses common gestures such as double tap, long press, fling, and so on, you can take advantage of the `GestureDetector` class for detecting common gestures. Use `GestureDetectorCompat`, which is provided as a compatibility implementation of the framework's `GestureDetector` class which guarantees the newer focal point scrolling behavior from Jellybean MR1 on all platform versions. This class should be used only with motion events reported for touch devices—don't use it for trackball or other hardware events.
`GestureDetectorCompat` lets you detect common gestures without processing the individual touch events yourself. It detects various gestures and events using `MotionEvent` objects, which report movements by a finger (or mouse, pen, or trackball).

The following snippets show how you would use `GestureDetectorCompat` and the `GestureDetector.SimpleOnGestureListener` class.

Creating an instance of GestureDetectorCompat

To use `GestureDetectorCompat`, create an instance (`mDetector` in the snippet below) of the `GestureDetectorCompat` class, using the `onCreate()` method in the `Activity` (such as `MainActivity`):

```
public class MainActivity extends Activity {
    private GestureDetectorCompat mDetector;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mDetector = new GestureDetectorCompat(this, new
                MyGestureListener());
        // ... Rest of onCreate code.
    }
    // ... Rest of code.
}
```

When you instantiate a `GestureDetectorCompat` object, one of the parameters it takes is a class that you must create, which is `MyGestureListener` in the snippet above. The class you create should do one of the following:

- Implement the `GestureDetector.OnGestureListener` interface to detect all standard gestures, or
- Extend the `GestureDetector.SimpleOnGestureListener` class, which you can use to process only a few gestures by overriding the methods you need.

`SimpleOnGestureListener` provides methods such as `onDown()`, `onLongPress()`, `onFling()`, `onScroll()`, and `onSingleTapUp()`.

Extending GestureDetector.SimpleOnGestureListener

Create the class `MyGestureListener` as a separate `Activity` (**MyGestureListener**) to extend `GestureDetector.SimpleOnGestureListener`. Override the `onFling()` and `onDown()` methods to show log statements about the event:

```
class MyGestureListener
                  extends GestureDetector.SimpleOnGestureListener {
   private static final String DEBUG_TAG = "Gestures";

   @Override
   public boolean onDown(MotionEvent event) {
      Log.d(DEBUG_TAG,"onDown: " + event.toString());
      return true;
   }

   @Override
   public boolean onFling(MotionEvent event1, MotionEvent event2,
                  float velocityX, float velocityY) {
       Log.d(DEBUG_TAG, "onFling: " +
            event1.toString()+event2.toString());
       return true;
   }
}
```

Intercepting touch events

To intercept touch events, override the `onTouchEvent()` callback of the `GestureDetectorCompat` class:

```
@Override
public boolean onTouchEvent(MotionEvent event){
      this.mDetector.onTouchEvent(event);
      return super.onTouchEvent(event);
}
```

Detecting all gestures

To detect all types of gestures, you need to perform two essential steps:

1. Gather data about touch events.
2. Interpret the data to see if it meets the criteria for any of the gestures your app supports.

The gesture starts when the user first touches the screen, continues as the system tracks the position of the user's finger or fingers, and ends when the system captures the event of the user's fingers leaving the screen. Throughout this interaction, an object of the `MotionEvent` class is delivered to `onTouchEvent()`, providing the details. Your app can use the data provided by the `MotionEvent` to determine if a gesture it cares about happened.

For example, when the user first touches the screen, the `onTouchEvent()` method is triggered on the `View` that was touched, and a `MotionEvent` object reports movement by a finger (or mouse, pen, or trackball) in terms of:

- *An action code:* Specifies the state change that occurred, such as a finger tapping down or lifting up.
- *A set of axis values:* Describes the position in X and Y coordinates of the touch and information about the pressure, size and orientation of the contact area.

The individual fingers or other objects that generate movement traces are referred to as *pointers*. Some devices can report multiple movement traces at the same time. Multi-touch screens show one movement trace for each finger. Motion events contain information about all of the pointers that are currently active even if some of them have not moved since the last event was delivered. Based on the interpretation of the `MotionEvent` object, the `onTouchEvent()` method triggers the appropriate callback on the `GestureDetector.OnGestureListener` interface.
Each `MotionEvent` pointer has a unique id that is assigned when it first goes down (indicated by `ACTION_DOWN` or `ACTION_POINTER_DOWN`). A pointer id remains valid until the pointer eventually goes up (indicated by `ACTION_UP` or `ACTION_POINTER_UP`) or when the gesture is canceled (indicated by `ACTION_CANCEL`). The `MotionEvent` class provides methods to query the position and other properties of pointers, such as `getX(int)`, `getY(int)`, `getAxisValue(int)`, `getPointerId(int)`, and `getToolType(int)`. The interpretation of the contents of a `MotionEvent` varies significantly depending on the source class of the device. On touchscreens, the pointer coordinates specify absolute positions such as view X/Y coordinates. Each complete gesture is represented by a sequence of motion events with actions that describe pointer state transitions and movements.
A gesture starts with a motion event with `ACTION_DOWN` that provides the location of the first pointer down. As each additional pointer goes down or up, the framework generates a motion event with `ACTION_POINTER_DOWN` or `ACTION_POINTER_UP` accordingly. Pointer movements are described by motion events with `ACTION_MOVE`. A gesture ends when the final pointer goes up as represented by a motion event with `ACTION_UP`, or when the gesture is canceled with `ACTION_CANCEL`.
To intercept touch events in an `Activity` or `View`, override the `onTouchEvent()` callback as shown in the snippet below. You can use the `getActionMasked()` method of the `MotionEventCompat` class to extract the action the user performed from the event parameter. (`MotionEventCompat` is a helper for accessing features in a `MotionEvent`, which was introduced after API level 4 in a backwards compatible fashion.)

This gives you the raw data you need to determine if a gesture you care about occurred:

```
public class MainActivity extends Activity {
// ...
// This example shows an Activity, but you would use the same approach if
// you were subclassing a View.
   @Override
   public boolean onTouchEvent(MotionEvent event){
       int action = MotionEventCompat.getActionMasked(event);
       switch(action) {
          case (MotionEvent.ACTION_DOWN) :
             Log.d(DEBUG_TAG,"Action was DOWN");
             return true;
          case (MotionEvent.ACTION_MOVE) :
             Log.d(DEBUG_TAG,"Action was MOVE");
             return true;
          case (MotionEvent.ACTION_UP) :
             Log.d(DEBUG_TAG,"Action was UP");
             return true;
          case (MotionEvent.ACTION_CANCEL) :
             Log.d(DEBUG_TAG,"Action was CANCEL");
             return true;
          case (MotionEvent.ACTION_OUTSIDE) :
             Log.d(DEBUG_TAG,"Movement occurred outside bounds " +
                    "of current screen element");
             return true;
          default :
             return super.onTouchEvent(event);
    }
}
```

You can then do your own processing on these events to determine if a gesture occurred.

Related practical

The related practical is 4.1: 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
- Input events overview
- Styles and themes
- Use touch gestures

Material Design spec:

- Components - Buttons
- Gestures design guide

Other:

- Codelabs: Using `ConstraintLayout` to design your views
- Android Developers Blog: Holo Everywhere
- Developer blog: Implementing Material Design in Your Android app

4.2: Input controls

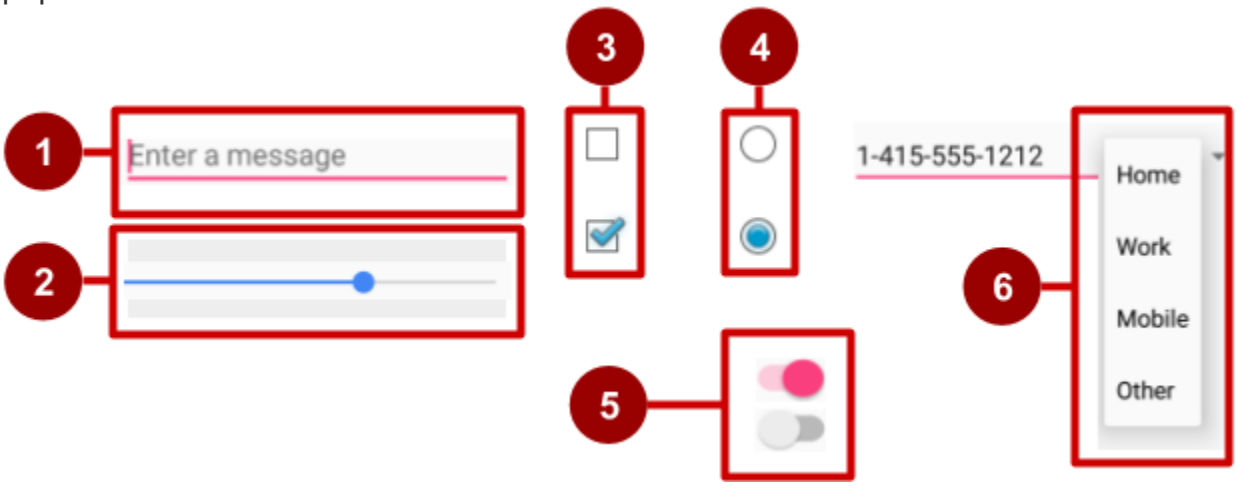
Contents:

- Input controls
- Checkboxes
- Radio buttons
- Spinner
- Toggle buttons and switches
- Text input
- Related practical
- Learn more

Input controls

This chapter introduces the Android *input controls.* Input controls are interactive elements in your app's UI that accept data input. Users input data to apps by entering text or numbers into fields using the on-screen keyboard. Users also select options from checkboxes, radio buttons, and drop-down menus, and they change settings and turn on or turn off certain features.

Android provides a variety of input controls for your UI. The figure below shows some popular ones.



In the figure above:

1. `EditText` field (subclass of `TextView`) for entering text using a keyboard
2. `SeekBar` for sliding left or right to a setting
3. `CheckBox` elements for selecting one or more options
4. `RadioGroup` of `RadioButton` elements for selecting one option
5. `Switch` for turning on or turning off an option
6. `Spinner` drop-down menu for selecting one option

When your app needs to get data from the user, try to make the process as easy for the user as it can be. For example, anticipate the source of the data, minimize the number of user gestures such as taps and swipes, and pre-fill forms when possible.

The user expects input controls to work in your app the same way they work in other apps. For example, users expect a `Spinner` to show a drop-down menu, and they expect text-editing fields to show a keyboard when tapped. Don't violate established expectations, or you'll make it harder for your users to use your app.

Input controls for making choices

Android offers ready-made input controls for the user to select one or more choices:

- `Checkbox`: Select one or more choices from a set of choices by tapping or clicking checkboxes.
- `RadioGroup` of radio buttons: Select one choice from a set of choices by clicking one circular "radio" button. Radio buttons are useful if you are providing only two or three choices.
- `ToggleButton` and `Switch`: Turn an option on or off.
- `Spinner`: Select one choice from a set of choices in a drop-down menu. A `Spinner` is useful for three or more choices, and takes up little room in your layout.

Input controls and the View focus

If your app has several UI input elements, which element gets input from the user first? For example, if you have several `EditText` elements for the user to enter text, which element (that is, which `View`) receives the text? The `View` that "has the focus" receives user input.
Focus indicates which `View` is selected. The user can initiate focus by tapping on a `View`, for example a specific `EditText` element. You can define a *focus order* that defines how focus moves from one element to another when the user taps the **Return** key, **Tab** key, or arrow keys. You can also control focus programmatically by calling `requestFocus()` on any `View` that is focusable.
In addition to being focusable, input controls can be *clickable*. If a view's `clickable` attribute is set to `true`, then the view can react to click events. You can also make an element clickable programmatically.
What's the difference between *focusable* and *clickable*?

- A *focusable* view is allowed to gain focus from a touchscreen, external keyboard, or other input device.
- A *clickable* view is any view that reacts to being tapped or clicked.

Android-powered devices use many input methods, including directional pads (D-pads), trackballs, touchscreens, external keyboards, and more. Some devices, like tablets and smartphones, are navigated primarily by touch. Other device have no touchscreen. Because a user might navigate through your UI with an input device such as D-pad or a trackball, make sure you do the following:

- Make it visually clear which `View` has focus, so that the user knows where the input goes.
- Explicitly set the focus in your code to provide a path for users to navigate through the input elements using directional keys or a trackball.

Fortunately, in most cases you don't need to control focus yourself. Android provides "touch mode" for devices that respond to touch, such as smartphones and tablets. When the user begins interacting with the UI by touching it, only `View` elements with `isFocusableInTouchMode()` set to `true`, such as text input fields, are focusable. Other `View` elements that are touchable, such as `Button` elements, don't take focus when touched. If the user clicks a directional key or scrolls with a trackball, the device exits "touch mode" and finds a view to take focus.
Focus movement is based on a natural algorithm that finds the nearest neighbor in a given direction:

- When the user taps the screen, the topmost `View` under the tap is in focus, providing touch access for the child `View` elements of the topmost `View`.
- If you set an `EditText` view to a single line (such as the `textPersonName` value for the `android:inputType` attribute), the user can tap the right-arrow key on the on-screen keyboard to close the keyboard and shift focus to the next input control `View` based on what the Android system finds.

The system usually finds the nearest input control in the same direction the user was navigating (up, down, left, or right).

If there are multiple input controls that are nearby and in the same direction, the system scans from left to right, top to bottom.

- Focus can also shift to a different `View` if the user interacts with a directional control, such as a D-pad or trackball.

You can influence the way Android handles focus by arranging input controls such as `EditText` elements in a certain layout from left to right and top to bottom, so that focus shifts from one to the other in the sequence you want.

If the algorithm does not give you what you want, you can override it by adding the `nextFocusDown`, `nextFocusLeft`, `nextFocusRight`, and `nextFocusUp` XML attributes to your layout file:

1. Add one of these attributes to a `View` to decide where to go upon leaving the `View`—in other words, which `View` should be the *next* `View`.
2. Define the value of the attribute to be the `id` of the next `View`. For example:

```
<LinearLayout
    <!-- Other attributes... -->
    android:orientation="vertical" >

    <Button android:id="@+id/top"
          <!-- Other Button attributes... -->
          android:nextFocusUp="@+id/bottom" />

    <Button android:id="@+id/bottom"
          <!-- Other Button attributes... -->
          android:nextFocusDown="@+id/top"/>

</LinearLayout>
```

In a vertical `LinearLayout`, navigating up from the first `Button` would not ordinarily go anywhere, nor would navigating down from the second `Button`. But in the example above, the `top Button` has specified the `bottom Button` as the `nextFocusUp` (and vice versa), so the navigation focus will cycle from top-to-bottom and bottom-to-top.
To declare a `View` as focusable in your UI (when it is traditionally not), add the `android:focusable` XML attribute to the `View` in the layout, and set its value to `true`.
You can also declare a `View` as focusable while in "touch mode" by setting `android:focusableInTouchMode` set to `true`.
You can also explicitly set the focus or find out which `View` has focus by using the following methods:

- Call `onFocusChanged` to determine where focus came from.
- To find out which `View` currently has the focus, call `Activity.getCurrentFocus()`, or use `ViewGroup.getFocusedChild()` to return the focused child of a `View` (if any).
- To find the `View` in the hierarchy that currently has focus, use `findFocus()`.
- Use `requestFocus` to give focus to a specific `View`.
- To change whether a `View` can take focus, call `setFocusable`.
- To set a listener that is notified when the `View` gains or loses focus, use `setOnFocusChangeListener`.

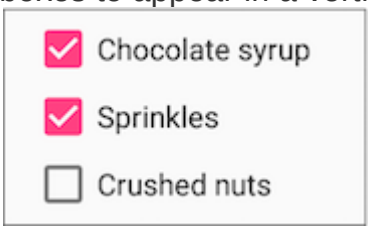
In this chapter, you learn more about focus with `EditText` elements.

Checkboxes

Use a set of checkboxes when you want the user to select *any number* of choices, including zero choices:

- Each checkbox is independent of the other boxes in the set, so selecting one box doesn't clear the other boxes. (If you want to limit the user's selection to one choice, use radio buttons.)
- A user can clear a checkbox that was already selected.

Users expect checkboxes to appear in a vertical list, like a to-do list, or side-by-side if the labels are short.



Each checkbox is a separate `CheckBox` element in your XML layout. To create multiple checkboxes in a vertical orientation, use a vertical `LinearLayout`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

        <CheckBox android:id="@+id/checkbox1_chocolate"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/chocolate_syrup" />
        <CheckBox android:id="@+id/checkbox2_sprinkles"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/sprinkles" />
        <CheckBox android:id="@+id/checkbox3_nuts"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/crushed_nuts" />

</LinearLayout>
```

Typically programs retrieve the state of each `CheckBox` when a user taps or clicks a **Submit** or **Done** `Button` in the same `Activity`, which uses the `android:onClick` attribute to call a method such as `onSubmit()`:

```
<Button
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="@string/submit"
   android:onClick="onSubmit"/>
```

The callback method—`onSubmit()` in the example above—must be `public`, return `void`, and define a `View` as a parameter (the view that was clicked). In this callback method you can determine whether a `CheckBox` is selected by using the `isChecked()` method (inherited from `CompoundButton`).
The `isChecked()` method returns `true` if there is a check mark in the box. For example, the following statement assigns `true` or `false` to `checked`, depending on whether the checkbox is checked:

```
boolean checked = ((CheckBox) view).isChecked();
```

The following code snippet shows the `onSubmit()` method checking to see which `CheckBox` is selected, using the resource `id` for the `CheckBox`:

```
public void onSubmit(View view) {
   StringBuffer toppings = new
                  StringBuffer().append(getString(R.string.toppings_label));
   if  (((CheckBox) findViewById(R.id.checkbox1_chocolate)).isChecked()) {
      toppings.append(getString(R.string.chocolate_syrup_text));
   }
   if  (((CheckBox) findViewById(R.id.checkbox2_sprinkles)).isChecked()) {
      toppings.append(getString(R.string.sprinkles_text));
   }
   if  (((CheckBox) findViewById(R.id.checkbox3_nuts)).isChecked()) {
      toppings.append(getString(R.string.crushed_nuts_text));
   }
// Code to display the result...
}
```

Tip: To respond quickly to a `CheckBox`—such as display a message (like an alert), or show a set of further options—you can use the `android:onClick` attribute in the XML layout for each `CheckBox` to declare the callback method for that `CheckBox`. The callback method must be defined within the `Activity` that hosts this layout.
For more information about checkboxes, see Checkboxes in the Android developer documentation.

Radio buttons

Use radio buttons when you have two or more options that are mutually exclusive. When the user selects one, the others are automatically deselected. (If you want to enable more than one selection from the set, use checkboxes.)

Users expect radio buttons to appear as a vertical list, or side-by-side if the labels are short.

Each radio button is an instance of the `RadioButton` class. Radio buttons are normally placed within a `RadioGroup` in a layout. When several `RadioButton` elements are inside a `RadioGroup`, selecting one `RadioButton` clears all the others.
Add `RadioButton` elements to your XML layout within a `RadioGroup`:

```
<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>
```

Use the `android:onClick` attribute for each `RadioButton` to declare the click handler, which must be defined within the `Activity` that hosts the layout. In the layout above, clicking any `RadioButton` calls the same `onRadioButtonClicked()` method in the `Activity`. You could also create separate click handlers in the `Activity` for each `RadioButton`.
The click handler method must be `public`, return `void`, and define a `View` as its only parameter (the view that was clicked). The following shows one click handler, `onRadioButtonClicked()`, for all the `RadioButton` elements in the `RadioGroup`.

It uses a `switch case` block to check the resource `id` for the `RadioButton` element to determine which one was checked:

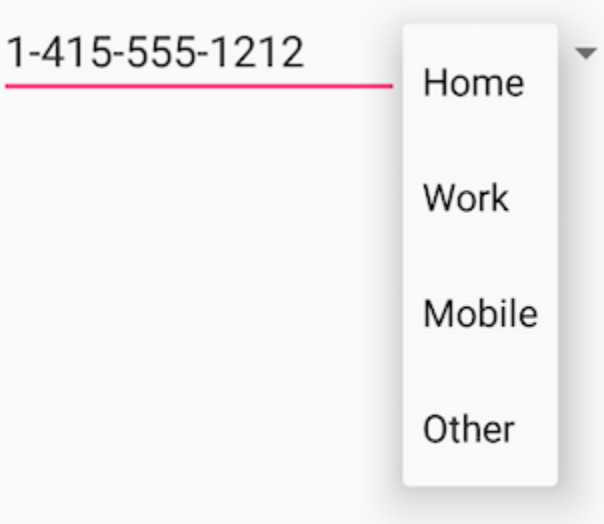
```
public void onRadioButtonClicked(View view) {
   // Check to see if a button has been clicked.
   boolean checked = ((RadioButton) view).isChecked();
   // Check which radio button was clicked.
   switch(view.getId()) {
      case R.id.sameday:
         if (checked)
            // Code for same day service ...
            break;
      case R.id.nextday:
         if (checked)
            // Code for next day delivery ...
            break;
      case R.id.pickup:
         if (checked)
            // Code for pick up ...
            break;
   }
}
```

Tip: To give users a chance to review their radio button selection before the app responds, you could implement a **Submit** or **Done** button as shown previously with checkboxes, and remove the `android:onClick` attributes from the radio buttons. Then add the `onRadioButtonClicked()` method to the `android:onClick` attribute for the **Submit** or **Done** button.
For more information about radio buttons, see Radio Buttons in the Android developer documentation.

Spinner

A `Spinner` provides a quick way for the user to select one value from a set. The user taps on the spinner to see a drop-down list with all available values.
A spinner works well when the user has more than three choices, because spinners scroll as needed, and a spinner doesn't take up much space in your layout. If you are providing only two or three choices and you have space in your layout, you might want to use radio buttons instead of a spinner.

Tip: For more information about spinners, see the Spinners guide.



If you have a long list of choices, a spinner might extend beyond your layout, forcing the user to scroll. A spinner scrolls automatically, with no extra code needed. However, making the user scroll through a long list (such as a list of countries) isn't recommended, because it can be hard for the user to select an item.

To create a spinner, use the `Spinner` class, which creates a `View` that displays individual spinner values as child `View` elements and lets the user pick one. Follow these steps:

1. Create a `Spinner` element in your XML layout, and specify its values using an array and an `ArrayAdapter`.
2. Create the `Spinner` and its adapter using the `SpinnerAdapter` class.
3. To define the selection callback for the `Spinner`, update the `Activity` that uses the `Spinner` to implement the `AdapterView.OnItemSelectedListener` interface.

Create the Spinner UI element

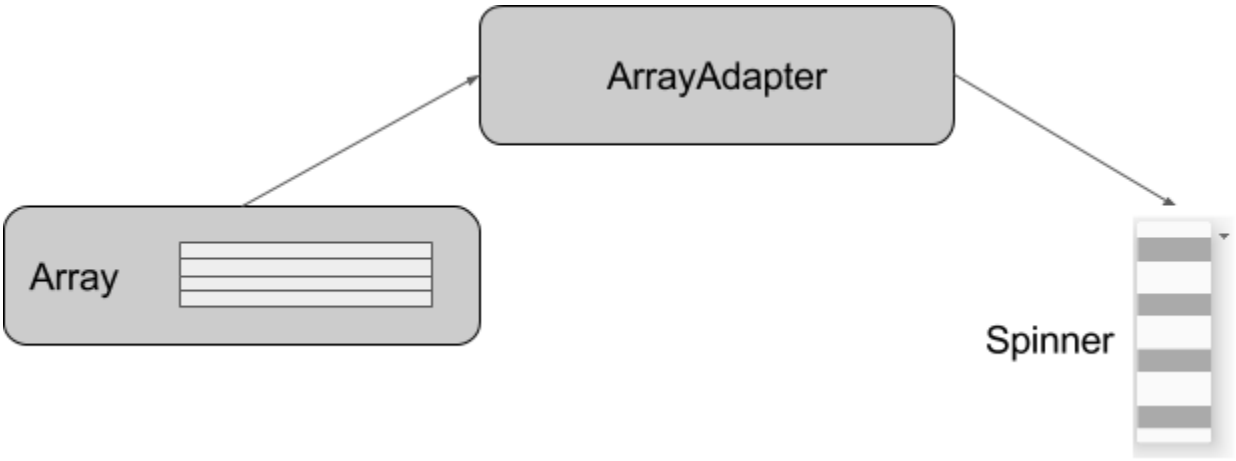
To create a spinner in your XML layout, add a `Spinner` element, which provides the drop-down list:

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

Specify values for the Spinner

Add an adapter that fills the `Spinner` list with values. An *adapter* is like a bridge, or intermediary, between two incompatible interfaces. For example, a memory card reader acts as an adapter between the memory card and a laptop. You plug the memory card into the card reader, and plug the card reader into the laptop, so that the laptop can read the memory card.

The adapter takes the data set you've specified (an array in this example), and makes a `View` for each item in the data set (a `View` within the `Spinner`), as shown in the figure below.



The `SpinnerAdapter` class, which implements the `Adapter` class, allows you to define two different views: one that shows the data values in the `Spinner` itself, and one that shows the data in the drop-down list when the `Spinner` is touched or clicked.
The values you provide for the `Spinner` can come from any source, but must be provided through a `SpinnerAdapter`, such as an `ArrayAdapter` if the values are easily stored in an array. The following shows a simple array called `labels_array` of predetermined values in the `strings.xml` file:

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

Tip: You can use a `CursorAdapter` if the values are provided from a source such as a stored file or a database. You learn more about stored data in another lesson.

Implement the OnItemSelectedListener interface in the Activity

To define the selection callback for the `Spinner`, update the `Activity` that uses the `Spinner` to implement the `AdapterView.OnItemSelectedListener` interface:

```
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`.
After typing **OnItemSelectedListener** in the statement above, wait a few seconds for a red light bulb to appear in the left margin. Click the bulb and choose **Implement methods**. The `onItemSelected()` and `onNothingSelected()` methods, which are required for `OnItemSelectedListener`, should already be highlighted, and the "Insert @Override" option should be checked. Click **OK**.

Android Studio automatically adds empty `onItemSelected()` and `onNothingSelected()` callback methods to the bottom of the `Activity`. 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.

Create the Spinner and its adapter

Create the `Spinner`, and set its listener to the `Activity` that implements the callback methods. The best place to do this is after the `Activity` layout is inflated in the `onCreate()` method. Follow these steps:

1. 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.
```

The code snippet above uses `findViewById()` to find the `Spinner` by its `id` (`label_spinner`). It then sets the `onItemSelectedListener` to whichever `Activity` implements the callbacks (`this`) using the `setOnItemSelectedListener()` method

11. 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.
```

As shown in the snippet above, you use the `createFromResource()` method to create the adapter. It takes as arguments the `Activity` (`this`) that implements the callbacks for processing the results of the `Spinner`, the array (`labels_array`), and the layout for each spinner item (`layout.simple_spinner_item`).
The `simple_spinner_item` layout used in this step, and the `simple_spinner_dropdown_item` layout used in the next step, are the default 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.

16. 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 ...
```

The snippet above uses `setAdapter()` to apply the adapter to the `Spinner`. You should use the `simple_spinner_dropdown_item` default layout, unless you want to define your own layout for the `Spinner` appearance.

Add code to respond to Spinner selections

When the user chooses an item from the spinner's drop-down list, here's what happens and how you retrieve the item:

1. The `Spinner` receives an on-item-selected event.
2. The event triggers the calling of the `onItemSelected()` callback method of the `AdapterView.OnItemSelectedListener` interface.
3. Retrieve the selected item in the `Spinner` using the `getItemAtPosition()` method of the `AdapterView` class:

```
public void onItemSelected(AdapterView<?> adapterView, View view, int
                i, long l) {
    String spinner_item = adapterView.getItemAtPosition(i).toString();
    // Do something with spinner_item string.
}
```

The arguments for `onItemSelected()` are as follows:

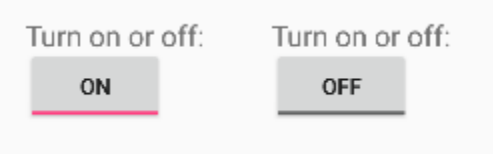
parent AdapterView	**The AdapterView where the selection happened**
`view View`	The `View` within the `AdapterView` that was clicked
`int pos`	The position of the `View` in the adapter
`long id`	The row id of the item that is selected

Implement/override the `onNothingSelected()` callback method of the `AdapterView.OnItemSelectedListener` interface to do something if nothing is selected.
For more information about using spinners, see Spinners in the Android developer documentation.

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

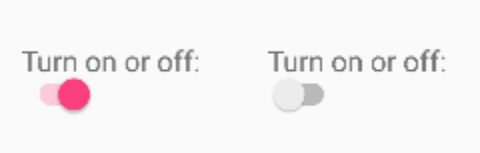
Toggle buttons and switches

A toggle input control lets the user change a setting between on and off. Android provides the `ToggleButton` class, which shows a raised button with "OFF" and "ON".



Examples of toggles include the on/off switches for Wi-Fi, Bluetooth, and other options in the Settings app.

Android also provides the `Switch` class, which is a short slider that looks like a rocker switch for on and off. Both are extensions of the `CompoundButton` class.



Using a toggle button

Create a toggle button by using a `ToggleButton` element in your XML layout:

```
<ToggleButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/my_toggle"
        android:text="
        android:onClick="onToggleClick"/>
```

Tip: The `android:text` attribute does not provide a text label for a toggle button—the toggle button always shows either "ON" or "OFF". To provide a text label next to or above the toggle button, use a separate `TextView`.
To respond to the toggle tap, declare an `android:onClick` callback method for the `ToggleButton`:

- The `onClick()` method must be defined in the `Activity` hosting the layout, and it must be `public` and return `void`.
- As its only parameter, the `onClick()` method must define a `View`—this will be the `View` that is clicked.

Use `CompoundButton.OnCheckedChangeListener()` to detect the state change of the toggle. Create a `CompoundButton.OnCheckedChangeListener` object and assign it to the toggle by calling `setOnCheckedChangeListener()`. For example, the `onToggleClick()` method checks whether the toggle is on or off, and displays a `Toast` message:

```
public void onToggleClick(View view) {
   ToggleButton toggle = findViewById(R.id.my_toggle);
   toggle.setOnCheckedChangeListener(new
                              CompoundButton.OnCheckedChangeListener() {
      public void onCheckedChanged(CompoundButton buttonView,
                              boolean isChecked) {
         StringBuffer onOff = new StringBuffer().append("On or off? ");
         if (isChecked) { // The toggle is enabled
            onOff.append("ON ");
         } else { // The toggle is disabled
            onOff.append("OFF ");
         }
         Toast.makeText(getApplicationContext(), onOff.toString(),
                              Toast.LENGTH_SHORT).show();
      }
   });
}
```

Tip: You can also programmatically change the state of a `ToggleButton` using the `setChecked(boolean)` method. Be aware, however, that the method specified by the `android:onClick()` attribute will not be executed in this case.

Using a switch

A switch is a separate instance of the `Switch` class, which extends the `CompoundButton` class just like `ToggleButton`. Create a toggle switch by using a `Switch` element in your XML layout:

```
<Switch
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/my_switch"
        android:text="@string/turn_on_or_off"
        android:onClick="onSwitchClick"/>
```

The `android:text` attribute defines a string that appears to the left of the `Switch`, as shown below:

To respond to the `Switch` tap, declare an `android:onClick` callback method for the `Switch`—the code is basically the same as for a `ToggleButton`. The method must be defined in the `Activity` hosting the layout, and it must be `public`, return `void`, and define a `View` as its only parameter (this will be the `View` that was clicked).

Use `CompoundButton.OnCheckedChangeListener()` to detect the state change of the `Switch`. Create a `CompoundButton.OnCheckedChangeListener` object and assign it to the `Switch` by calling `setOnCheckedChangeListener()`.

For example, the `onSwitchClick()` method checks whether the `Switch` is on or off, and displays a `Toast` message:

```
public void onSwitchClick(View view) {
   Switch aSwitch = findViewById(R.id.my_switch);
   aSwitch.setOnCheckedChangeListener(new
                           CompoundButton.OnCheckedChangeListener() {
      public void onCheckedChanged(CompoundButton buttonView,
                           boolean isChecked) {
         StringBuffer onOff = new StringBuffer().append("On or off? ");
         if (isChecked) { // The switch is enabled
            onOff.append("ON ");
         } else { // The switch is disabled
            onOff.append("OFF ");
         }
         Toast.makeText(getApplicationContext(), onOff.toString(),
                           Toast.LENGTH_SHORT).show();
      }
   });
}
```

Tip: You can also programmatically change the state of a `Switch` using the `setChecked(boolean)` method. Be aware, however, that the method specified by the `android:onClick()` attribute will not be executed in this case.
For more information about toggles, see Toggle Buttons in the Android developer documentation.

Text input

Use the `EditText` class to get user input that consists of textual characters, including numbers and symbols. `EditText` extends the `TextView` class, to make the `TextView` editable.

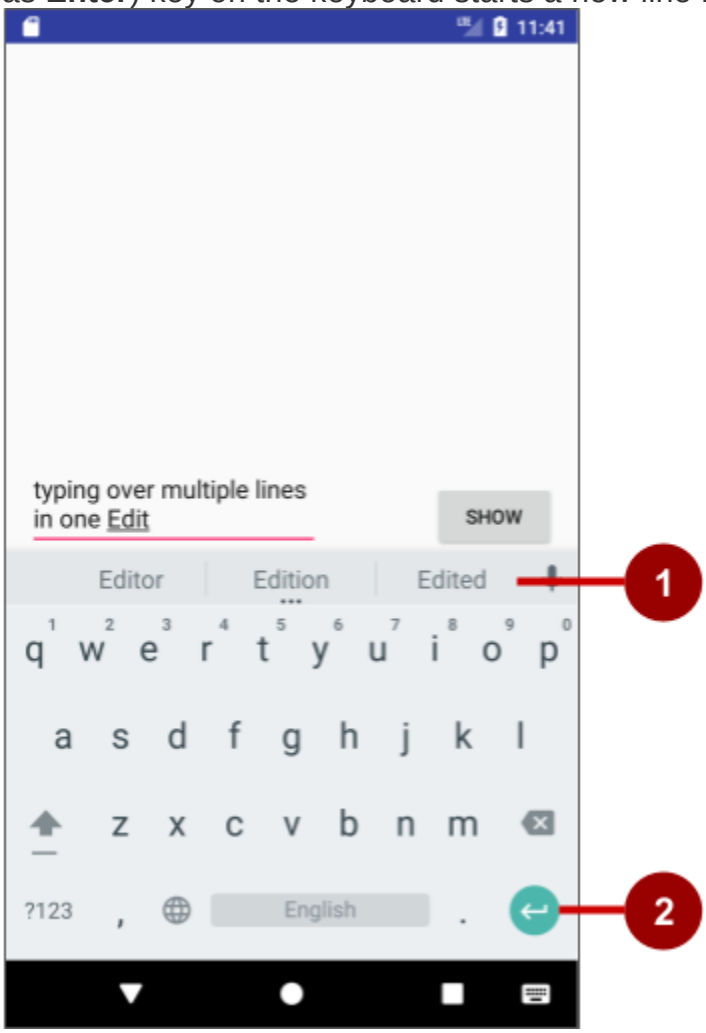
Customizing an EditText for user input

Create an `EditText` view by adding an `EditText` to your layout with the following XML:

```
<EditText
    android:id="@+id/edit_simple"
    android:layout_height="wrap_content"
    android:layout_width="match_parent">
    android:hint="@string/enter_text_here"
</EditText>
```

Multiple lines of text

By default, the `EditText` view allows multiple lines of input as shown in the figure below, and suggests full words the user can tap. Tapping the **Return** (also known as **Enter**) key on the keyboard starts a new line in the same `EditText`.



In the figure above:

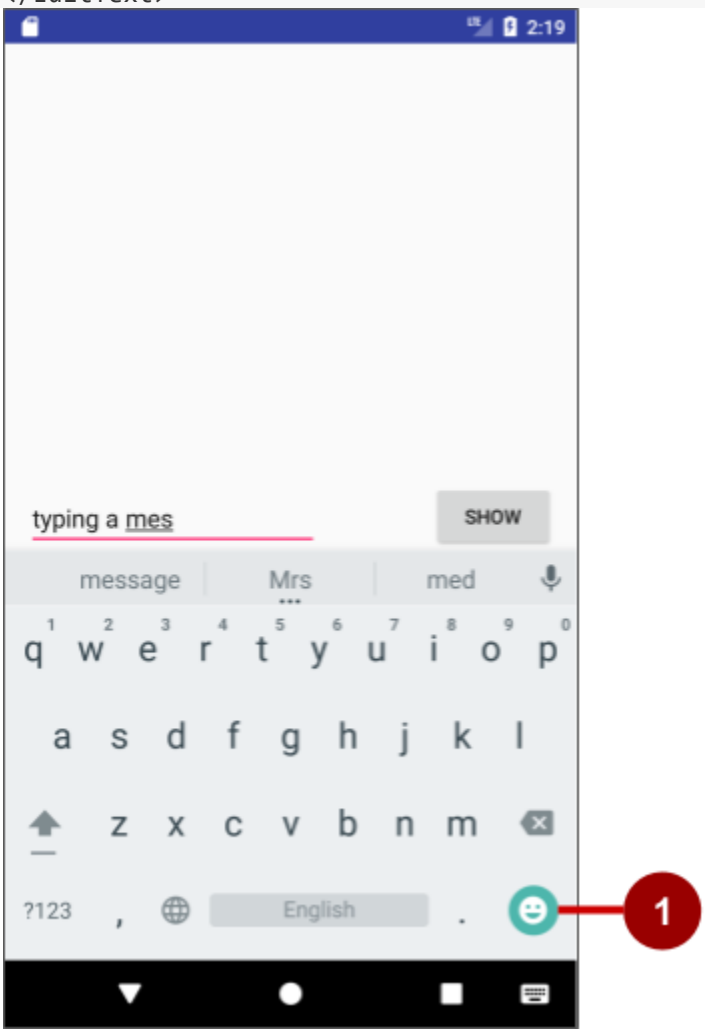
1. Suggestions to tap
2. Return (Enter) key

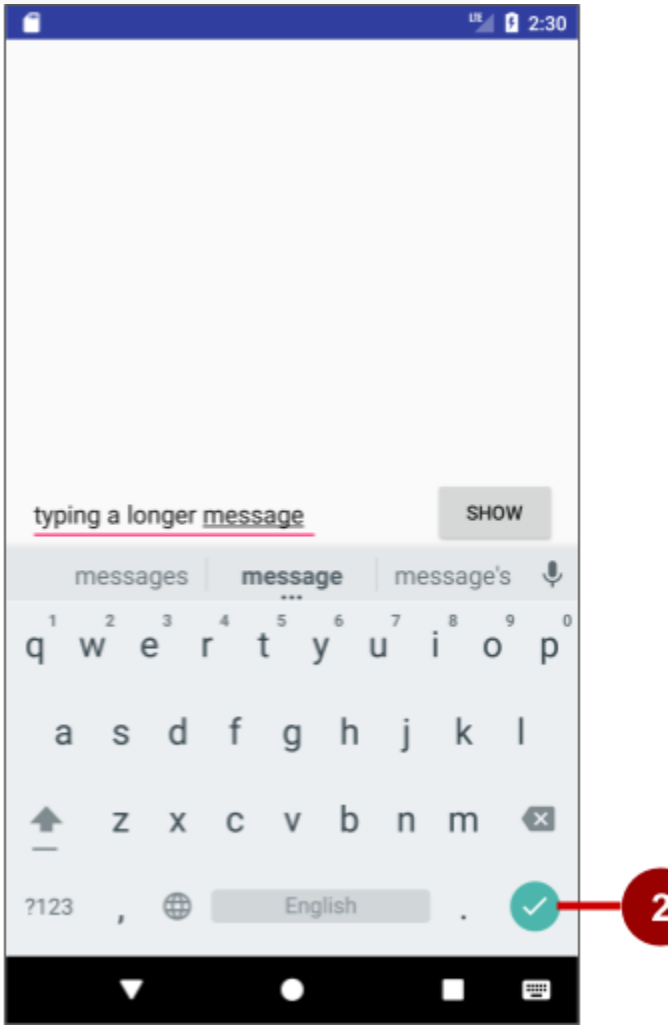
Message or single line of text

If you add the `android:inputType` attribute to the `EditText` you can customize it. For example, the following code uses `android:inputType="textShortMessage"` to show an `EditText` that offers a single line for typing a message, as shown on the left side of the figure below. An emoji (smiley face) takes the place of a **Return** key, changing the keyboard to emoji. To close the keyboard, the user taps the down-arrow key, which replaces the Back button in the bottom row of buttons.

```
<EditText
```

```
    android:id="@+id/edit_simple"
    android:layout_height="wrap_content"
    android:layout_width="match_parent">
    android:hint="@string/enter_text_here"
    android:inputType="textShortMessage"
</EditText>
```



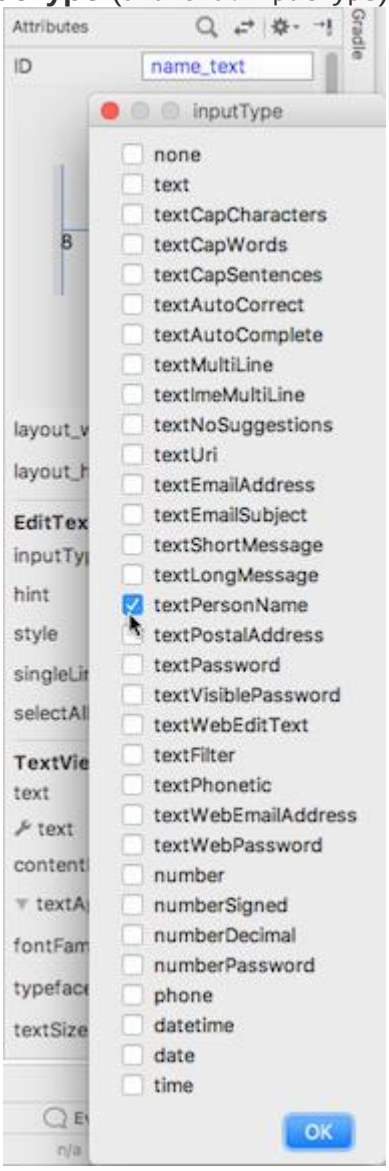


In the figure above:

1. Emoji key
2. Done key

Use `android:inputType="textLongMessage"` to show an `EditText` that offers a single line for typing a message, as shown on the right side of the figure above, with a **Done** key that closes the keyboard and advances the focus to the next `View`. This behavior is useful if you want the user to fill out a form consisting of `EditText` fields, so that the user can advance quickly to the next `EditText` field.
Tip: The Android Studio layout editor lets you drag a **Plain Text** element from the **Palette** to the layout. This element is, by default, an `EditText` with its `android:inputType` set to `textPersonName` for entering a person's name.

It provides a single line of text with suggestions and a **Done** key, just like `textLongMessage`. You can change the element to use a different **inputType** (`android:inputType`) value in the **Attributes** pane, as shown in the figure below.



Attributes for customizing an EditText view

The following are generally used attributes for customizing an `EditText`:

- `android:inputType="textCapCharacters"`: Set the text entry to all capital letters.
- `android:inputType="textCapSentences"`: Start each sentence with a capital letter.
- `android:inputType="textMultiLine"`: Set the text field to enable multiple lines. This value is useful for combining with other attributes. To combine values, concatenate them using the pipe (|) character.
- `android:inputType="textPassword"`: Turn each character the user enters into a dot to conceal an entered password.
- `android:inputType="number"`: Restrict text entry to numbers.
- `android:textColorHighlight="#7cff88"`: Set the background color of selected (highlighted) text.
- `android:hint="@string/my_hint"`: Set text to appear in the field that provides a hint for the user, such as "Enter a message".

For a list of `EditText` attributes, including inherited `TextView` attributes, see the "Summary" of the `EditText` class description.

Getting the user's input

To use the user's input in your app, set up layout and code by following these steps:

1. Add an `EditText` element to the XML layout for the `Activity`. Be sure to identify this element with an `android:id` so that you can refer to it by its `id`:

```
android:id="@+id/editText_main"
```

3. In the Java code for the same `Activity`, refer to the `EditText` by using the findViewById() method to find the `View` by its `id` (`editText_main`):

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

5. Use the `getText()` method to obtain the text as a character sequence (`CharSequence`). You can convert the character sequence into a string using the `toString()` method, which returns a string for the character sequence.

```
String showString = editText.getText().toString();
```

Use the `valueOf()` method of the `Integer` class to convert the string to an integer if the input is an integer.

Customizing the keyboard

The Android system shows an on-screen keyboard—known as a *soft* input method—when an `EditText` in the UI receives focus. To provide the best user experience, you can customize the keyboard to show, for example, a numeric keypad for entering phone number, or a keyboard for entering email addresses with the @ symbol conveniently located near the space key.
To customize the keyboard, use the android:inputType attribute for the `EditText` with the following values:

- `textEmailAddress`: Show an email keyboard with the @ symbol conveniently located next to the space key.
- `phone`: Show a numeric phone keypad.
- `date`: Show a numeric keypad with a slash for entering the date.
- `time`: Show a numeric keypad with a colon for entering the time.
- `datetime`: Show a numeric keypad with a slash *and* colon for entering the date and time.

Tip: You can use the pipe (`|`) character (Java bitwise OR) to combine attribute values for the `android:inputType` attribute:

```
android:inputType="textAutoCorrect|textCapSentences"
```

For details about the `android:inputType` attribute, see Specify the input method type in the developer documentation. For a complete list of constant values for `android:inputType`, see `android:inputType`.

Changing the "action" key in the keyboard

The "action" key for an `EditText` keyboard is the **Done** or **Return** key. You can change the "action" key to something else, such as a **Send** key, and change the action it performs.
To use a **Send** key and an action that dials a phone number, use the `android:inputType` attribute for the `EditText` element set to `phone`. Then use the `android:imeOptions` attribute with the `actionSend` value:

```
android:inputType="phone"
android:imeOptions="actionSend"
```

In the `onCreate()` method for the `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.
   });
```

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

```
@Override
public boolean onEditorAction(TextView textView, int actionId, KeyEvent keyEvent)
{
   boolean mHandled = false;
   if (actionId == EditorInfo.IME_ACTION_SEND) {
      dialNumber();
      mHandled = true;
   }
   return mHandled;
}
```

You would then need a `dialNumber()` method, which would use an implicit intent with `ACTION_DIAL` to pass the phone number to another app that can dial the number.

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

Related practical

The related practical is 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`

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

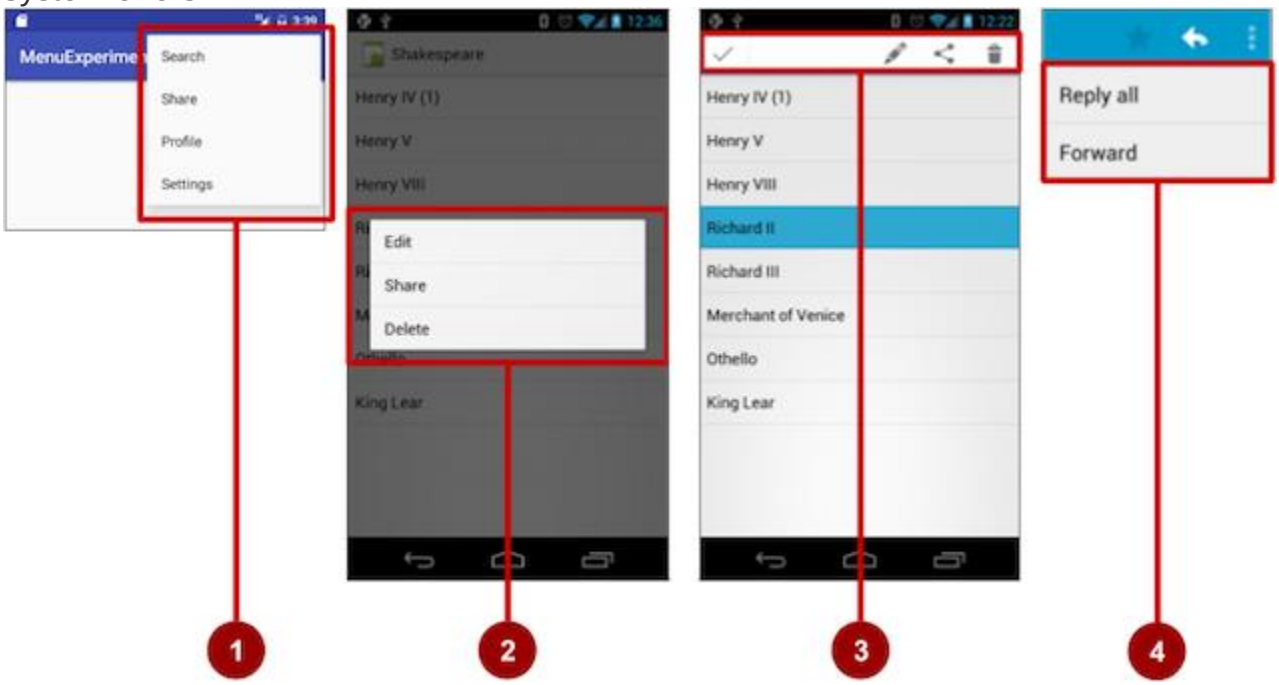
4.3: Menus and pickers

Contents:

- Types of menus
- The app bar and options menu
- Contextual menus
- Popup menu
- Dialogs and pickers
- Related practical
- Learn more

Types of menus

A *menu* is a set of options. The user can select from a menu to perform a function, for example searching for information, saving information, editing information, or navigating to a screen. The figure below shows the types of menus that the Android system offers.



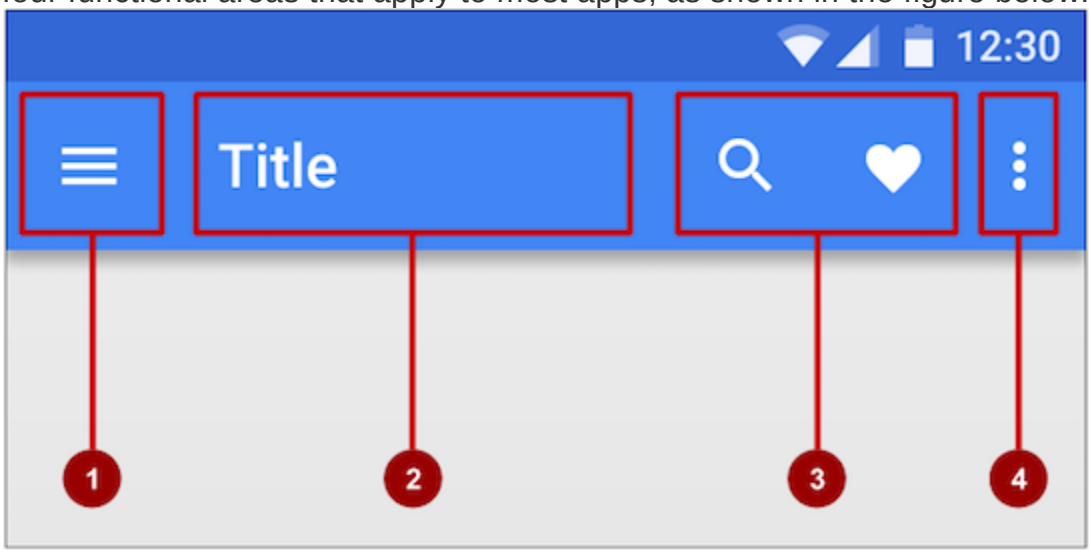
1. *Options menu:* Appears in the app bar and provides the primary options that affect use of the app itself. Examples of menu options: **Search** to perform a search, **Share** to share a link, and **Settings** to navigate to a Settings `Activity`.
2. *Contextual menu:* Appears as a floating list of choices when the user performs a long tap on an element on the screen. Examples of menu options: **Edit** to edit the element, **Delete** to delete it, and **Share** to share it over social media.
3. *Contextual action bar:* Appears at the top of the screen overlaying the app bar, with action items that affect the selected element or elements. Examples of menu options: **Edit**, **Share**, and **Delete** for one or more selected elements.
4. *Popup menu:* Appears anchored to a `View` such as an `ImageButton`, and provides an overflow of actions or the second part of a two-part command. Example of a popup menu: the Gmail app anchors a popup menu to the app bar for the message view with **Reply**, **Reply All**, and **Forward**.

The app bar and options menu

The *app bar* (also called the *action bar*) is a dedicated space at the top of each `Activity` screen. When you create an `Activity` from a template (such as Empty Template), an app bar is automatically included for the `Activity`.
The app bar by default shows the app title, or the name defined in `AndroidManifest.xml` by the `android:label` attribute for the `Activity`. The app bar may also include the *Up* button for navigating up to the parent activity. Up navigation is described in the chapter on using the app bar for navigation.
The *options menu* in the app bar usually provides navigation to other screens in the app, or options that affect using the app itself. (The options menu should *not* include options that act on an element on the screen. For that you use a *contextual menu*, described later in this chapter.)

For example, your options menu might let the user navigate to another activity to place an order. Or your options menu might let the user change settings or profile information, or do other actions that have a global impact on the app.

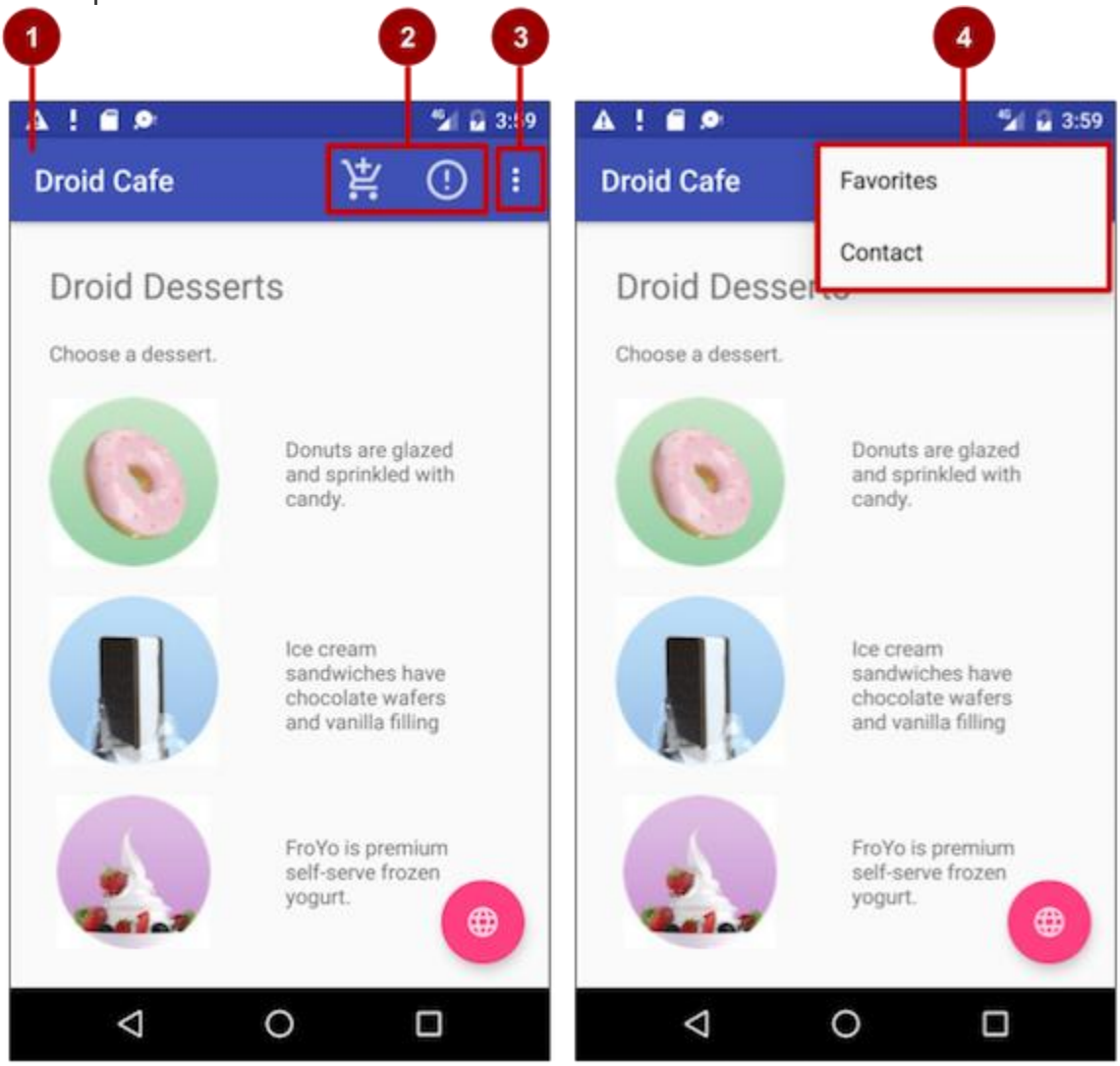
The options menu appears in the right corner of the app bar. The app bar is split into four functional areas that apply to most apps, as shown in the figure below.



In the figure above:

1. *Navigation button or Up button:* Use a navigation button in this space to open a navigation drawer, or use an *Up* button for navigating up through your app's screen hierarchy to the parent activity. Both are described in the next chapter.
2. *Title:* The title in the app bar is the app title, or the name defined in AndroidManifest.xml by the `android:label` attribute for the activity.
3. *Action icons for the options menu:* Each action icon appears in the app bar and represents one of the options menu's most frequently used items. Less frequently used options menu items appear in the overflow options menu.
4. *Overflow options menu:* The overflow icon opens a popup with option menu items that are not shown as icons in the app bar.

Frequently used options menu items should appear as icons in the app bar. The overflow options menu shows the rest of the menu:



In the above figure:

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

Adding the app bar

Each activity that uses the default theme also has an `ActionBar` as its app bar. Some themes also set up an `ActionBar` as an app bar by default. When you start an app from a template such as Empty Activity, an `ActionBar` appears as the app bar. Features were added to the native `ActionBar` over time, so the behavior of the native `ActionBar` depends on the version of Android that the device is running. For this reason, if you add an options menu, use the v7 appcompat support library's `Toolbar` as an app bar:

- `Toolbar` makes it easy to set up an app bar that works on a wide range of devices.
- `Toolbar` gives you room to customize your app bar later, as your app develops.
- `Toolbar` includes the most recent features, and it works for any device that can use the support library.

To use `Toolbar` as an activity's app bar (instead of using the default `ActionBar`), you can start your project with the Basic Activity template. That template implements `Toolbar` for the activity, and it implements a rudimentary options menu with one item, **Settings**.
Tip: If you use the Basic Activity template, you can skip the rest of this section, because the template provides everything you need.

If you are not using the Basic Activity template, this section describes how to add `Toolbar` yourself. The following are the general steps:

1. Add the support libraries `appcompat` and `design`.
2. Use a `NoActionBar` theme and styles for the app bar and background.
3. Add an `AppBarLayout` and a `Toolbar` to the layout.
4. Add code to the `Activity` to set up the app bar.

Adding the support libraries

If you start an app project using the Basic Activity template, the template adds the following support libraries for you, so you can skip this step.

If you are *not* using the Basic Activity template, add two things to your project: the appcompat support library for the `Toolbar` class, and the design library for the `NoActionBar` themes:

1. Choose **Tools > Android > SDK Manager** to check whether the Android Support Repository is installed. If the repository is not installed, install it.
2. Open the `build.gradle` file for your app, and add the support library feature project identifiers to the `dependencies` section. For example, to include `support:appcompat` and `support:design`, add:

```
3.  compile 'com.android.support:appcompat-v7:26.1.0'
4.  compile 'com.android.support:design:26.1.0'
```

Note: If necessary, update the version numbers for dependencies. If the version number you specified is lower than the currently available library version number, Android Studio warns you. Update the version number to the one Android Studio tells you to use.

Using themes to design the app bar

If you start an app project using the Basic Activity template, the template adds the theme to replace the `ActionBar` with a `Toolbar`, so you can skip this step.
If you are *not* using the Basic Activity template, you can use the `Toolbar` class for the app bar by turning off the default `ActionBar` using a `NoActionBar` theme for the activity.
Themes in Android are similar to styles, except that they are applied to an entire app or activity rather than to a specific view.
When you create a new project in Android Studio, an app theme is automatically generated for you. For example, if you start an app project with the Empty Activity or Basic Activity template, the `AppTheme` theme is provided in `styles.xml`. To see this file, expand the **res > values** folder in the **Project > Android** pane.
Tip: You learn more about themes in the chapter on drawables, styles, and themes.

You can modify the theme to provide a style for the app bar and app background. Follow these steps to make the app bar stand out against its background:

1. Open **styles.xml**. You should already have the following in the file within the `<resources>` section:

```
 <!-- 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>
```

   `AppTheme` "inherits"—takes on all the styles—from a parent theme called `Theme.AppCompat.Light.DarkActionBar`, which is a standard theme supplied with Android. However, you can override an inherited style with another style by adding the other style to **styles.xml**.
9. Add the `AppTheme.NoActionBar`, `AppTheme.AppBarOverlay`, and `AppTheme.PopupOverlay` styles under the `AppTheme` style, as shown below. These styles will override the style attributes with the same names in `AppTheme`, affecting the appearance of the app bar and the app's background:

```
 <style name="AppTheme.NoActionBar">
       <item name="windowActionBar">false</item>
       <item name="windowNoTitle">true</item>
 </style>

 <style name="AppTheme.AppBarOverlay"
            parent="ThemeOverlay.AppCompat.Dark.ActionBar" />

 <style name="AppTheme.PopupOverlay"
            parent="ThemeOverlay.AppCompat.Light" />
```

20. Open **AndroidManifest.xml** and add the `NoActionBar` theme in `appcompat` to the `<application>` element. Using this theme prevents the app from using the native `ActionBar` class to provide the app bar:

```
<activity
    <!-- android:name and android:label code goes here. -->
    android:theme="@style/AppTheme.NoActionBar">
    <!-- intent filter code would go here if needed. -->
</activity>
```

Adding AppBarLayout and a Toolbar to the layout

If you start an app project using the Basic Activity template, the template adds the `AppBarLayout` and `Toolbar` for you, so you can skip this step.
If you are *not* using the Basic Activity template, you can include the `Toolbar` in an `Activity` layout by adding an `AppBarLayout` and a `Toolbar` element. `AppBarLayout` is a vertical `LinearLayout` which implements many of the features of the material designs app bar concept, such as scrolling gestures. Keep in mind the following:

1. `AppBarLayout` must be a direct child within a `CoordinatorLayout` root view group, and `Toolbar` must be a direct child within `AppBarLayout`, placed at the top of the `Activity` layout. Shown below is a layout that uses this structure:

```
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.android.droidcafeinput.MainActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main" />

</android.support.design.widget.CoordinatorLayout>
```

2. `AppBarLayout` also requires a separate content layout sibling for the content that scrolls underneath the app bar. You can add this sibling as a view group (such as `RelativeLayout` or `LinearLayout`) in the same layout file, or in a separate layout file. The above XML snippet uses an `include layout` to include the content layout in `content_main.xml`.
3. Set the content sibling's view group to use the scrolling behavior `AppBarLayout.ScrollingViewBehavior`:

```
app:layout_behavior="@string/appbar_scrolling_view_behavior"
```

The layout behavior for the `RelativeLayout` is set to the string resource `@string/appbar_scrolling_view_behavior`. This string resource controls how the screen scrolls in relation to the app bar at the top. The resource represents the following string, which is defined in the `values.xml` file (which you should not modify):

```
android.support.design.widget.AppBarLayout$ScrollingViewBehavior
```

This behavior is defined by the `AppBarLayout.ScrollingViewBehavior` class. Any `View` or view group that can scroll vertically to support nested scrolling for `AppBarLayout` siblings should use this behavior.

Adding code to set up the app bar

If you start an app project using the Basic Activity template, the template adds the code needed to set up the app bar, so you can skip this step.

If you are *not* using the Basic Activity template, you can follow these steps to set up the app bar in the `Activity`:

1. Make sure that any `Activity` that you want to show an app bar extends `AppCompatActivity`:

```
 public class MainActivity extends AppCompatActivity {
     // ... Activity code
 }
```

5. In the `onCreate()` method for the `Activity`, call `setSupportActionBar()` with the `Toolbar`. The `setSupportActionBar()` method sets the `Toolbar` as the app bar for the `Activity`:

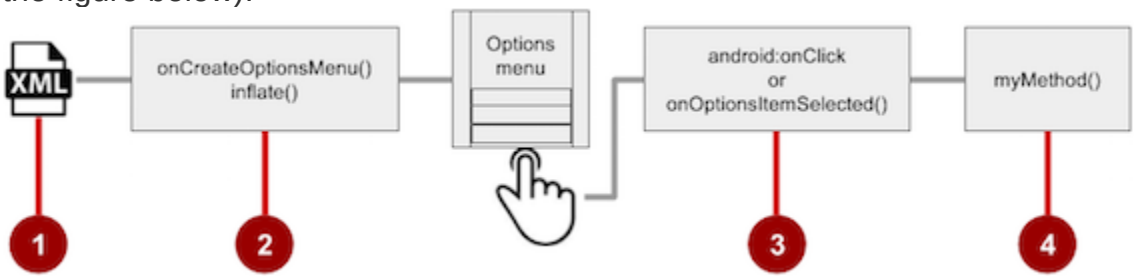
```
 @Override
 protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
 }
```

The `Activity` now shows the app bar. By default, the app bar contains just the name of the app.

Adding the options menu

Android provides a standard XML format to define options menu items. Instead of building the menu in your `Activity` code, you can define the menu and all its items in an XML menu resource. A menu resource defines an application menu (options menu, context menu, or popup menu) that can be inflated with `MenuInflater`, which loads the resource as a `Menu` object in your `Activity`.
If you start an app project using the Basic Activity template, the template adds the menu resource for you and inflates the options menu with `MenuInflater`, so you can skip this step and go right to "Defining how menu items appear".
If you are *not* using the Basic Activity template, use the resource-inflate design pattern, which makes it easy to create an options menu. Follow these steps (refer to

the figure below):



1. **XML menu resource**. Create an XML menu resource file for the menu items, and assign appearance and position attributes as described in the next section.
2. **Inflating the menu**. Override the `onCreateOptionsMenu()` method in your `Activity` to inflate the menu.
3. **Handling menu-item clicks**. Menu items are `View` elements, so you can use the `android:onClick` attribute for each menu item. However, the `onOptionsItemSelected()` method can handle all the menu-item clicks in one place and determine which menu item the user clicked, which makes your code easier to understand.
4. **Performing actions**. Create a method to perform an action for each options menu item.

Creating an XML resource for the menu

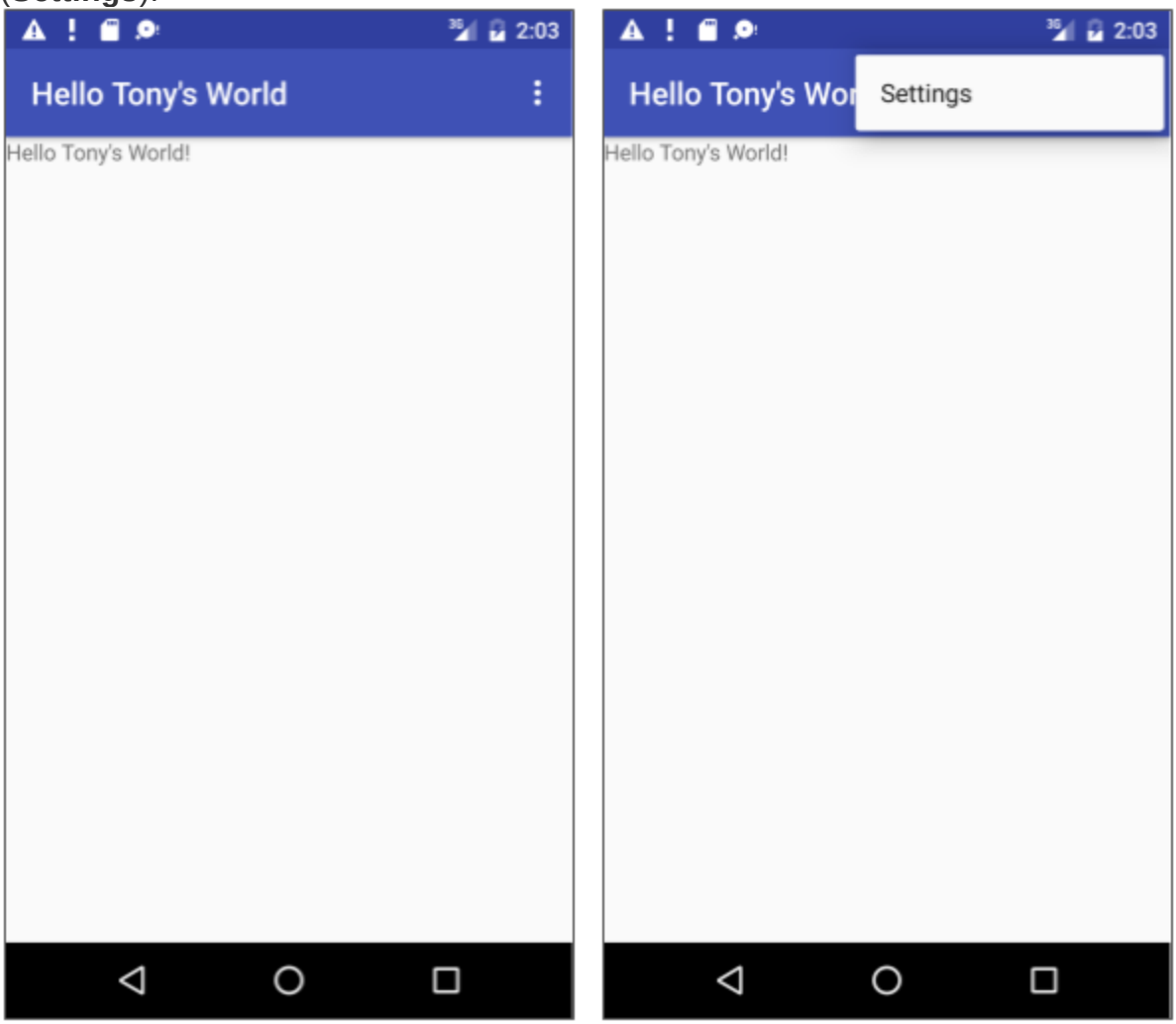
Follow these steps to add menu items to an XML menu resource:

1. Select the **res** folder in the **Project > Android** pane and choose **File > New > Android resource directory**.
2. Choose **menu** in the **Resource** type drop-down menu, and click **OK**.
3. Select the new **menu** folder, and choose **File > New > Menu resource file**.
4. Enter the name, such as **menu_main**, and click **OK**. The new `menu_main.xml` file now resides within the **menu** folder.
5. Open **menu_main.xml** and click the **Text** tab to show the XML code.
6. Add menu items using the `<item ... />` tag. In this example, the item is **Settings**:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.example.android.droidcafeinput.MainActivity">
    <item
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:title="Settings"
        app:showAsAction="never" />
```

After setting up and inflating the XML resource in the `Activity`, the overflow icon in the app bar, when clicked, would show the options menu with just one option

(**Settings**).



Defining how menu items appear

If you start an app project using the Basic Activity template, the template adds the options menu with one option: **Settings**.

To add more options menu items, add more `<item ... />` tags in the **menu_main.xml** file. For example, in the following snippet, two menu items are defined: `@string/settings` (**Settings**) and `@string/action_order` (**Order**):

```
<item
      android:id="@+id/action_settings"
      android:title="@string/settings" />
<item
      android:id="@+id/action_order"
      android:icon="@drawable/ic_order_white"
      android:title="@string/action_order"/>
```

Within each `<item ... />` tag, you add attributes that define how the menu item appears. For example, you can define the order of the item's appearance relative to other items, and whether the item can appear as an icon in the app bar. The following items are placed in the overflow menu:

- Any item that you set to *not* appear in the app bar.
- Any item that can't fit in the app bar, given the display orientation.

Whenever possible, show the most-used actions using icons in the app bar so that the user can tap these actions without having to first tap the overflow button.

Adding icons for menu items

To specify icons for actions, first add the icons as image assets to the **drawable** folder by following the steps below. (For a comlete description, see Create app icons with Image Asset Studio.)

1. Expand **res** in the Project > Android pane, and right-click (or Command-click) **drawable**.
2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.
3. Choose **Action Bar and Tab Items** in the drop-down menu.
4. Edit the name of the icon (for example, **ic_order_white** for the **Order** menu item).
5. Click the clip art image (the Android logo) to select a clip art image as the icon. A page of icons appears. Click the icon you want to use.
6. (Optional) 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.

Icon and appearance attributes

Use the following attributes to govern the menu item's appearance:

- `android:icon`: An image to use as the menu item icon. For example, the following menu item defines `ic_order_white` as its icon:

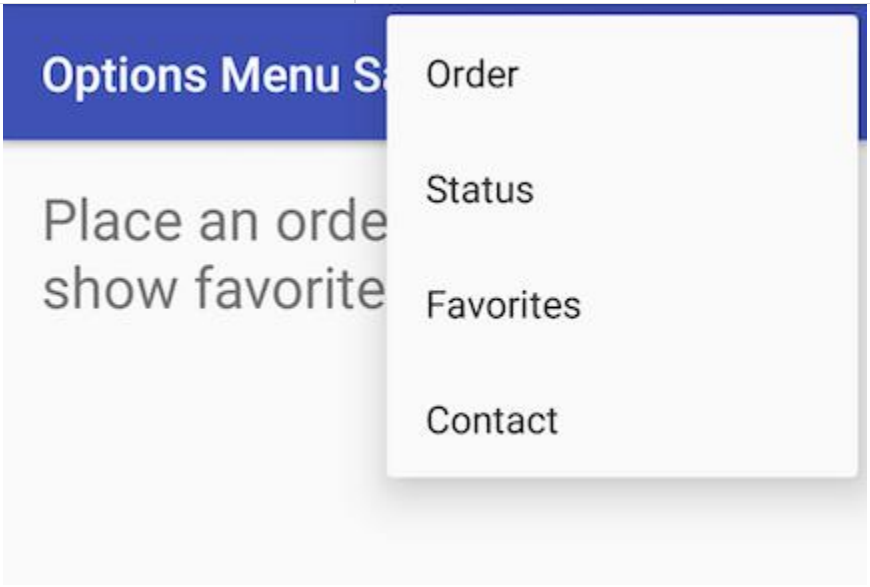
```
<item
      android:id="@+id/action_order"
      android:icon="@drawable/ic_order_white"
      android:title="@string/action_order"/>
```

- `android:title`: A string for the title of the menu item.
- `android:titleCondensed`: A string to use as a condensed title for situations in which the normal title is too long.

Position attributes

Use the `android:orderInCategory` attribute to specify the order in which the menu items appear in the menu, with the lowest number appearing higher in the menu. This is usually the order of importance of the item within the menu. For example, if you want **Order** to be first, followed by **Status**, **Favorites**, and **Contact**, the following table shows the priority of these items in the menu:

Menu item	**orderInCategory attribute**
Order	`10`
Status	`20`
Favorites	`30`
Contact	`40`



Note: While the numbers 1, 2, 3, and 4 would work in the above example, the numbers 10, 20, 30, and 40 leave ample room for menu items to be added later, between these four items.

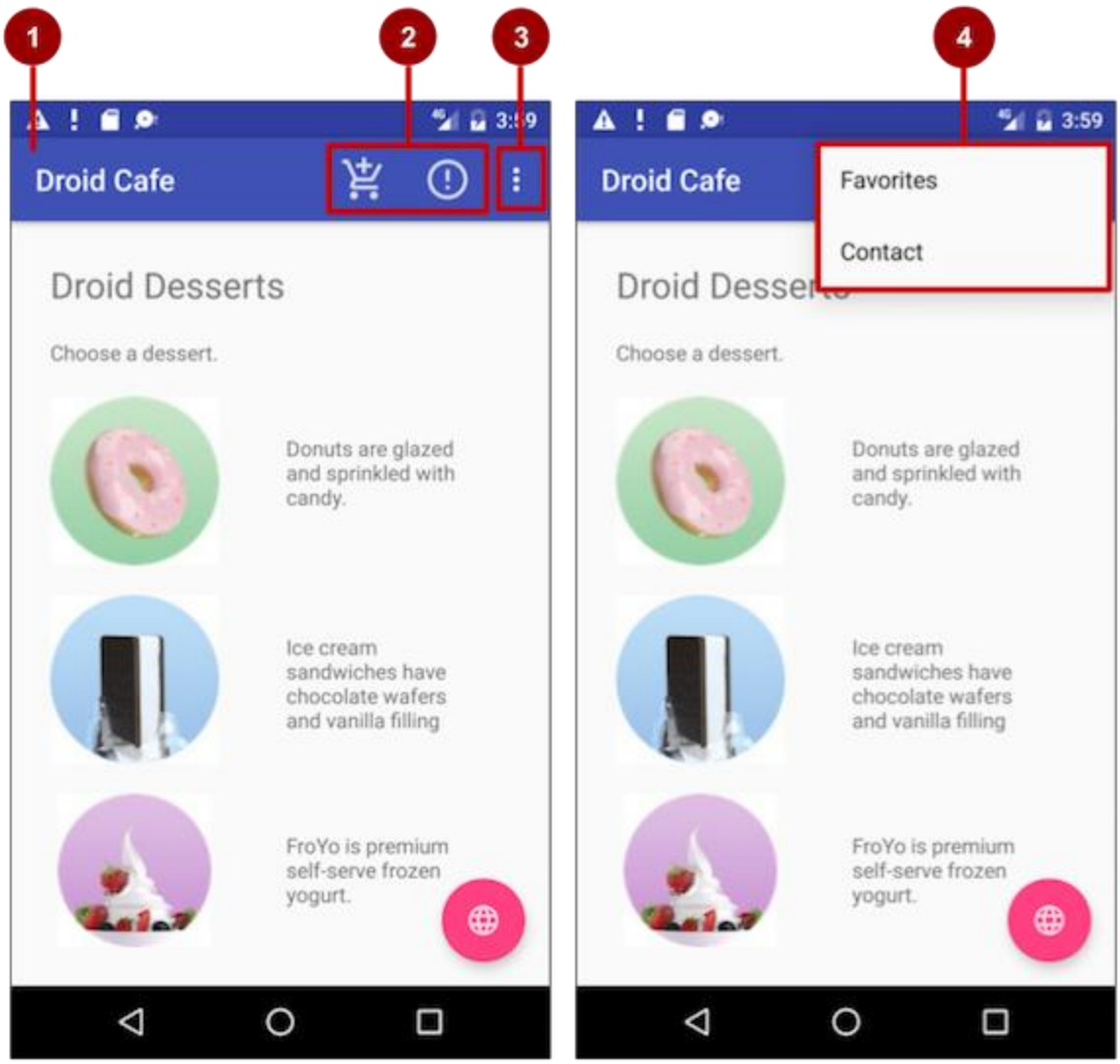
Use the `app:showAsAction` attribute to show menu items as icons in the app bar, with the following values:

- `"always"`: Always place this item in the app bar. Use this only if it's critical that the item appear in the app bar (such as a Search icon). If you set multiple items to always appear in the app bar, they might overlap something else in the app bar, such as the app title.
- `"ifRoom"`: Only place this item in the app bar if there is room for it. If there is not enough room for all the items marked `"ifRoom"`, the items with the lowest `orderInCategory` values are displayed in the app bar. The remaining items are displayed in the overflow menu.
- `"never"`: Never place this item in the app bar. Instead, list the item in the app bar's overflow menu.
- `"withText"`: Also include the title text (defined by `android:title`) with the item. This attribute is used primarily to include the title with the icon in the app bar, because if the item appears in the overflow menu, the title text appears regardless.

For example, the following menu item's icon appears in the app bar only if there is room for it:

```
<item
   android:id="@+id/action_favorites"
   android:icon="@drawable/ic_favorites_white"
   android:orderInCategory="40"
   android:title="@string/action_favorites"
   app:showAsAction="ifRoom" />
```

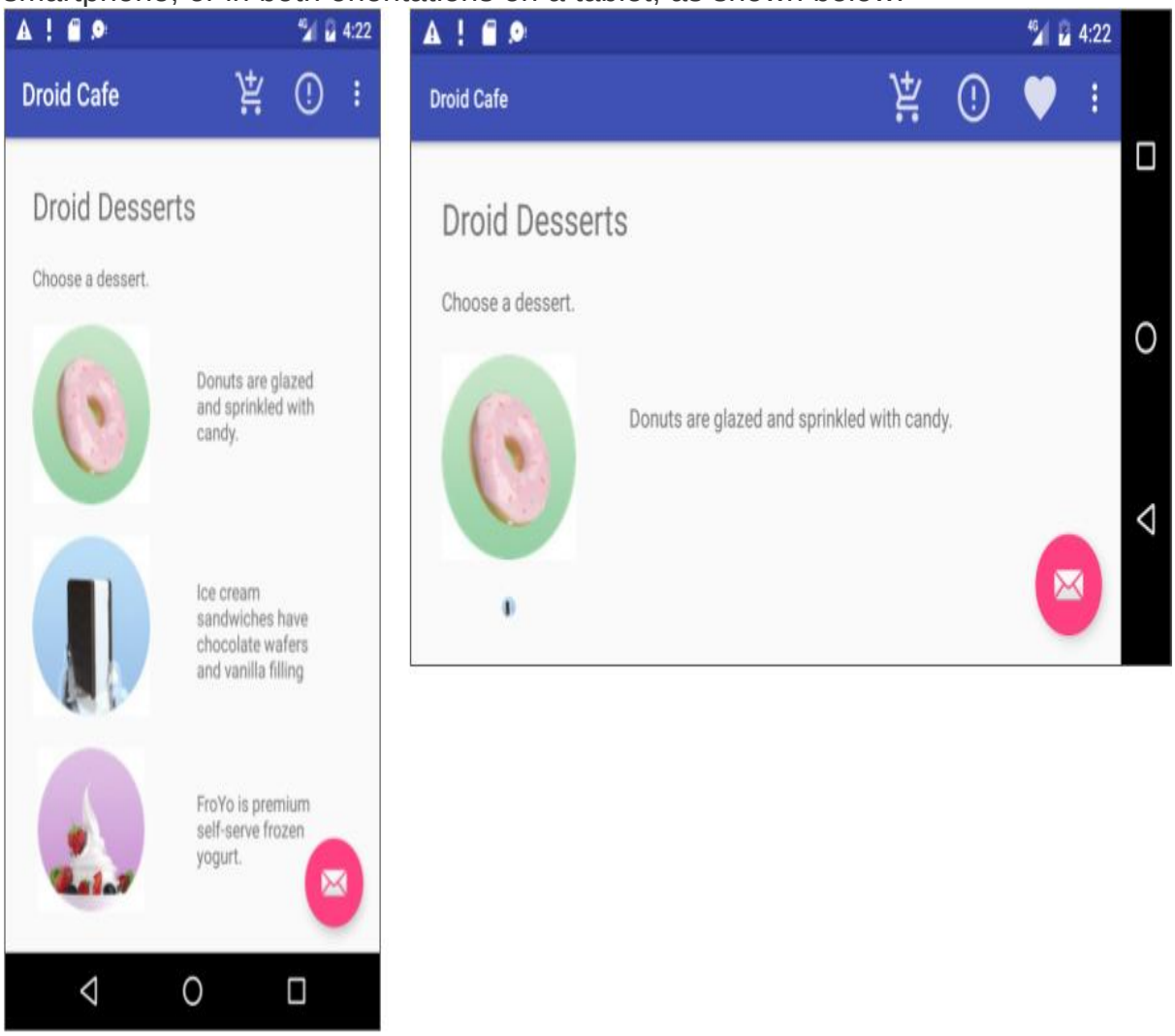
https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/



In the above figure:

1. **Options menu action icons**. The first two menu items appear as action icons in the app bar: **Order** (the shopping cart icon) and **Info** (the "**i**" icon).
2. **Overflow button**. Clicking the overflow button shows the overflow menu.
3. **Options overflow menu**. The overflow menu shows more of the options menu: **Favorites** (the heart icon) and **Contact**. **Favorites** doesn't fit into the app bar in vertical orientation, but may appear in horizontal orientation on a

smartphone, or in both orientations on a tablet, as shown below.



Inflating the menu resource

If you start an app project using the Basic Activity template, the template adds the code for inflating the options menu with `MenuInflater`, so you can skip this step.
If you are *not* using the Basic Activity template, inflate the menu resource in your activity by overriding the `onCreateOptionsMenu()` method and using the `getMenuInflater()` method of the `Activity` class.
The `getMenuInflater()` method returns a `MenuInflater`, which is a class used to instantiate menu XML files into `Menu` objects. The `MenuInflater` class provides the `inflate()` method, which takes two parameters:

- The resource `id` for an XML layout resource to load (`R.menu.menu_main` in the following example).
- The `Menu` to inflate into (`menu` in the following example).

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
 getMenuInflater().inflate(R.menu.menu_main, menu);
 return true;
}
```

Handling the menu-item click

As with a `Button`, the `android:onClick` attribute defines a method to call when this menu item is clicked. You must declare the method in the `Activity` as `public` and accept a `MenuItem` as its only parameter, which indicates the item clicked.
For example, you could define the **Favorites** item in the menu resource file to use the `android:onClick` attribute to call the `onFavoritesClick()` method:

```
<item
        android:id="@+id/action_favorites"
        android:icon="@drawable/ic_favorites_white"
        android:orderInCategory="40"
        android:title="@string/action_favorites"
        app:showAsAction="ifRoom"
        android:onClick="onFavoritesClick" />
```

You would declare the `onFavoritesClick()` method in the `Activity`:

```
public void onFavoritesClick(MenuItem item) {
    // The item parameter indicates which item was clicked.
    // ... Add code to handle the Favorites click.
}
```

However, the `onOptionsItemSelected()` method can handle all the menu-item clicks in one place and determine which menu item the user clicked. This makes your code easier to understand.
For example, you can use a `switch case` block to call the appropriate method (such as `showOrder`) based on the menu item's `id`. You retrieve the `id` using the `getItemId()` method:

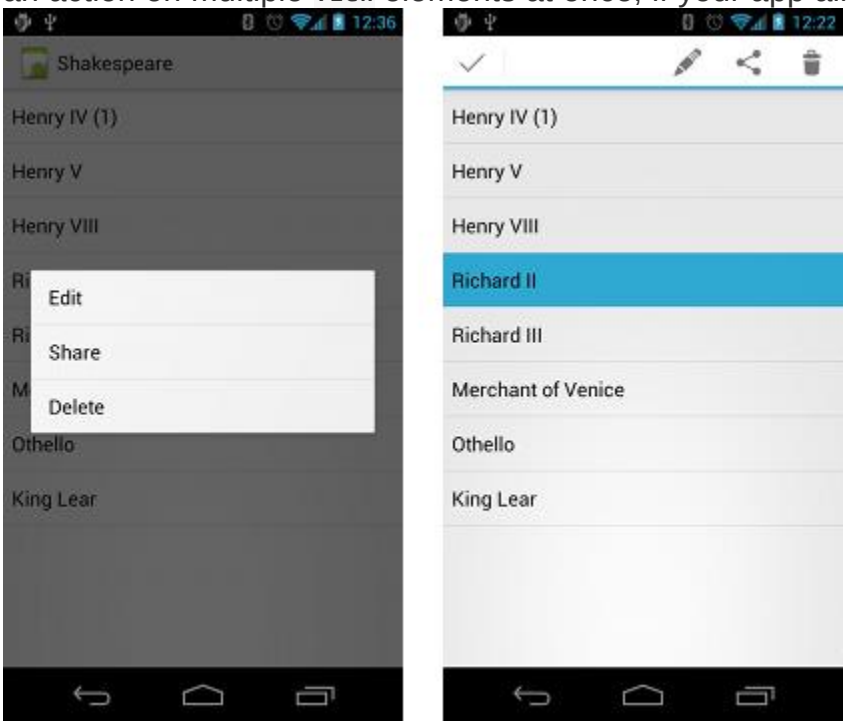
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      case R.id.action_order:
         showOrder();
         return true;
      case R.id.action_status:
         showStatus();
         return true;
      case R.id.action_contact:
         showContact();
         return true;
      default:
         // Do nothing
   }
   return super.onOptionsItemSelected(item);
}
```

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Contextual menus

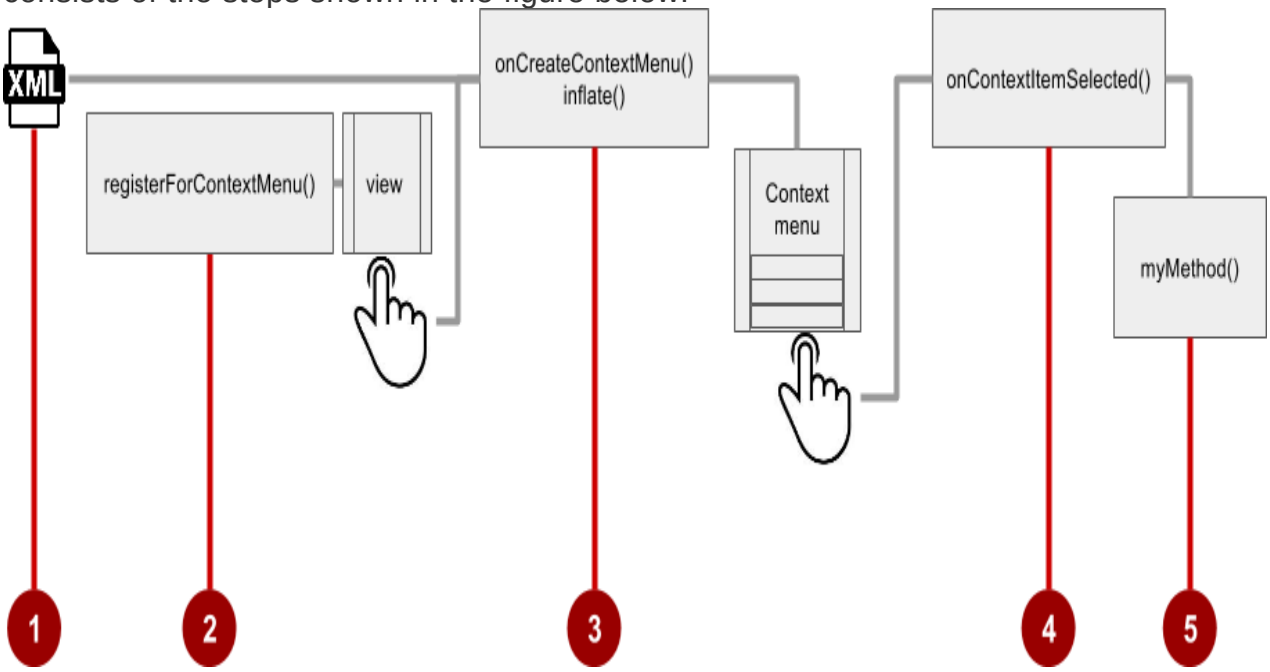
Use a *contextual menu* to allow users to take an action on a selected `View`. Contextual menus are most often used for items in a `RecyclerView`, `GridView`, or other view collection in which the user can perform direct actions on each item.
Android provides two kinds of contextual menus:

- A *context menu*, shown on the left side in the figure below, appears as a floating list of menu items when the user performs a long tap on a `View`. It is typically used to modify the `View` or use it in some fashion. For example, a context menu might include **Edit** to edit the contents of a `View`, **Delete** to delete a `View`, and **Share** to share a `View` over social media. Users can perform a contextual action on one selected `View` at a time.
- A *contextual action bar*, shown on the right side of the figure below, appears at the top of the screen in place of the app bar or underneath the app bar, with action items that affect one or more selected `View` elements. Users can perform an action on multiple `View` elements at once, if your app allows it.



Floating context menu

The familiar resource-inflate design pattern is used to create a context menu, modified to include registering (associating) the context menu with a `View`. The pattern consists of the steps shown in the figure below.



1. Create an XML menu resource file for the menu items. Assign appearance and position attributes as described in the previous section for the options menu.
2. Register a `View` to the context menu using the `registerForContextMenu()` method of the `Activity` class.
3. Implement the `onCreateContextMenu()` method in your `Activity` to inflate the menu.
4. Implement the `onContextItemSelected()` method in your `Activity` to handle menu-item clicks.
5. Create a method to perform an action for each context menu item.

Creating the XML resource file

To create the XML menu resource directory and file, follow the steps in the previous section for the options menu. However, use a different name for the file, such as `menu_context`. Add the context menu items within `<item ... />` tags.
For example, the following code defines the **Edit** menu item:

```
<item
   android:id="@+id/context_edit"
   android:title="Edit"
   android:orderInCategory="10"/>
```

Registering a View to the context menu

To register a `View` to the context menu, call the `registerForContextMenu()` method with the `View`. Registering a context menu for a view sets the `View.OnCreateContextMenuListener` on the `View` to this activity, so that `onCreateContextMenu()` is called when it's time to show the context menu. (You implement `onCreateContextMenu` in the next section.)
For example, in the `onCreate()` method for the `Activity`, you would add `registerForContextMenu()`:

```
// Registering the context menu to the TextView of the article.
TextView article_text = findViewById(R.id.article);
registerForContextMenu(article_text);
```

Multiple views can be registered to the same context menu. If you want each item in a `ListView` or `GridView` to provide the same context menu, register all items for a context menu by passing the `ListView` or `GridView` to `registerForContextMenu()`.

Implementing the onCreateContextMenu() method

When the registered `View` receives a long-click event, the system calls the `onCreateContextMenu()` method, which you can override in your `Activity`. (Long-click events are also called *touch & hold* events and *long-press* events.)
The `onCreateContextMenu()` method is where you define the menu items, usually by inflating a menu resource.
For example:

```
@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);
}
```

In the code above:

- The `menu` parameter for `onCreateContextMenu()` is the context menu to be built.
- The `v` parameter is the `View` registered for the context menu.
- The `menuInfo` parameter is extra information about the `View` registered for the context menu. This information varies depending on the class of the `v` parameter, which could be a `RecyclerView` or a `GridView`.

If you are registering a `RecyclerView` or a `GridView`, you instantiate a `ContextMenu.ContextMenuInfo` object to provide information about the item selected, and pass it as `menuInfo`, such as the row `id`, position, or child `View`.
The `MenuInflater` class provides the `inflate()` method, which takes two parameters:

- The resource `id` for an XML layout resource to load. In the example above, the `id` is `menu_context`.
- The `Menu` to inflate into. In the example above, the `Menu` is `menu`.

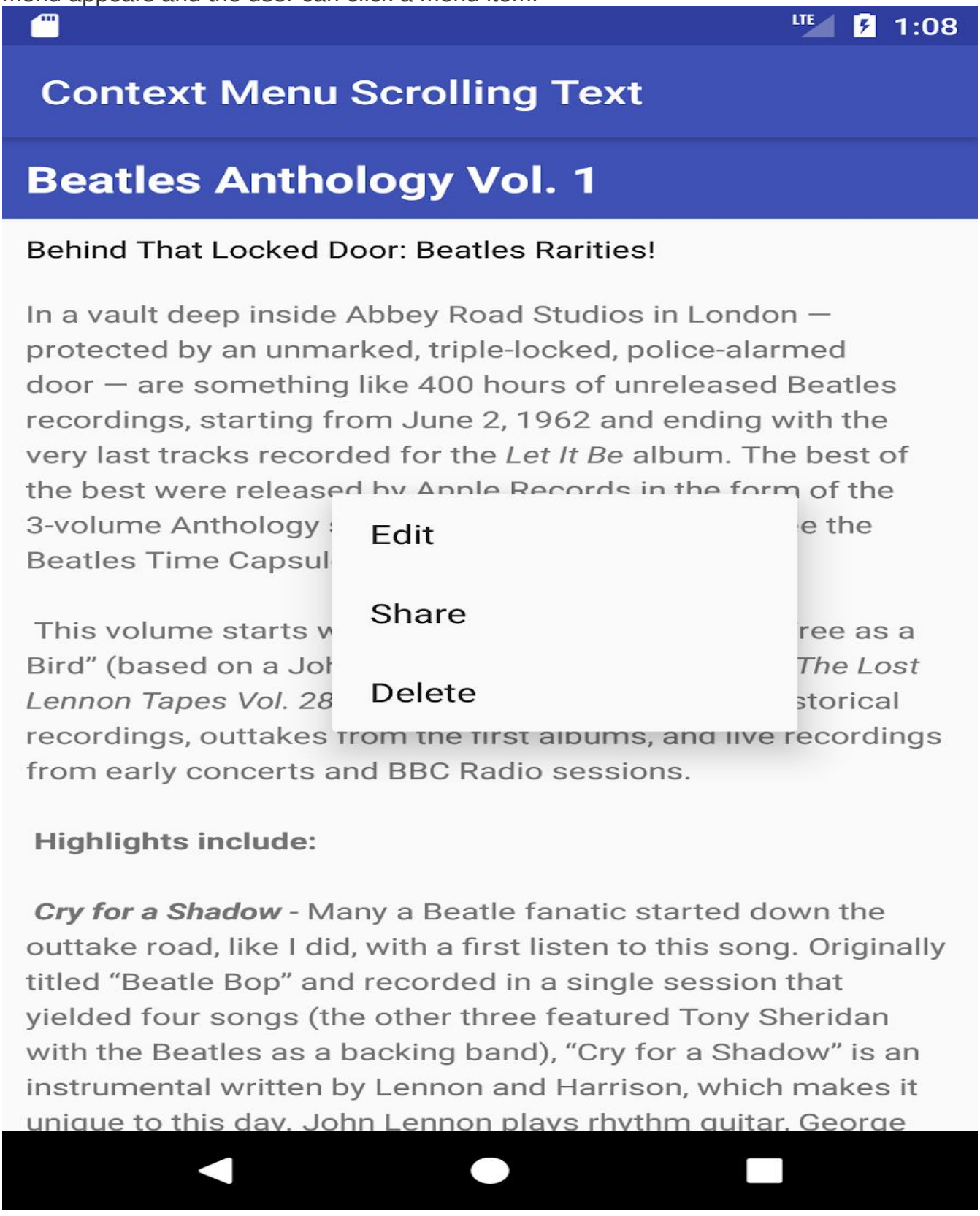
Implementing the onContextItemSelected() method

When the user clicks on a menu item, the system calls the `onContextItemSelected()` method. You override this method in your `Activity` in order to determine which menu item was clicked, and for which view the menu is appearing. You also use it to implement the appropriate action for the menu items, such as `editNote()` and `shareNote()` in the following code snippet for the **Edit** and **Share** menu items:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      case R.id.context_edit:
         editNote();
         return true;
      case R.id.context_share:
         shareNote();
         return true;
      default:
         return super.onContextItemSelected(item);
   }
}
```

The above code snippet uses the `getItemId()` method to get the `id` for the selected menu item, and uses it in a `switch case` block to determine which action to take. The `id` is the `android:id` attribute assigned to the menu item in the XML menu resource file.

When the user performs a long-click on the article in the `TextView`, the floating context menu appears and the user can click a menu item.

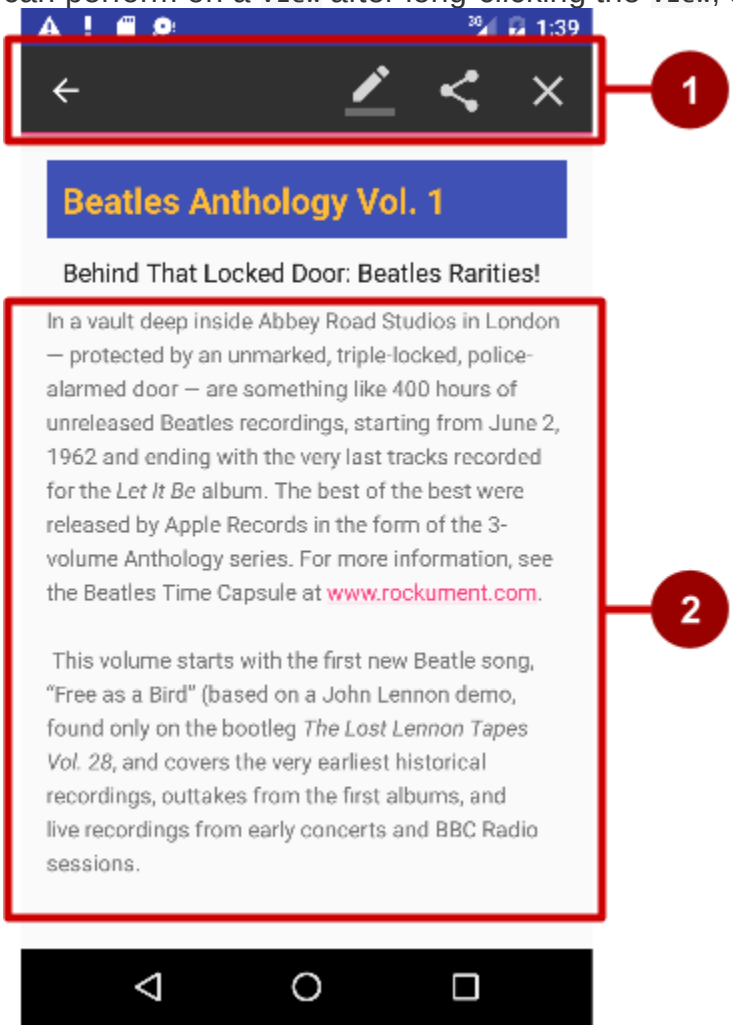


If you are using the `menuInfo` information for a `RecyclerView` or a `GridView`, you would add a statement before the `switch case` block to gather the specific information about the selected `View` (for `info`) by using `AdapterView.AdapterContextMenuInfo`:

```
AdapterView.AdapterContextMenuInfo info =
            (AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
```

Contextual action bar

A *contextual action bar* appears at the top of the screen to present actions the user can perform on a `View` after long-clicking the `View`, as shown in the figure below.



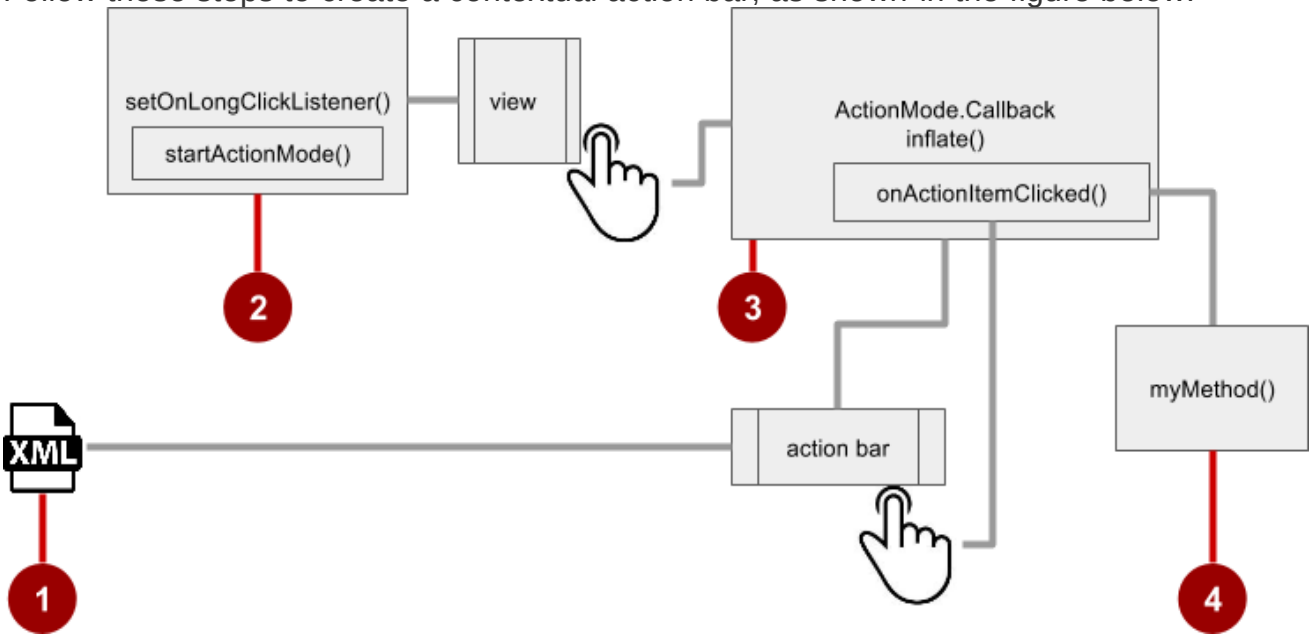
In the above figure:

1. **Contextual action bar**. The bar offers three actions on the right side (**Edit**, **Share**, and **Delete**) and the **Done** button (left arrow icon) on the left side.
2. **View**. `View` on which a long-click triggers the contextual action bar.

The contextual action bar appears only when *contextual action mode*, a system implementation of `ActionMode`, occurs as a result of the user performing a long-click on one or more selected `View` elements.
`ActionMode` represents UI mode for providing alternative interaction, replacing parts of the normal UI until finished. For example, text selection is implemented as an `ActionMode`, as are contextual actions that work on a selected item on the screen.
Selecting a section of text or long-clicking a view triggers `ActionMode`.
While this mode is enabled, the user can select multiple items, if your app allows it. The user can also deselect items, and continue to navigate within the activity. `ActionMode` is disabled when one of the following things occur:

- The user deselects all items.
- The user presses the Back button.
- The user taps **Done** (the left-arrow icon) on the left side of the action bar.

When `ActionMode` is disabled, the contextual action bar disappears.
Follow these steps to create a contextual action bar, as shown in the figure below:



1. Create an XML menu resource file for the menu items, and assign an icon to each one (as described in a previous section).
2. Set the long-click listener using `setOnLongClickListener()` to the `View` that should trigger the contextual action bar. Call `startActionMode()` within the `setOnLongClickListener()` method when the user performs a long tap on the `View`.
3. Implement the `ActionMode.Callback` interface to handle the `ActionMode` lifecycle. Include in this interface the action for responding to a menu-item click in the `onActionItemClicked()` callback method.
4. Create a method to perform an action for each context menu item.

Creating the XML resource file

Create the XML menu resource directory and file by following the steps in the previous section on the options menu. Use a suitable name for the file, such as `menu_context`. Add icons for the context menu items. For example, the **Edit** menu item would have these attributes:

```
<item
   android:id="@+id/context_edit"
   android:orderInCategory="10"
   android:icon="@drawable/ic_action_edit_white"
   android:title="Edit" />
```

The standard contextual action bar has a dark background. Use a light or white color for the icons. If you are using clip art icons, choose **HOLO_DARK** for the **Theme** drop-down menu when creating the new image asset.

Setting the long-click listener

Use `setOnLongClickListener()` to set a long-click listener to the `View` that should trigger the contextual action bar. Add the code to set the long-click listener to the `Activity` using the `onCreate()` method. Follow these steps:

1. Declare the member variable `mActionMode`:

```
2.      private ActionMode mActionMode;
```

You will call `startActionMode()` to enable `ActionMode`, which returns the `ActionMode` created. By saving this in a member variable (`mActionMode`), you can make changes to the contextual action bar in response to other events.

3. Set up the contextual action bar listener in the `onCreate()` method, using `View` as the type in order to use the `setOnLongClickListener`:

```
4.   @Override
5.   protected void onCreate(Bundle savedInstanceState) {
6.      // ... The rest of the onCreate code.
7.      View articleView = findViewById(article);
8.      articleView.setOnLongClickListener(new View.OnLongClickListener()
9.      {
10.        // Start ActionMode after long-click.
11.     });
12.  }
```

Implementing the ActionMode.Callback interface

Before you can add the code to `onCreate()` to start `ActionMode`, you must implement the `ActionMode.Callback` interface to manage the `ActionMode` lifecycle. In its callback methods, you can specify the actions for the contextual action bar, and respond to clicks on action items.

1. Add the following method to the `Activity` to implement the interface:

```
2.   public ActionMode.Callback mActionModeCallback = new
3.                                      ActionMode.Callback() {
4.      // ... Code to create ActionMode.
5.   }
```

6. Add the `onCreateActionMode()` code within the brackets of the above method to create `ActionMode`:

```
@Override
public boolean onCreateActionMode(ActionMode mode, Menu menu) {
      // Inflate a menu resource providing context menu items
      MenuInflater inflater = mode.getMenuInflater();
      inflater.inflate(R.menu.menu_context, menu);
      return true;
}
```

The `onCreateActionMode()` method inflates the menu using the same pattern used for a floating context menu. But this inflation occurs *only* when `ActionMode` is created, which is when the user performs a long-click. The `MenuInflater` class provides the `inflate()` method, which takes as a parameter the resource `id` for an XML layout resource to load (`menu_context` in the above example), and the `Menu` to inflate into (`menu` in the above example).

14. Add the `onActionItemClicked()` method with your handlers for each menu item:

```
@Override
public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
   switch (item.getItemId()) {
      case R.id.context_edit:
         editNote();
         mode.finish();
         return true;
      case R.id.context_share:
         shareNote();
         mode.finish();
         return true;
      default:
         return false;
}
```

The above code above uses the `getItemId()` method to get the `id` for the selected menu item, and uses it in a `switch case` block to determine which action to take.
The `id` in each `case` statement is the `android:id` attribute assigned to the menu item in the XML menu resource file.
The actions shown are the `editNote()` and `shareNote()` methods, which you create in the `Activity`. After the action is picked, you use the `mode.finish()` method to close the contextual action bar.

29. Add the `onPrepareActionMode()` and `onDestroyActionMode()` methods, which manage the ActionMode lifecycle:

```
@Override
public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
      return false; // Return false if nothing is done.
}
```

The `onPrepareActionMode()` method shown above is called each time `ActionMode` occurs, and is always called after `onCreateActionMode()`.

```
@Override
public void onDestroyActionMode(ActionMode mode) {
   mActionMode = null;
}
```

The `onDestroyActionMode()` method shown above is called when the user exits `ActionMode` by clicking **Done** in the contextual action bar, or clicking on a different view.
The following is the full code for the `ActionMode.Callback` interface implementation:

```
public ActionMode.Callback mActionModeCallback = new
                                        ActionMode.Callback() {
   @Override
   public boolean onCreateActionMode(ActionMode mode, Menu menu) {
```

```
      // Inflate a menu resource providing context menu items
      MenuInflater inflater = mode.getMenuInflater();
      inflater.inflate(R.menu.menu_context, menu);
      return true;
   }

   // Called each time ActionMode is shown. Always called after
   // onCreateActionMode.
   @Override
   public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
      return false; // Return false if nothing is done
   }

   // Called when the user selects a contextual menu item
   @Override
   public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
      switch (item.getItemId()) {
         case R.id.context_edit:
            editNote();
            mode.finish();
            return true;
         case R.id.context_share:
            shareNote();
            mode.finish();
            return true;
         default:
            return false;
      }
   }

   // Called when the user exits the action mode
   @Override
   public void onDestroyActionMode(ActionMode mode) {
      mActionMode = null;
   }
};
```

Starting ActionMode

You use `startActionMode()` to start `ActionMode` after the user performs a long-click. To start `ActionMode`, add the `onLongClick()` method within the brackets of the `setOnLongClickListener` method in `onCreate()`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   // ... Rest of onCreate code
   articleView.setOnLongClickListener(new View.OnLongClickListener() {
      // Called when the user long-clicks on articleView
      public boolean onLongClick(View view) {
         if (mActionMode != null) return false;
         // Start the contextual action bar
         // using the ActionMode.Callback.
         mActionMode =
               MainActivity.this.startActionMode(mActionModeCallback);
         view.setSelected(true);
         return true;
      }
   });
}
```

The above code first ensures that the `ActionMode` instance is not recreated if it's already active by checking whether `mActionMode` is `null` before starting the action mode:

```
if (mActionMode != null) return false;
```

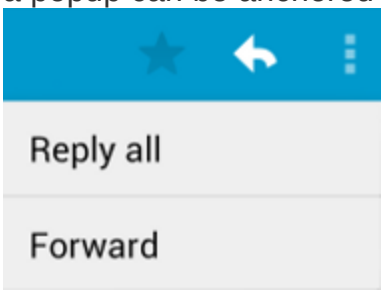
When the user performs a long-click, the call is made to `startActionMode()` using the `ActionMode.Callback` interface, and the contextual action bar appears at the top of the display. The `setSelected()` method changes the state of this `View` to selected (set to `true`).

The following is the code for the `onCreate()` method in the `Activity`, which now includes `setOnLongClickListener()` and `startActionMode()`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   // set up the contextual action bar listener
   View articleView = findViewById(article);
   articleView.setOnLongClickListener(new View.OnLongClickListener() {
      // Called when the user long-clicks on articleView.
      public boolean onLongClick(View view) {
         if (mActionMode != null) return false;
         // Start the contextual action bar using the ActionMode.Callback.
         mActionMode =
               MainActivity.this.startActionMode(mActionModeCallback);
         view.setSelected(true);
         return true;
      }
   });
}
```

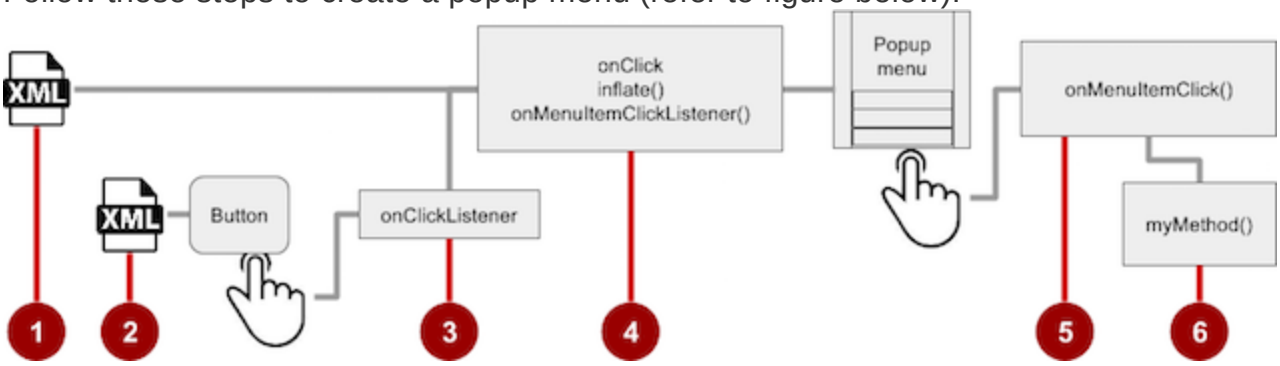
Popup menu

A `PopupMenu` is a vertical list of items anchored to a `View`. It appears below the anchor `View` if there is room, or above the `View` otherwise.
A popup menu is typically used to provide an overflow of actions (similar to the overflow action icon for the options menu) or the second part of a two-part command. Use a popup menu for extended actions that relate to regions of content in your `Activity`. Unlike a context menu, a popup menu is anchored to a `Button`, is always available, and its actions generally do not affect the content of the `View`.
For example, the Gmail app uses a popup menu anchored to the overflow icon in the app bar when showing an email message. The popup menu items **Reply**, **Reply All**, and **Forward** are *related* to the email message, but don't *affect* or *act on* the message. Actions in a popup menu should not directly affect the corresponding content (use a contextual menu to directly affect selected content). As shown below, a popup can be anchored to the overflow action button in the app bar.



Creating a popup menu

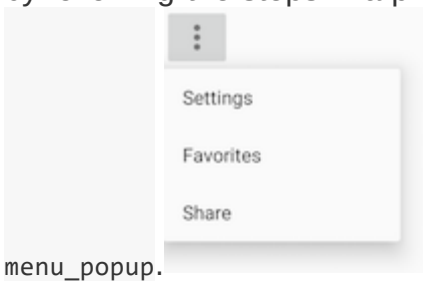
Follow these steps to create a popup menu (refer to figure below):



1. Create an XML menu resource file for the popup menu items, and assign appearance and position attributes (as described in a previous section).
2. Add an `ImageButton` for the popup menu icon in the XML activity layout file.
3. Assign `onClickListener()` to the `ImageButton`.
4. Override the `onClick()` method to inflate the popup menu and register it with `PopupMenu.OnMenuItemClickListener`.
5. Implement the `onMenuItemClick()` method.
6. Create a method to perform an action for each popup menu item.

Creating the XML resource file

Create the XML menu resource directory and file by following the steps in a previous section. Use a suitable name for the file, such as `menu_popup`.



Adding an ImageButton for the icon to click

Use an `ImageButton` in the `Activity` layout for the icon that triggers the popup menu. Popup menus are anchored to a `View` in the `Activity`, such as an `ImageButton`. The user clicks it to see the menu.

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button_popup"
    android:src="@drawable/@drawable/ic_action_popup"/>
```

Assigning onClickListener to the button

1. Create a member variable (`mButton`) in the `Activity` class definition:

```
 public class MainActivity extends AppCompatActivity {
    private ImageButton mButton;
    // ... Rest of Activity code
 }
```

6. In the `onCreate()` method for the same `Activity`, assign `onClickListener()` to the `ImageButton`:

```
 // ... Rest of Activity code
 @Override
 protected void onCreate(Bundle savedInstanceState) {
    // ... Rest of onCreate code
    mButton = (ImageButton) findViewById(R.id.button_popup);
    mButton.setOnClickListener(new View.OnClickListener() {
        // Define onClick here ...
    });
 }
```

Inflating the popup menu

As part of the `setOnClickListener()` method within `onCreate()`, add the `onClick()` method to inflate the popup menu and register it with `PopupMenu.OnMenuItemClickListener`:

```
// Define onClick here ...
@Override
public void onClick(View v) {
   // Create the instance of PopupMenu.
   PopupMenu popup = new PopupMenu(MainActivity.this, mButton);
   // Inflate the Popup using XML file.
   popup.getMenuInflater().inflate(R.menu.menu_popup, popup.getMenu());
   // Register the popup with OnMenuItemClickListener.
   popup.setOnMenuItemClickListener(new
                              PopupMenu.OnMenuItemClickListener() {
      // Add onMenuItemClick here...
      // Perform action here ...
}
```

The method instantiates a `PopupMenu` object, which is `popup` in the example above. Then the method uses the `MenuInflater` class and its `inflate()` method. The `inflate()` method takes the following parameters:

- The resource `id` for an XML layout resource to load, which is `menu_popup` in the example above.
- The `Menu` to inflate into, which is `popup.getMenu()` in the example above.

The code then registers the popup with the listener, `PopupMenu.OnMenuItemClickListener`.

Implementing onMenuItemClick

To perform an action when the user selects a popup menu item, implement the `onMenuItemClick()` callback within the above `setOnClickListener()` method. Finish the method with `popup.show` to show the popup menu:

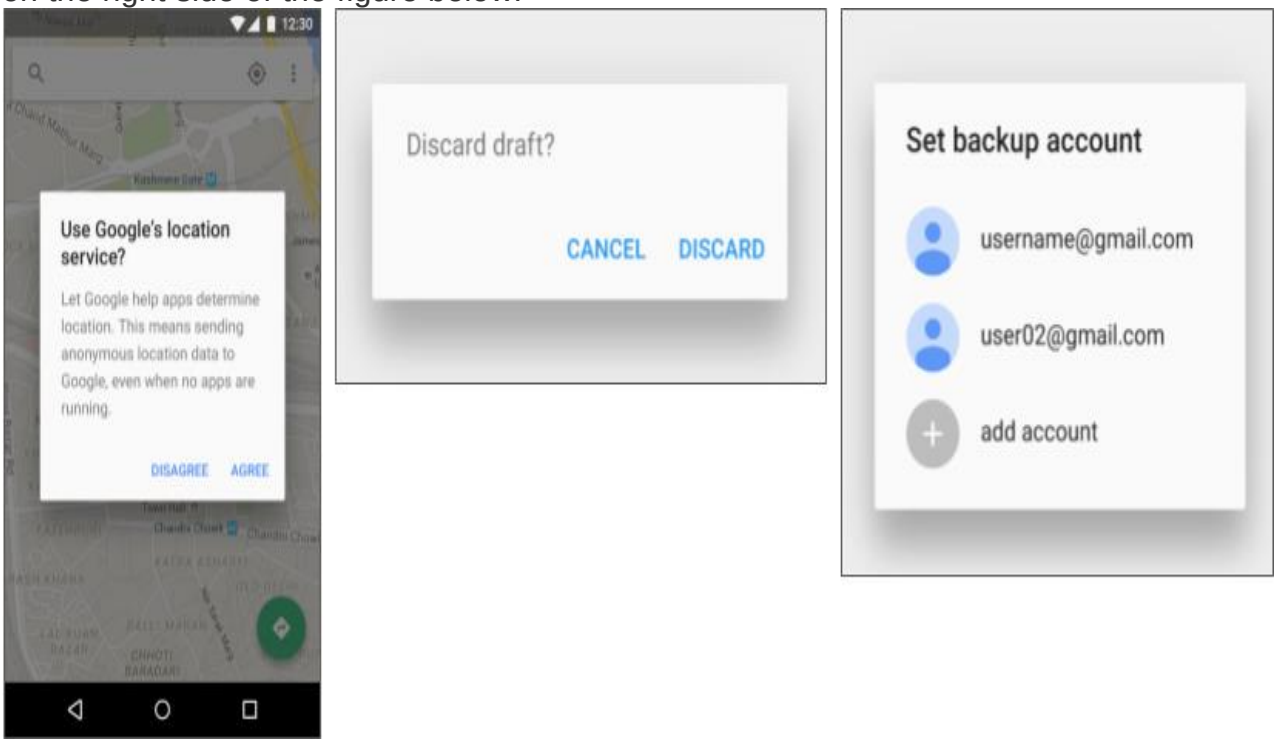
```
    // Add onMenuItemClick here...
    public boolean onMenuItemClick(MenuItem item) {
        // Perform action here ...
        return true;
    }
});
// Show the popup menu.
popup.show();
```

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Dialogs and pickers

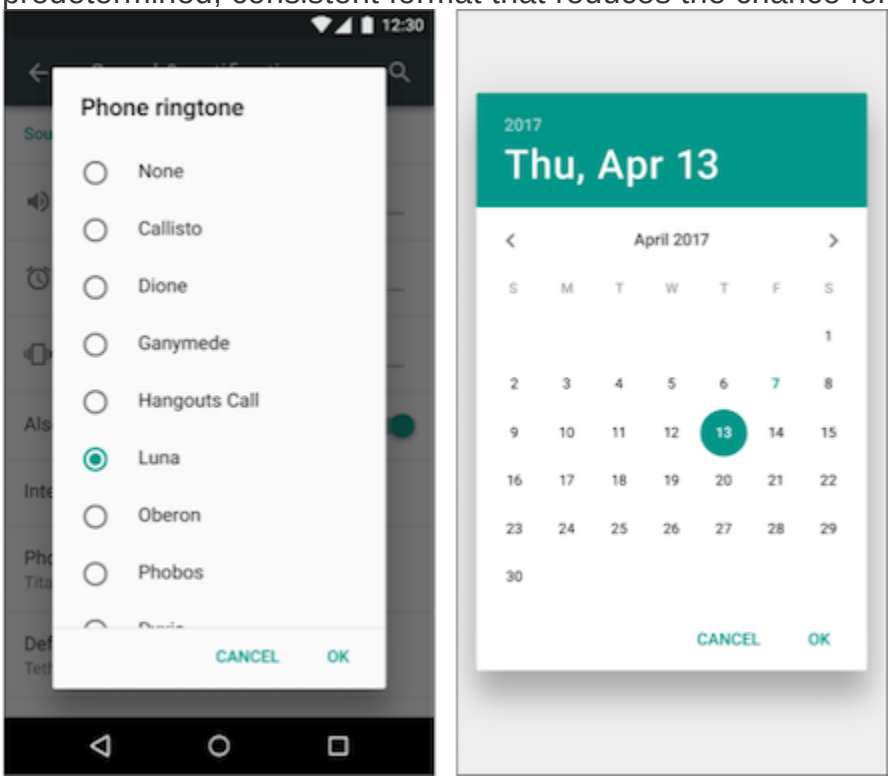
A *dialog* is a window that appears on top of the display or fills the display, interrupting the flow of `Activity`. Dialogs inform users about a specific task and may contain critical information, require decisions, or involve multiple tasks.
For example, an alert dialog might require the user to click **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**).

You can also use a dialog to provide choices in the style of radio buttons, as shown on the right side of the figure below.



The base class for all dialog components is a `Dialog`. There are several useful `Dialog` subclasses for alerting the user on a condition, showing status or progress, displaying information on a secondary device, or selecting or confirming a choice, as shown on the left side of the figure below. The Android SDK also provides ready-to-use dialog subclasses such as *pickers* for picking a time or a date, as shown on the right side of the figure below. Pickers allow users to enter information in a

predetermined, consistent format that reduces the chance for input error.



Dialogs always retain focus until dismissed or a required action has been taken.

Tip: Best practices recommend using dialogs sparingly as they interrupt the user's workflow. Read the Dialogs design guide for additional best design practices, and Dialogs in the Android developer documentation for code examples.

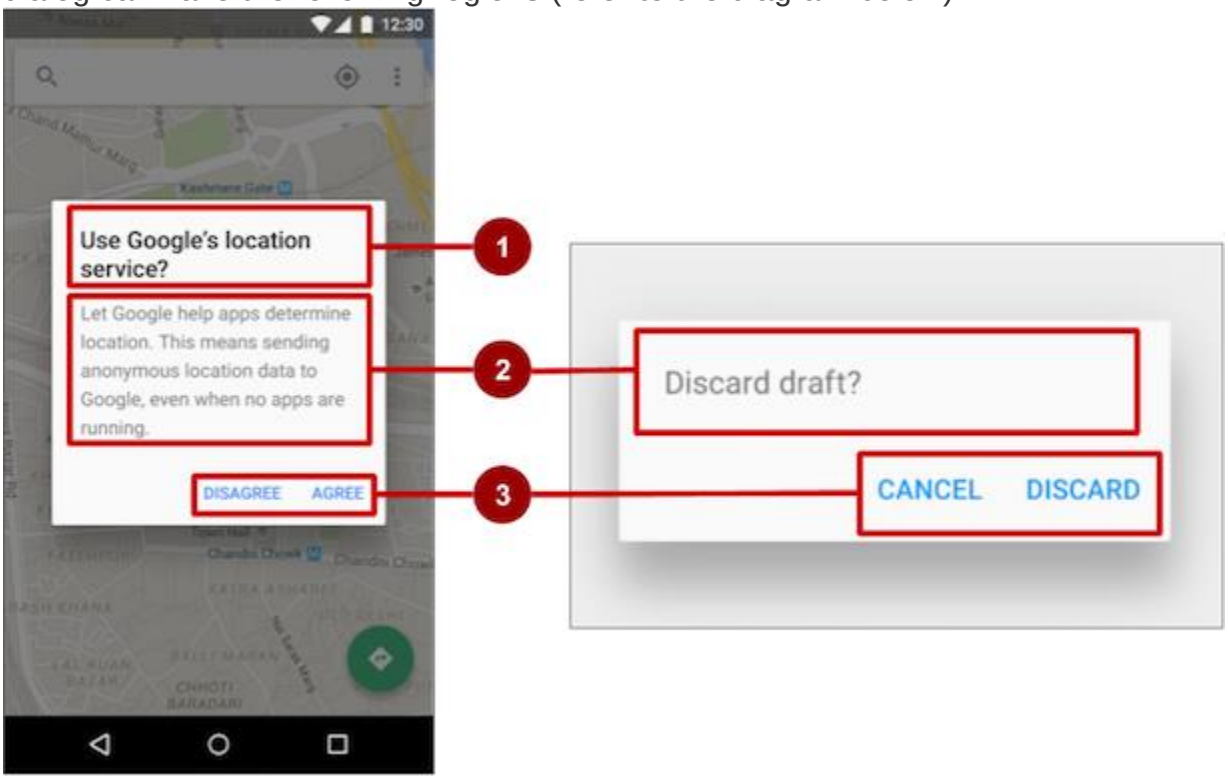
The `Dialog` class is the base class for dialogs, but you should avoid instantiating `Dialog` directly unless you are creating a custom dialog. For standard Android dialogs, use one of the following subclasses:

- `AlertDialog`: A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.
- `DatePickerDialog`: A dialog with a predefined UI that lets the user select a date.
- `TimePickerDialog`: A dialog with a predefined UI that lets the user select a time.

Showing an alert dialog

Alerts are urgent interruptions, requiring acknowledgement or action, that inform the user about a situation as it occurs, or an action *before* it occurs (as in discarding a draft). You can provide buttons in an alert to make a decision. 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 **Disagree** or **Cancel**).

Use the `AlertDialog` subclass of the `Dialog` class to show a standard dialog for an alert. The `AlertDialog` class allows you to build a variety of dialog designs. An alert dialog can have the following regions (refer to the diagram below):



1. Title: A title is optional. Most alerts don't need titles. If you can summarize a decision in a sentence or two by either asking a question (such as, "Discard draft?") or making a statement related to the action buttons (such as, "Click OK to continue"), don't bother with a title. Use a title if the situation is high-risk, such as the potential loss of connectivity or data, and the content area is occupied by a detailed message, a list, or custom layout.
2. Content area: The content area can display a message, a list, or other custom layout.
3. Action buttons: You should use no more than three action buttons in a dialog, and most have only two.

Building the AlertDialog

The `AlertDialog.Builder` class uses the *builder* design pattern, which 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.

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.
If `AlertDialog.Builder` is not recognized as you enter it, you may need to add the following `import` statements to the `Activity`:

```
import android.content.DialogInterface;
import android.support.v7.app.AlertDialog;
```

The following creates the dialog object (`myAlertBuilder`) and sets the title (the string resource `alert_title`) and message (the string resource `alert_message`):

```
AlertDialog.Builder myAlertBuilder = new
                    AlertDialog.Builder(MainActivity.this);
myAlertBuilder.setTitle(R.string.alert_title);
myAlertBuilder.setMessage(R.string.alert_message);
```

Setting the button actions for the alert dialog

Use the `setPositiveButton()` and `setNegativeButton()` methods to set the button actions for the alert dialog. These methods require a title for the button and the `DialogInterface.OnClickListener` class that defines the action to take when the user presses the button:

```
myAlertBuilder.setPositiveButton("OK", new
                                 DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int which) {
           // User clicked OK button.
           // ... Action to take when OK is clicked.
      }
});
myAlertBuilder.setNegativeButton("Cancel", new
                                 DialogInterface.OnClickListener() {
      public void onClick(DialogInterface dialog, int which) {
         // User clicked the CANCEL button.
         // ... Action to take when CANCEL is clicked.
      }
});
```

You can add only one of each button type to an `AlertDialog`. For example, you can't have more than one "positive" button.
Tip: You can also set a "neutral" button with `setNeutralButton()`. The neutral button appears between the positive and negative buttons. Use a neutral button, such as **Remind me later**, if you want the user to be able to dismiss the dialog and decide later.

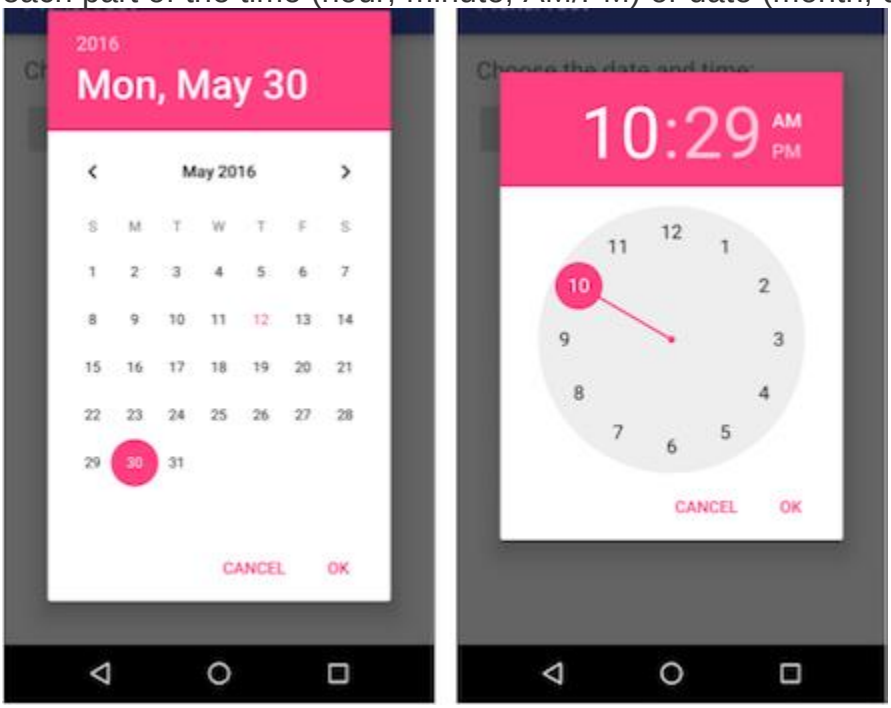
Displaying the dialog

To display the dialog, call its `show()` method:

```
alertDialog.show();
```

Date and time pickers

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



When showing a picker, you should use an instance of `DialogFragment`, a subclass of `Fragment`, which displays a dialog window floating on top of its `Activity` window. A `Fragment` 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. A `Fragment` receives its own input events, and you can add or remove it while the `Activity` is running. You might combine multiple `Fragment` elements in a single `Activity` to build a multiple-pane UI, or reuse a `Fragment` in more than one `Activity`. To learn about `Fragment`, 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. You can also use `DialogFragment` to manage the dialog lifecycle.

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

Adding a fragment

To add a `Fragment` for the date picker, create a blank `Fragment` without a layout XML, and without factory methods or interface callbacks:

1. Expand **app > java > com.example.android...** and select an `Activity` (such as **MainActivity**).
2. Choose **File > New > Fragment > Fragment (Blank)**, and name the `Fragment` (such as **DatePickerFragment**). Clear all three checkbox options so that you do *not* create a layout XML, do *not* include `Fragment` factory methods, and do *not* include interface callbacks. You don't need to create a layout for a standard picker. Click **Finish** to create the `Fragment`.

Extending DialogFragment for the picker

The next step is to create a standard picker with a listener. Follow these steps:

1. 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:

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

As you type **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. When you extend `DialogFragment`, you should override the `onCreateDialog()` callback method, rather than `onCreateView`. 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.

```
6.   @NonNull
```

```
7.   @Override
8.   public Dialog onCreateDialog(Bundle savedInstanceState) {
9.   }
```

10. You use your version of the callback method to initialize the `year`, `month`, and `day` for the date picker. For example, you can 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**.

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

The `Calendar` class sets the default date as the current date—it converts between a specific instant in time and a set of calendar fields such as `YEAR`, `MONTH`, `DAY_OF_MONTH`, and `HOUR`. `Calendar` is locale-sensitive. The `Calendar` `getInstance()` method returns a `Calendar` whose fields are initialized with the current date and time.

16. Add the following statement to the end of the method to create a new instance of the date picker and return it:

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

Showing the picker

To show the picker, add a method to the `Activity` that creates an instance of `FragmentManager` using `getSupportFragmentManager()`. You can then use the method as the handler for the `android:onClick` attribute for a button or other input control.

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

For more information about the `Fragment` class, see Fragments.

Processing the user's picker choice

When the user makes a selection in the date picker, the system calls the `onDateSet()` method, so you can use `onDateSet()` to manipulate the chosen date:

1. Open an `Activity` and add a method that takes the `year`, `month`, and `day` as arguments. You can then use this method to take action. For example, in this method you can convert the `month`, `day`, and `year` to separate strings so that you can concatenate them with slash marks for the U.S. date format:

```
2.   public void processDatePickerResult(
3.                          int year, int month, int day) {
4.       String month_string = Integer.toString(month + 1);
5.       String day_string = Integer.toString(day);
6.       String year_string = Integer.toString(year);
7.       String dateMessage = (month_string + "/"
8.                             + day_string + "/" + year_string);
9.       // ... Code to do some action with dateMessage.
10.  }
```

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.

11. Add code to the `onDateSet()` method in the `Fragment` to invoke `processDatePickerResult()` in the activity 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);
}
```

When you use the `getActivity()` method in a `Fragment`, the method returns the activity with which the fragment is associated. You need to do this because you can't call a method in the activity without the activity context—you would have to use an intent, as you learned in another lesson. The activity inherits the context, so you can use the activity as the context for calling the method, as in `activity.processDatePickerResult`.

The time picker

Follow the same procedures outlined above for a date picker, with the following differences:

- The `Fragment` should extend `DialogFragment` and implement `TimePickerDialog.OnTimeSetListener`.
- Override the `onTimeSet()` method.
- Use `onCreateDialog()` to initialize the time and return the dialog, as you did with the date picker.
- Create a method to instantiate the picker `DialogFragment`, as you did with the date picker.
- Create a method to process the result as you did with the date picker.
- Use `onTimeSet()` to get the time and pass it to the method to process the result.

You can read all about setting up pickers in Pickers.

Related practical

The related practical is 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

4.4: User navigation

Contents:

- Providing users with a path through your app
- Back-button navigation
- Hierarchical navigation patterns
- Ancestral navigation (the Up button)
- Descendant navigation
- Lateral navigation with tabs and swipes
- Related practical
- Learn more

Providing users with a path through your app

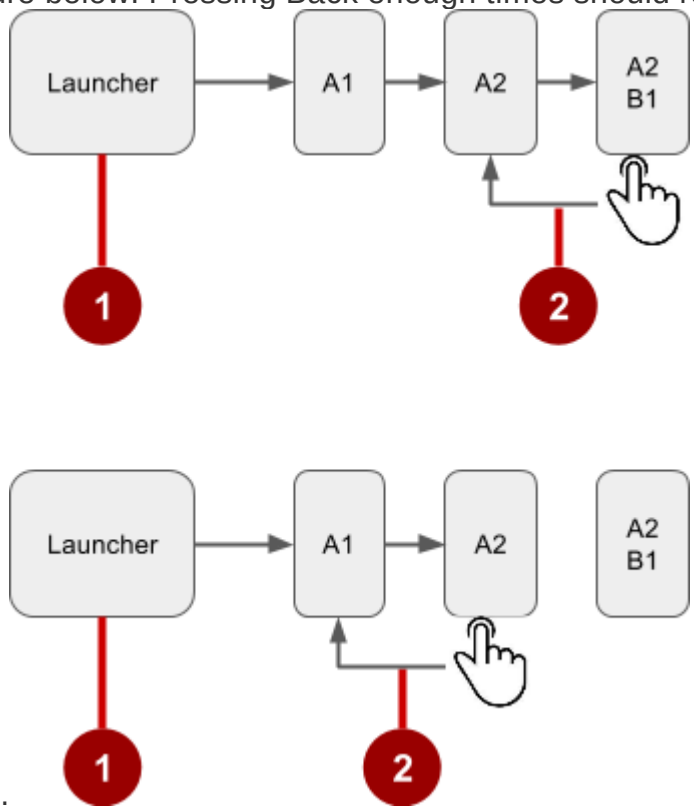
In the early stages of developing an app, you should determine the path you want users to take through your app to do each task. (The tasks are things like placing an order or browsing content.) Each path enables users to navigate across, into, and out of the tasks and pieces of content within the app.

Often you need several paths through your app that offer the following types of navigation:

- *Back* navigation, where users navigate to the previous screen using the Back button.
- *Hierarchical* navigation, where users navigate through a hierarchy of screens. The hierarchy is organized with a *parent* screen for every set of *child* screens.

Back-button navigation

Back-button navigation—navigation back through the history of screens—is deeply rooted in the Android system. Android users expect the Back button in the bottom left corner of every screen to take them to the previous screen. The set of historical screens always starts with the user's Launcher (the device's Home screen), as shown in the figure below. Pressing Back enough times should return the user back to the Launcher.



In the figure above:

1. Starting from Launcher.
2. Clicking the Back button to navigate to the previous screen.

You don't have to manage the Back button in your app. The system handles tasks and the *back stack*—the list of previous screens—automatically. The Back button by default simply traverses this list of screens, removing the current screen from the list as the user presses it.

There are, however, cases where you may want to override the behavior for the Back button. For example, if your screen contains an embedded web browser in which users can interact with page elements to navigate between web pages, you may wish to trigger the embedded browser's default back behavior when users press the device's Back button.

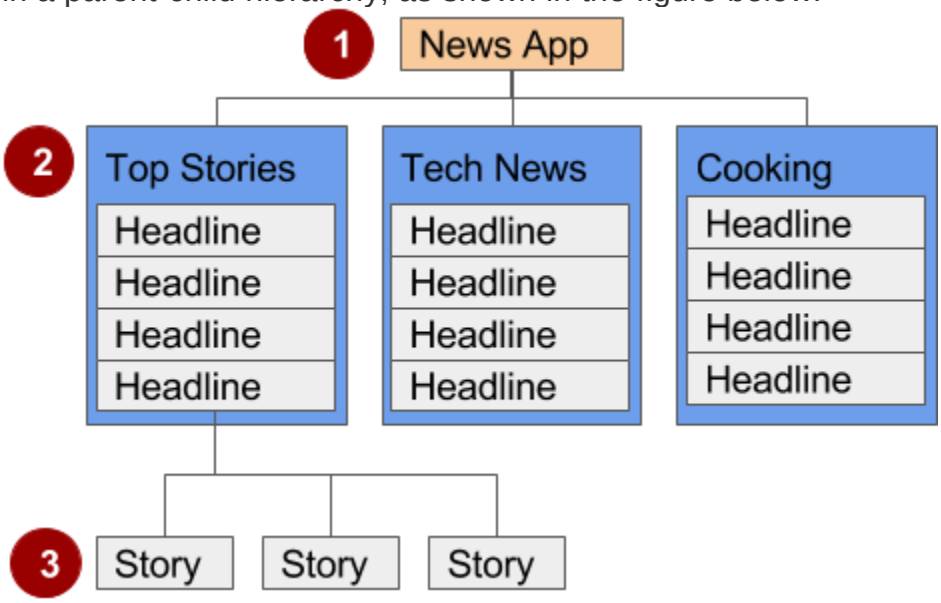
The `onBackPressed()` method of the `Activity` class is called whenever the `Activity` detects the user's press of the Back key. The default implementation simply finishes the current `Activity`, but you can override this to do something else:

```
@Override
public void onBackPressed() {
    // Add the Back key handler here.
    return;
}
```

If your code triggers an embedded browser with its own behavior for the Back key, you should return the Back key behavior to the system's default behavior if the user uses the Back key to go beyond the beginning of the browser's internal history.

Hierarchical navigation patterns

To give the user a path through the full range of an app's screens, the best practice is to use some form of hierarchical navigation. An app's screens are typically organized in a parent-child hierarchy, as shown in the figure below:



In the figure above:

1. Parent screen
2. First-level child screen siblings
3. Second-level child screen siblings

Parent screen

A parent screen (such as a news app's home screen) enables navigation down to *child* screens.

- The main `Activity` of an app is usually the parent screen.
- Implement a parent screen as an activity with *descendant* navigation to one or more child screens.

First-level child screen siblings

Siblings are screens in the same position in the hierarchy that share the same parent screen (like brothers and sisters).

- In the first level of siblings, the child screens may be *collection* screens that collect the headlines of stories, as shown above.
- Implement each child screen as an `Activity` or `Fragment`.
- Implement *lateral* navigation to navigate from one sibling to another on the same level.
- If there is a second level of screens, the first level child screen is the *parent* to the second level child screen siblings. Implement *descendant* navigation to the second-level child screens.

Second-level child screen siblings

In news apps and others that offer multiple levels of information, the second level of child screen siblings might offer content, such as stories.

- Implement each second-level child screen sibling as another `Activity` or `Fragment`.
- Stories at this level may include embedded story elements such as videos, maps, and comments, which might be implemented as fragments.

You can enable the user to navigate up to and down from a parent, and sideways among siblings:

- *Descendant* navigation: Navigating down from a parent screen to a child screen.
- *Ancestral* navigation: Navigating up from a child screen to a parent screen.
- *Lateral* navigation: Navigating from one sibling to another sibling (at the same level).

You can use the main `Activity` of the app as a parent screen, and then add an `Activity` or `Fragment` for each child screen.

Main Activity with an activity for each child

If the first-level child screen siblings have another level of child screens under them, you should implement each first-level screen as an activity, so that the lifecycle of each screen is managed properly before calling any second-level child screens.

For example, in the figure above, the parent screen is most likely the main activity. An app's main activity (usually `MainActivity.java`) is typically the parent screen for all other screens in your app. You implement a navigation pattern in the main activity to enable the user to go to another activity or fragment. For example, you can implement navigation using an `Intent` that starts an `Activity`.
Tip: Using an `Intent` in the current activity to start another activity adds the current activity to the call stack, so that the Back button in the other activity (described in the previous section) returns the user to the current activity.
As you've learned, the Android system initiates code in an `Activity` with callback methods that manage the `Act`activity`ivity` lifecycle for you. (A previous lesson covers the activity lifecycle; for more information, see Activities in the Android developer documentation.)
The declaration of each child activity is defined in the `AndroidManifest.xml` file with its parent activity. For example, the following defines `OrderActivity` as a child of the parent `MainActivity`:

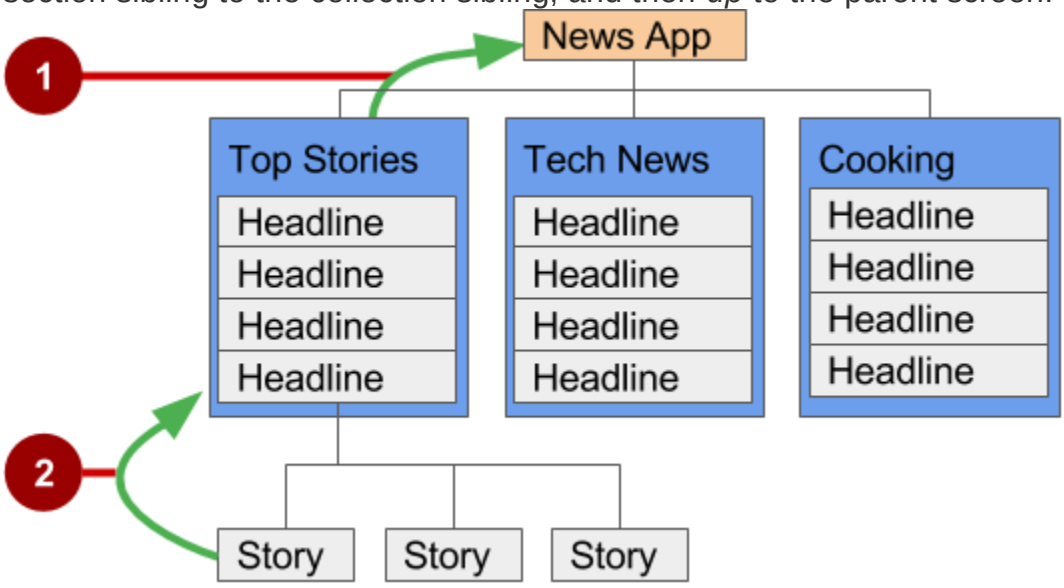
```
<activity android:name=".OrderActivity"
   android:label="@string/title_activity_order"
   android:parentActivityName=
                       "com.example.android.droidcafe.MainActivity">
   <meta-data
      android:name="android.support.PARENT_ACTIVITY"
      android:value=".MainActivity"/>
</activity>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

Main Activity with a Fragment for each child

If the child screen siblings do *not* have another level of child screens under them, you can define each one as a `Fragment`, which represents a behavior or portion of a UI within in an activity. Think of a fragment as a modular section of an activity which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running.
You can add more than one fragment in a single activity. For example, in a section sibling screen showing a news story and implemented as an `Activity`, you might have a child screen for a video clip implemented as a `Fragment`. You would implement a way for the user to navigate to the video clip `Fragment`, and then back to the `Activity` that shows the story.

Ancestral navigation (the Up button)

With ancestral navigation in a multitier hierarchy, you enable the user to go *up* from a section sibling to the collection sibling, and then *up* to the parent screen.



In the figure above:

1. **Up** button for ancestral navigation from the first-level siblings to the parent.
2. **Up** button for ancestral navigation from second-level siblings to the first-level child screen acting as a parent screen.

The **Up** button is used to navigate within an app based on the hierarchical relationships between screens. For example (referring to the figure above):

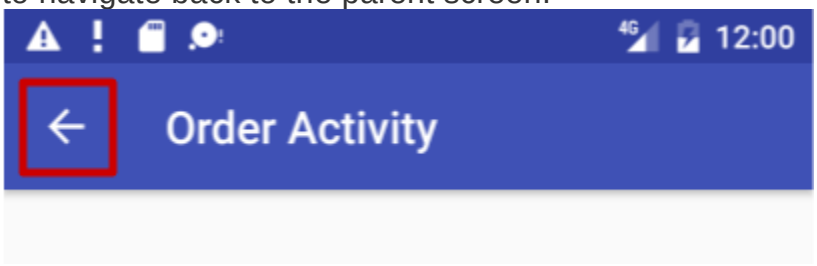
- If a first-level child screen offers headlines to navigate to second-level child screens, the second-level child screen siblings should offer **Up** buttons that return to the first-level child screen, which is their shared *parent*.
- If the parent screen offers navigation to first-level child siblings, then the first-level child siblings should offer an **Up** button that returns to the parent screen.
- If the parent screen is the topmost screen in an app (that is, the app's home screen), it should not offer an **Up** button.

Tip: The Back button below the screen differs from the **Up** button. The Back button provides navigation to whatever screen you viewed previously. If you have several children screens that the user can navigate through using a lateral navigation pattern (as described later in this chapter), the Back button would send the user back to the previous child screen, not to the parent screen. Use an **Up** button if you want to provide ancestral navigation from a child screen back to the parent screen. For more information about Up navigation, see Providing Up Navigation. See the concept chapter on menus and pickers for details on how to implement the app bar.

To provide the **Up** button for a child screen `Activity`, declare the parent of the `Activity` to be the main `Activity` in the `AndroidManifest.xml` file:

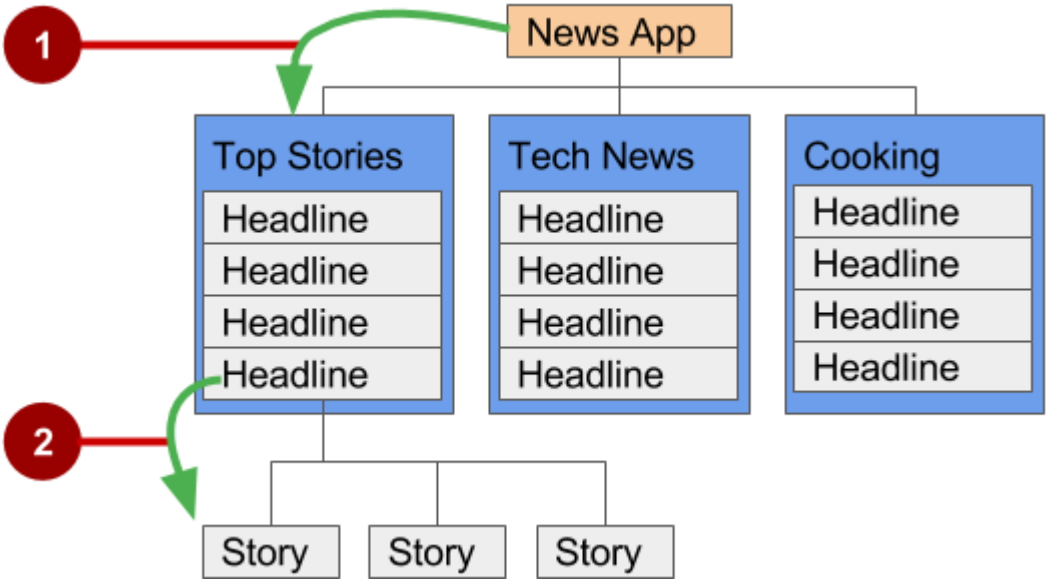
```
<activity android:name="com.example.android.droidcafeinput.OrderActivity"
    android:label="Order Activity"
    android:parentActivityName=".MainActivity">
    <meta-data android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>
</activity>
```

The snippet above in `AndroidManifest.xml` declares the parent for the child screen `OrderActivity` to be `MainActivity`. It also sets the `android:label` to a title for the `Activity` screen to be "Order Activity". The child screen now includes the **Up** button in the app bar (highlighted in the figure below), which the user can tap to navigate back to the parent screen.



Descendant navigation

With descendant navigation, you enable the user to go from the parent screen to a first-level child screen, and from a first-level child screen down to a second-level child screen.



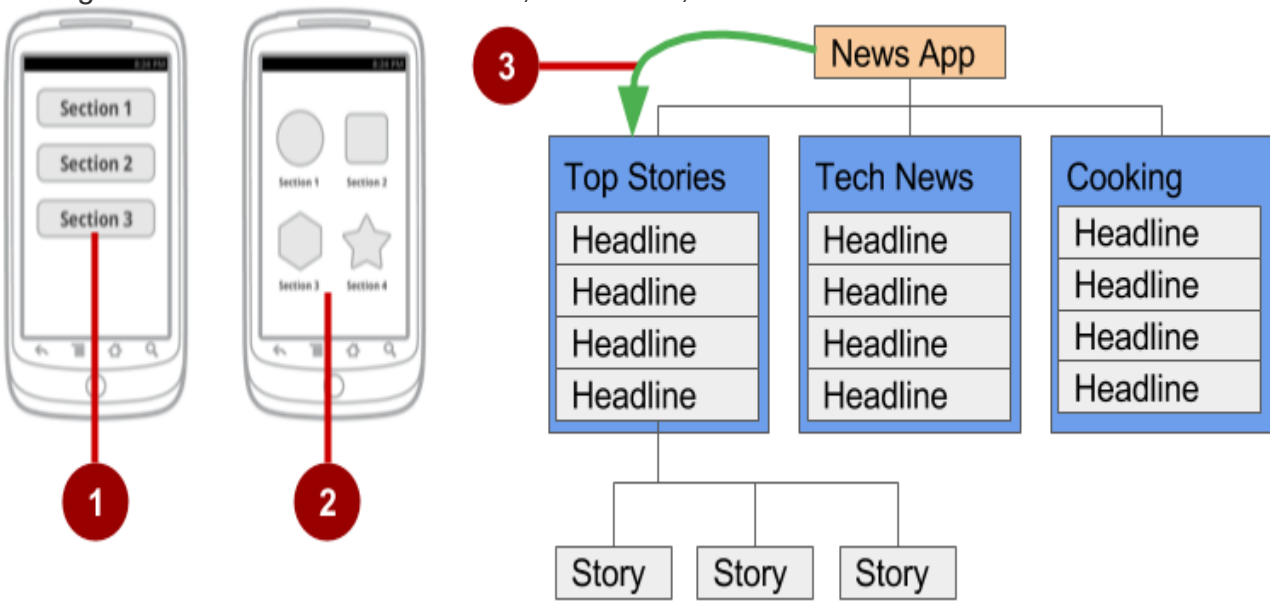
In the figure above:

1. Descendant navigation from parent to first-level child screen
2. Descendant navigation from headline in a first-level child screen to a second-level child screen

Buttons or targets

The best practice for descendant navigation from the parent screen to collection siblings is to use buttons or simple *targets* such as an arrangement of images or iconic buttons (also known as a *dashboard*). When the user touches a button, the collection sibling screen opens, replacing the current context (screen) entirely.

Tip: Buttons and simple targets are rarely used for navigating to section siblings *within* a collection. See lists, carousels, and cards in the next section.



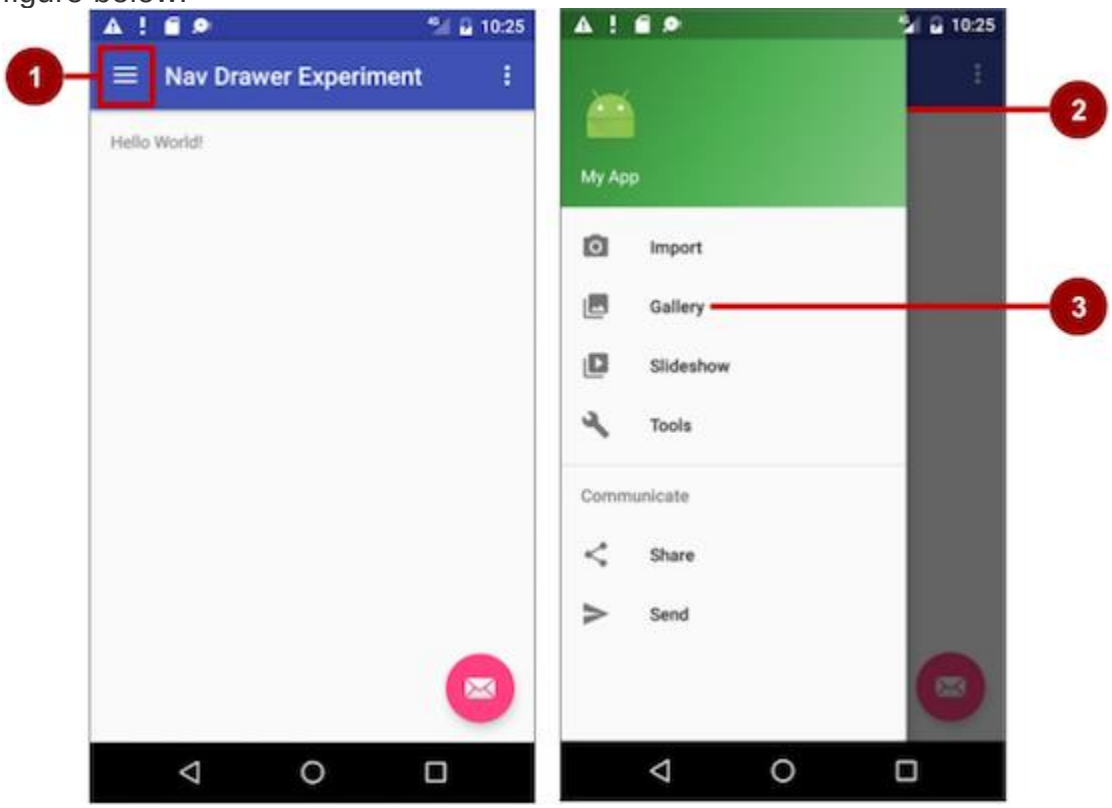
In the figure above:

1. Buttons on a parent screen
2. Targets (Image buttons or icons) on a parent screen
3. Descendant navigation pattern from parent screen to first-level child siblings

A dashboard usually has either two or three rows and columns, with large touch targets to make it easy to use. Dashboards are best when each collection sibling is equally important. You can use a `LinearLayout`, `RelativeLayout`, or `GridLayout`. See Layouts for an overview of how layouts work.

Navigation drawer

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

A good example of a navigation drawer is in the Gmail app, which provides access to the inbox, labeled email folders, and settings. The best practice for employing a navigation drawer is to provide descendant navigation from the parent `Activity` to all of the other child screens in an app. It can display many navigation targets at once—for example, it can contain buttons (like a dashboard), tabs, or a list of items (like the Gmail drawer).

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 includes room for a navigation icon button
- A content layout for the `Activity` that displays the navigation drawer
- A layout for the navigation drawer header

Follow these general steps:

1. Populate the navigation drawer menu with item titles and icons.
2. Set up the navigation drawer and item listeners in the `Activity` code.
3. Handle the navigation menu item selections.

Creating the navigation drawer layout

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 `ViewGroup` 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.
For example, the following layout uses:

- A `DrawerLayout` as the root of the `Activity` layout in `activity_main.xml`.
- The main content of screen defined in the `app_bar_main.xml` layout file.
- A `NavigationView` that represents a standard navigation menu that can be populated by a menu resource XML file.

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Refer to the figure below that corresponds to this layout:

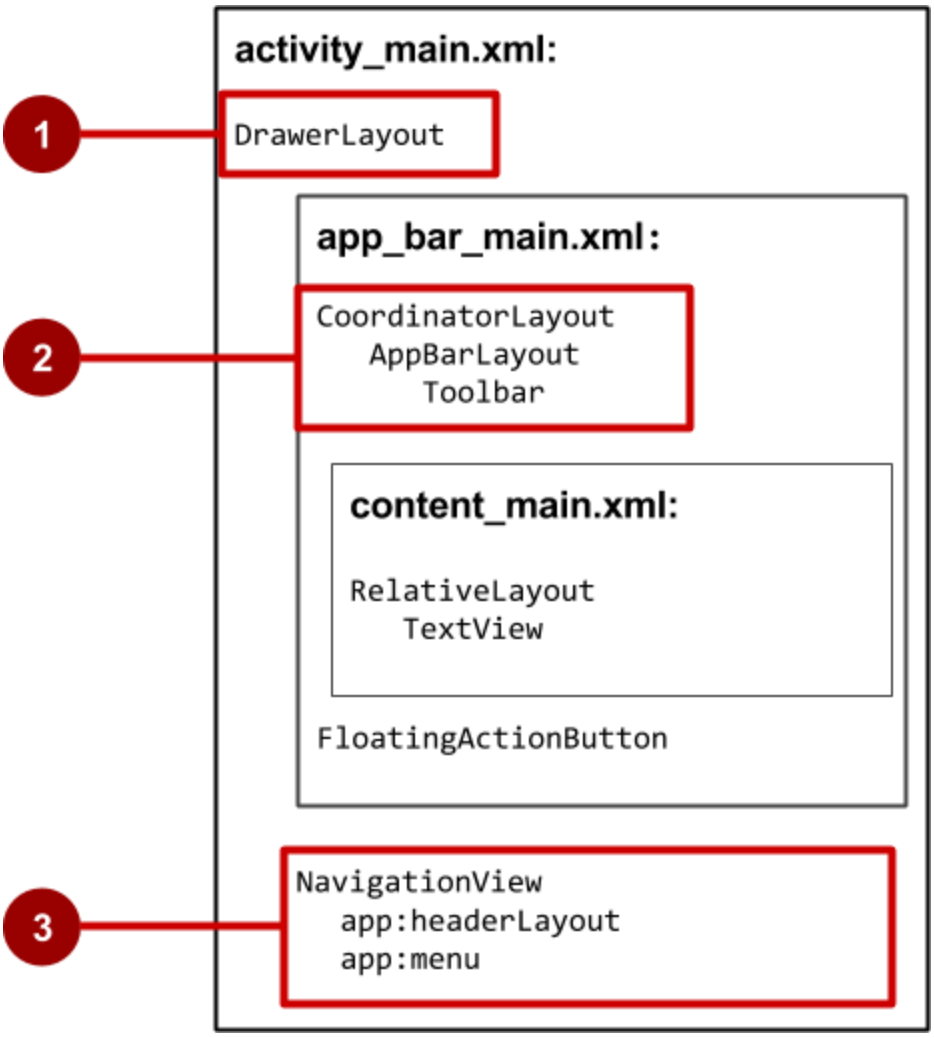
```
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <include
        layout="@layout/app_bar_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main"
        app:menu="@menu/activity_main_drawer" />

</android.support.v4.widget.DrawerLayout>
```

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/



In the figure above:

1. `DrawerLayout` is the root `ViewGroup` 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.

Note the following in the `activity_main.xml` layout:

- The `android:id` for the `DrawerLayout` is `drawer_layout`. You will use this `id` to instantiate a `drawer` object in your code.
- The `android:id` for the `NavigationView` is `nav_view`. You will use this `id` to instantiate a `navigationView` object in your code.
- The `NavigationView` must specify its horizontal gravity with the `android:layout_gravity` attribute. Use the `"start"` value for this attribute (rather than `"left"`), so that if the app is used with right-to-left (RTF) languages, the drawer appears on the right rather than the left side.

```
android:layout_gravity="start"
```

- Use the `android:fitsSystemWindows="true"` attribute to set the padding of the `DrawerLayout` and the `NavigationView` to ensure the contents don't overlay the system windows. `DrawerLayout` uses `fitsSystemWindows` as a sign that it needs to inset its children (such as the main content `ViewGroup`), but still draw the top status bar background in that space. As a result, the navigation drawer appears to be overlapping, but not obscuring, the translucent top status bar. The insets you get from `fitsSystemWindows` will be correct on all platform versions to ensure that your content does not overlap with system-provided UI components.

The navigation drawer header

The `NavigationView` specifies the layout for the *header* of the navigation drawer with the attribute `app:headerLayout="@layout/nav_header_main"`. The `nav_header_main.xml` file defines the layout of this header to include an `ImageView` and a `TextView`, which is typical for a navigation drawer, but you could also include other `View` elements.
Tip: The header's height should be 160dp, which you should extract into a dimension resource (`nav_header_height`).
The following is the code for the `nav_header_main.xml` file:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="@dimen/nav_header_height"
    android:background="@drawable/side_nav_bar"
    android:gravity="bottom"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:theme="@style/ThemeOverlay.AppCompat.Dark">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/nav_header_vertical_spacing"
        android:src="@android:drawable/sym_def_app_icon" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="@dimen/nav_header_vertical_spacing"
        android:text="@string/my_app_title"
        android:textAppearance="@style/TextAppearance.AppCompat.Body1" />

</LinearLayout>
```

The app bar layout

The `include` tag in the `activity_main.xml` layout file includes the `app_bar_main.xml` layout file, which uses a `CoordinatorLayout` as its root. The `app_bar_main.xml` file defines the app bar layout with the `Toolbar` class as shown previously in the chapter about menus and pickers. It also defines a floating action button, and uses an `include` tag to include the `content_main.xml` layout.
The following is the code for the `app_bar_main.xml` file:

```
<android.support.design.widget.CoordinatorLayout
    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"
    android:fitsSystemWindows="true"
    tools:context="com.example.android.navigationexperiments.MainActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        android:src="@android:drawable/ic_dialog_email" />

</android.support.design.widget.CoordinatorLayout>
```

Note the following:

- The `app_bar_main.xml` layout uses a `CoordinatorLayout` as its root, and includes the `content_main.xml` layout.
- The `app_bar_main.xml` layout uses the `android:fitsSystemWindows="true"` attribute to set the padding of the app bar to ensure that it doesn't overlay the system windows such as the status bar.

The content layout for the main activity screen

The layout above uses an `include` tag to include the `content_main.xml` layout, which defines the layout of the main `Activity` screen. In the example layout below, the main `Activity` screen shows a `TextView` that displays the string "Hello World!". The following is the code for the `content_main.xml` file:

```
<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:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.example.android.navigationexperiments.MainActivity"
    tools:showIn="@layout/app_bar_main">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</RelativeLayout>
```

Note the following:

- The `content_main.xml` layout must be the first child in the `DrawerLayout` because the drawer must be on top of the content. In our layout above, the `content_main.xm` layout is included in the `app_bar_main.xml` layout, which is the first child.
- The `content_main.xml` layout uses a `RelativeLayout` `ViewGroup` set to match the parent view's width and height, because it represents the entire UI when the navigation drawer is hidden.
- The layout *behavior* for the `RelativeLayout` is set to the string resource `@string/appbar_scrolling_view_behavior`, which controls the scrolling behavior of the screen in relation to the app bar at the top.
  The `AppBarLayout.ScrollingViewBehavior` class defines this behavior. View elements that scroll vertically should use this behavior, because it supports nested scrolling to automatically scroll any `AppBarLayout` siblings.

Populating the navigation drawer menu

The `NavigationView` in the `activity_main.xml` layout specifies the menu items for the navigation drawer using the following statement:

```
app:menu="@menu/activity_main_drawer"
```

The menu items are defined in the `activity_main_drawer.xml` file, which is located under **app > res > menu** in the **Project > Android** pane. The `<group></group>` tag defines a *menu group*—a collection of items that share traits, such as whether they are visible, enabled, or checkable. A group must contain one or more `<item></>` elements and be a child of a `<menu>` element, as shown below. In addition to defining each menu item's title with the `android:title` attribute, the file also defines each menu item's icon with the `android:icon` attribute.

The group is defined with the `android:checkableBehavior` attribute. This attribute lets you put interactive elements within the navigation drawer, such as toggle switches that can be turned on or off, and checkboxes and radio buttons that can be selected. The choices for this attribute are:

- `single`: Only one item from the group can be selected. Use for radio buttons.
- `all`: All items can be selected. Use for checkboxes.
- `none`: No items can be selected.

The following XML code snippet shows how to define a menu group:

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

    <group android:checkableBehavior="none">
        <item
            android:id="@+id/nav_camera"
            android:icon="@drawable/ic_menu_camera"
            android:title="@string/import_camera" />
        <item
            android:id="@+id/nav_gallery"
            android:icon="@drawable/ic_menu_gallery"
            android:title="@string/gallery" />
        <item
            android:id="@+id/nav_slideshow"
            android:icon="@drawable/ic_menu_slideshow"
            android:title="@string/slideshow" />
        <item
            android:id="@+id/nav_manage"
            android:icon="@drawable/ic_menu_manage"
            android:title="@string/tools" />
    </group>

    <item android:title="@string/communicate">
        <menu>
            <item
                android:id="@+id/nav_share"
                android:icon="@drawable/ic_menu_share"
                android:title="@string/share" />
            <item
```

```
                android:id="@+id/nav_send"
                android:icon="@drawable/ic_menu_send"
                android:title="@string/send" />
        </menu>
    </item>

</menu>
```

Setting up the navigation drawer and item listeners

To use a listener for the navigation drawer's menu items, the `Activity` hosting the navigation drawer must implement the `OnNavigationItemSelectedListener` interface:

1. Implement `NavigationView.OnNavigationItemSelectedListener` in the class definition:

```
2.      public class MainActivity extends AppCompatActivity implements
3.                  NavigationView.OnNavigationItemSelectedListener {
```

This interface offers the `onNavigationItemSelected()` method, which is called when an item in the navigation drawer menu item is tapped. As you enter `OnNavigationItemSelectedListener`, the red light bulb appears on the left margin.

4. Click the light bulb, choose **Implement methods**, and choose the **onNavigationItemSelected(item:MenuItem):boolean** method.

Android Studio adds a stub for the method:

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
    return false;
}
```

You learn how to use this stub in the next section.

5. Before setting up the navigation item listener, add code to the `onCreate()` method for the `Activity` to instantiate the `DrawerLayout` and `NavigationView` objects (`drawer` and `navigationView` in the code below):

```
6.   @Override
7.   protected void onCreate(Bundle savedInstanceState) {
8.      // ... Rest of onCreate code.
9.      DrawerLayout drawer = (DrawerLayout)
10.                          findViewById(R.id.drawer_layout);
11.     ActionBarDrawerToggle toggle =
12.                 new ActionBarDrawerToggle(this, drawer, toolbar,
13.                 R.string.navigation_drawer_open,
14.                 R.string.navigation_drawer_close);
15.     if (drawer != null) {
16.        drawer.addDrawerListener(toggle);
17.     }
18.     toggle.syncState();
19.
20.     NavigationView navigationView = (NavigationView)
21.                          findViewById(R.id.nav_view);
22.     if (navigationView != null) {
23.        navigationView.setNavigationItemSelectedListener(this);
24.     }
25.  }
```

The code above instantiates an `ActionBarDrawerToggle`, which substitutes a special drawable for the **Up** button in the app bar, and links the `Activity` to the `DrawerLayout`. The special drawable appears as a "hamburger" navigation icon when the drawer is closed, and animates into an arrow as the drawer opens.
Note: Be sure to use the `ActionBarDrawerToggle` in `support-library-v7.appcompact`, *not* the version in `support-library-v4`.

Tip: You can customize the animated toggle by defining the `drawerArrowStyle` in your `ActionBar` theme. For more detailed information about the `ActionBar` theme, see Adding the App Bar in the Android Developer documentation.
The code above implements `addDrawerListener()` to listen for drawer open and close events, so that when the user taps custom drawable button, the navigation drawer slides out.
You must also use the `syncState()` method of `ActionBarDrawerToggle` to synchronize the state of the drawer indicator. The synchronization must occur after the `DrawerLayout` instance state has been restored, and any other time when the state may have diverged in such a way that the `ActionBarDrawerToggle` was not notified.
The code above ends by setting a listener, `setNavigationItemSelectedListener()`, to the navigation drawer to listen for item clicks.
The `ActionBarDrawerToggle` also lets you specify the strings to use to describe the open/close drawer actions for accessibility services. Define the strings in your `strings.xml` file:

```
<string name="navigation_drawer_open">Open navigation drawer</string>
<string name="navigation_drawer_close">Close navigation drawer</string>
```

Handling navigation menu item selections

Add code to the `onNavigationItemSelected()` method stub to handle menu item selections. This method is called when an item in the navigation drawer menu is tapped. You can use `switch case` statements to take the appropriate action based on the menu item's `id`, which you can retrieve using the `getItemId()` method:

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
   DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
   // Handle navigation view item clicks here.
   switch (item.getItemId()) {
      case R.id.nav_camera:
         // Handle the camera import action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_camera));
         return true;
      case R.id.nav_gallery:
         // Handle the gallery action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_gallery));
         return true;
      case R.id.nav_slideshow:
         // Handle the slideshow action (for now display a toast).
         drawer.closeDrawer(GravityCompat.START);
         displayToast(getString(R.string.chose_slideshow));
```

```
            return true;
        case R.id.nav_manage:
            // Handle the tools action (for now display a toast).
            drawer.closeDrawer(GravityCompat.START);
            displayToast(getString(R.string.chose_tools));
            return true;
        case R.id.nav_share:
            // Handle the share action (for now display a toast).
            drawer.closeDrawer(GravityCompat.START);
            displayToast(getString(R.string.chose_share));
            return true;
        case R.id.nav_send:
            // Handle the send action (for now display a toast).
            drawer.closeDrawer(GravityCompat.START);
            displayToast(getString(R.string.chose_send));
            return true;
        default:
            return false;
    }
}
```

After the user taps a navigation drawer selection or taps outside the drawer, the `DrawerLayout` `closeDrawer()` method closes the drawer.

Lists and carousels

Use a scrolling list, such as a `RecyclerView`, to provide navigation targets for descendant navigation. Vertically scrolling lists are often used for a screen that lists stories, with each list item acting as a button to each story. For more visual or media-rich content items such as photos or videos, you may want to use a horizontally scrolling list (also known as a *carousel*). These UI elements are good for presenting items in a collection (for example, a list of news stories).
You learn about `RecyclerView` in another chapter.

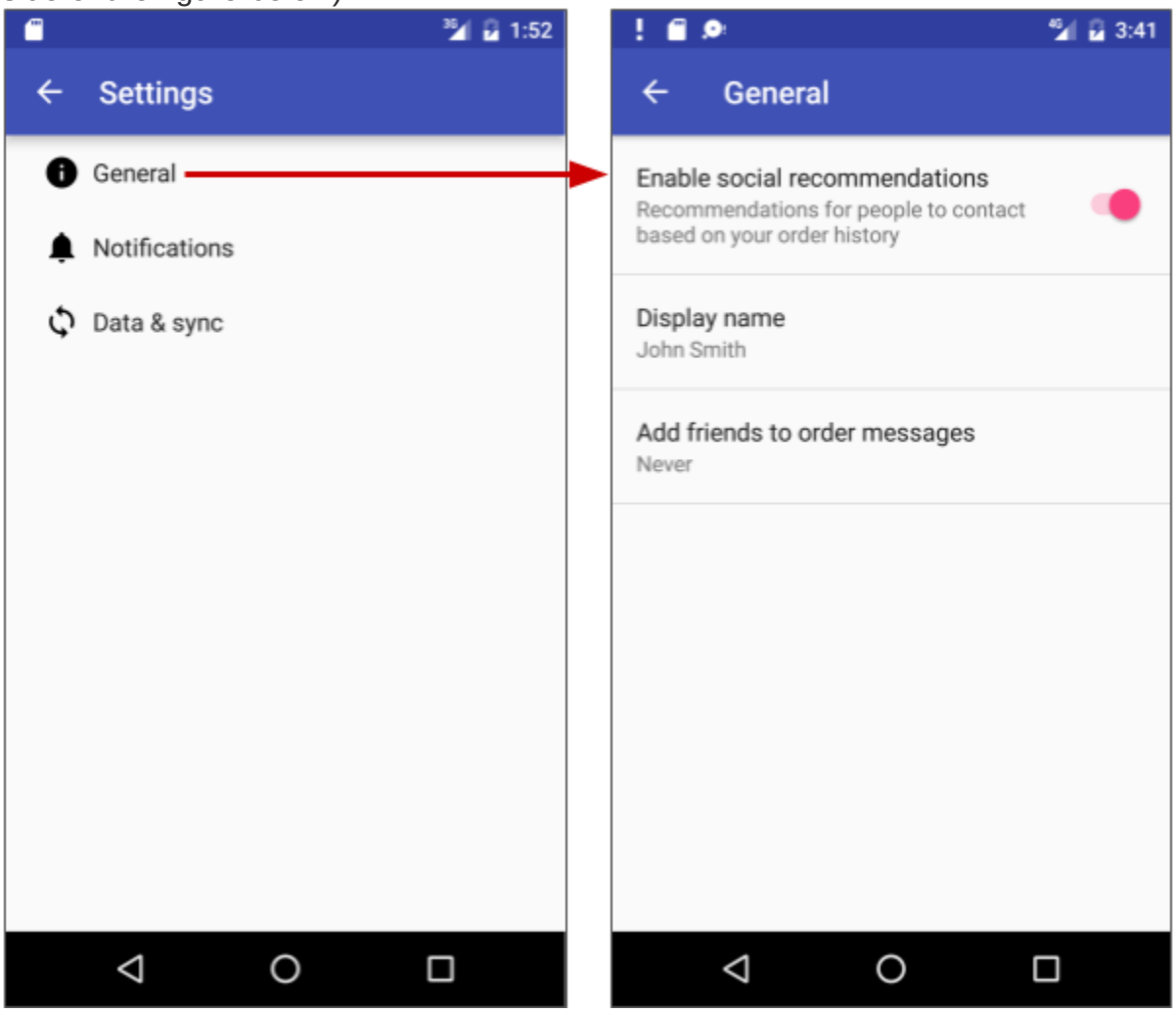
Master/detail navigation flow

In a master/detail navigation flow, a master screen contains a list of items, and a detail screen shows detailed information about one item. You usually implement descendant navigation using one of the following techniques:

- Use an intent to starts an activity that represents the detail screen. For more information about intents, see Intents and Intent Filters in the Android developer documentation.
- When adding a Settings `Activity`, extend `PreferenceActivity` to create a two-pane master/detail layout to support large screens. Replace the activity content with a Settings `Fragment`. This is a useful pattern if you have multiple groups of settings and need to support tablet-sized screens as well as smartphones. You learn about the Settings activity and `PreferenceActivity` in another chapter. For more information about using fragments, see Fragments in the Android developer documentation.

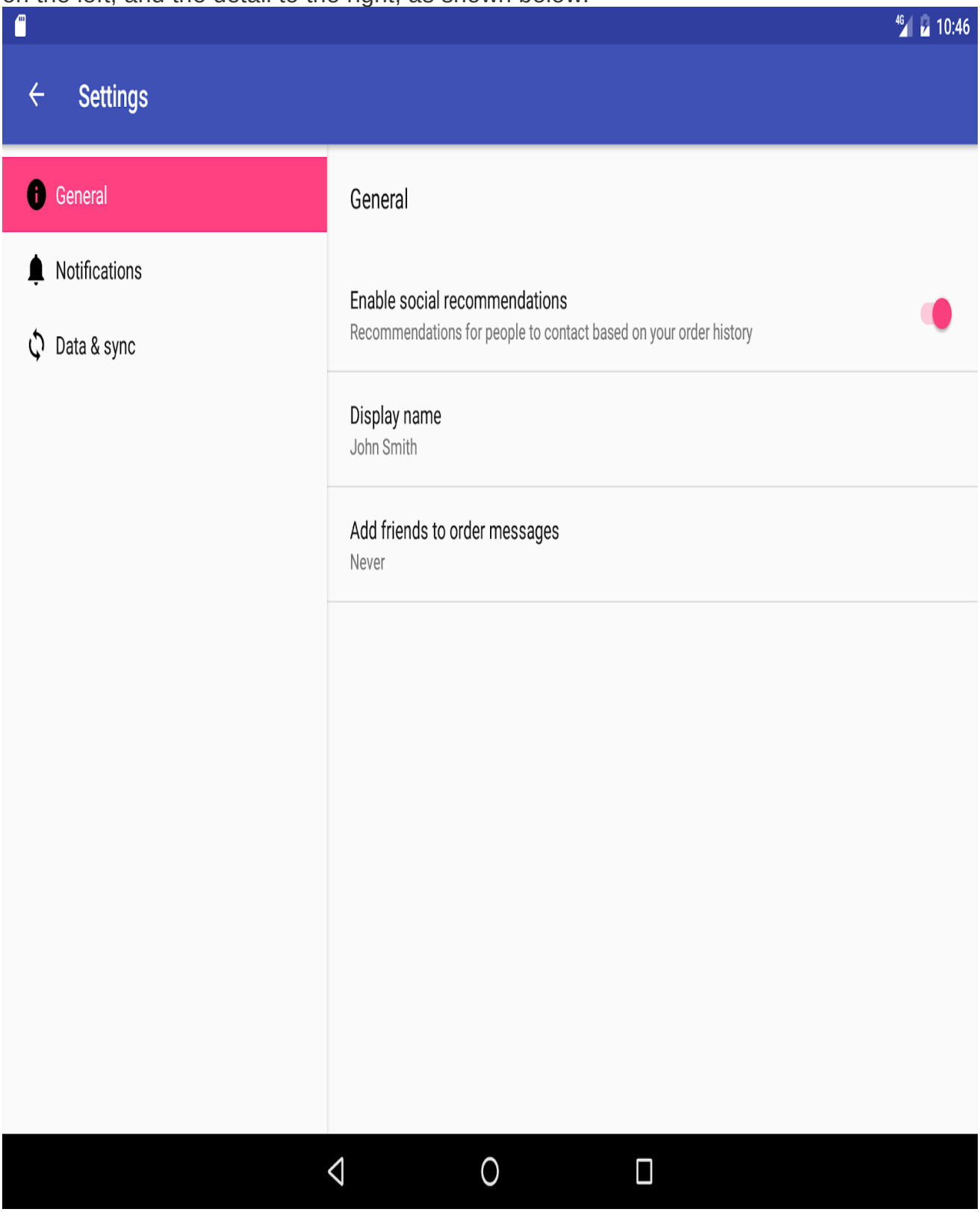
https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Smartphones are best suited for displaying one screen at a time—for example a master screen (on the left side of the figure below) and a detail screen (on the right side of the figure below).



On the other hand, tablet displays, especially when viewed in the landscape orientation, are best suited for showing multiple content panes at a time: the master

on the left, and the detail to the right, as shown below.

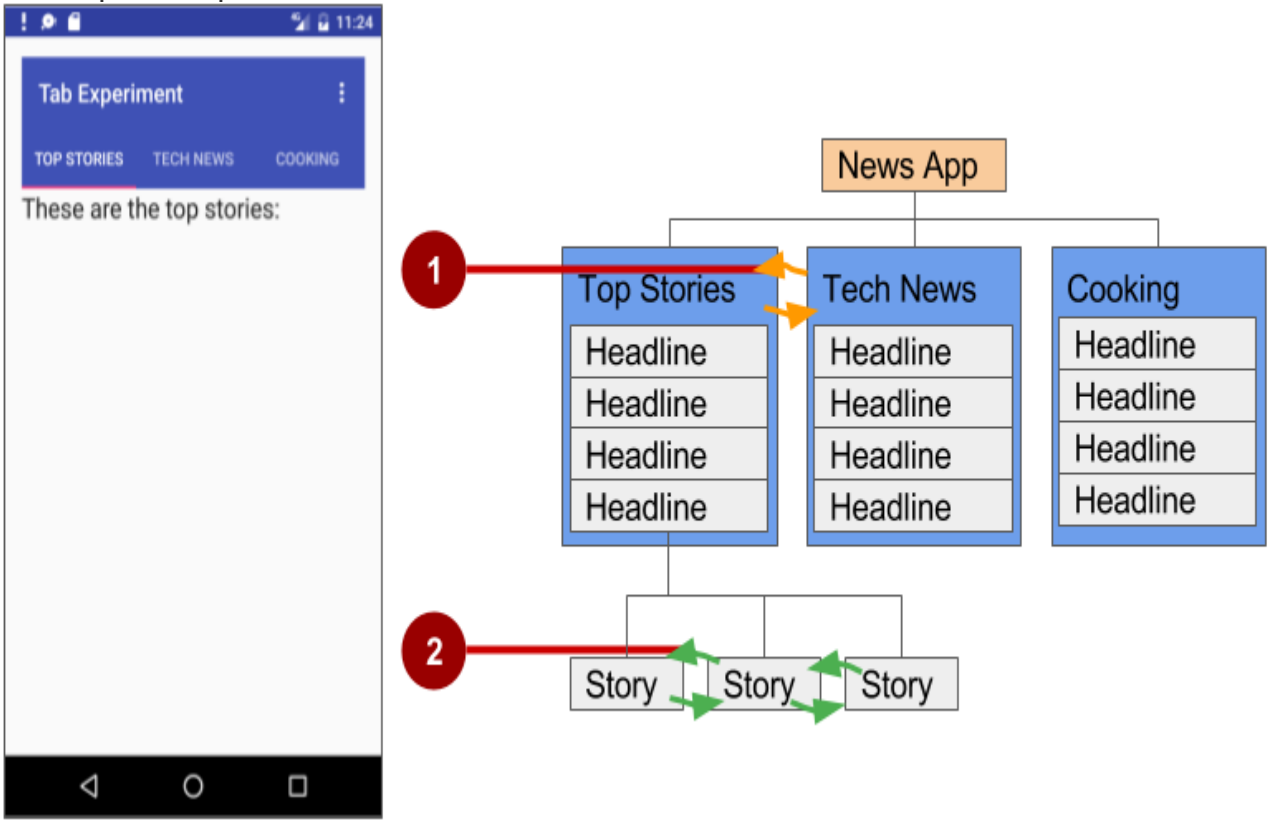


Options menu in the app bar

The app bar typically contains the options menu, which is most often used for navigation patterns for descendant navigation. It may also contain an **Up** button for ancestral navigation, a nav icon for opening a navigation drawer, and a filter icon to filter page views. You learn how to set up the options menu and the app bar in another chapter.

Lateral navigation with tabs and swipes

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, or from one top story to the next, without having to navigate back up to the parent screen.



In the figure above:

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

Another example of lateral navigation is the ability to swipe left or right in a Gmail conversation to view a newer or older email in the same inbox.

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, providing navigation to other screens. Tab navigation is a common solution for lateral navigation from one child screen to another child screen that is a *sibling*—in the same position in the hierarchy and sharing the same parent screen.

Tabs are most appropriate for small sets (four or fewer) of sibling screens. You can combine tabs with swipe views, so that the user can swipe across from one screen to another as well as tap a tab.

Tabs offer two benefits:

- Because there is a single, initially selected tab, users already have access to that tab's content from the parent screen without any further navigation.
- Users can navigate quickly between related screens, without needing to first revisit the parent.

Keep in mind the following best practices when using tabs:

- Tabs are usually laid out horizontally.
- Tabs should always run along the top of the screen, and should not be aligned to the bottom of the screen.
- Tabs should be persistent across related screens. Only the designated content region should change when tapping a tab, and tab indicators should remain available at all times.
- Switching to another tab should not be treated as history. For example, if a user switches from tab A to tab B, pressing the **Up** button in the app bar should not reselect tab A but should instead return the user to the parent screen.

The key steps for implementing tabs are as follows:

1. Define the tab layout. The main class used for displaying tabs is `TabLayout`. It provides a horizontal layout to display tabs. You can show the tabs below the app bar.
2. Implement a `Fragment` for each tab content screen. A `Fragment` 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. One benefit of using a `Fragment` for each tabbed content is that you can isolate the code for managing the tabbed content inside the `Fragment`. To learn about `Fragment`, see Fragments in the API Guide.
3. Add a pager adapter. Use the `PagerAdapter` class to populate "pages" (screens) inside of a `ViewPager`, which is a layout manager that lets the user flip left and right through screens of data. You supply an implementation of a `PagerAdapter` to generate the screens that the `View` shows. `ViewPager` is most often used in conjunction with `Fragment`, which is a convenient way to supply and manage the lifecycle of each screen.
4. Create an instance of the tab layout, and set the text for each tab.
5. Use `PagerAdapter` to manage screens ("pages"). Each screen is represented by its own `Fragment`.
6. Set a listener to determine which tab is tapped.

There are standard adapters for using a `Fragment` 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 another screen, minimizing memory usage.

Defining tab layout

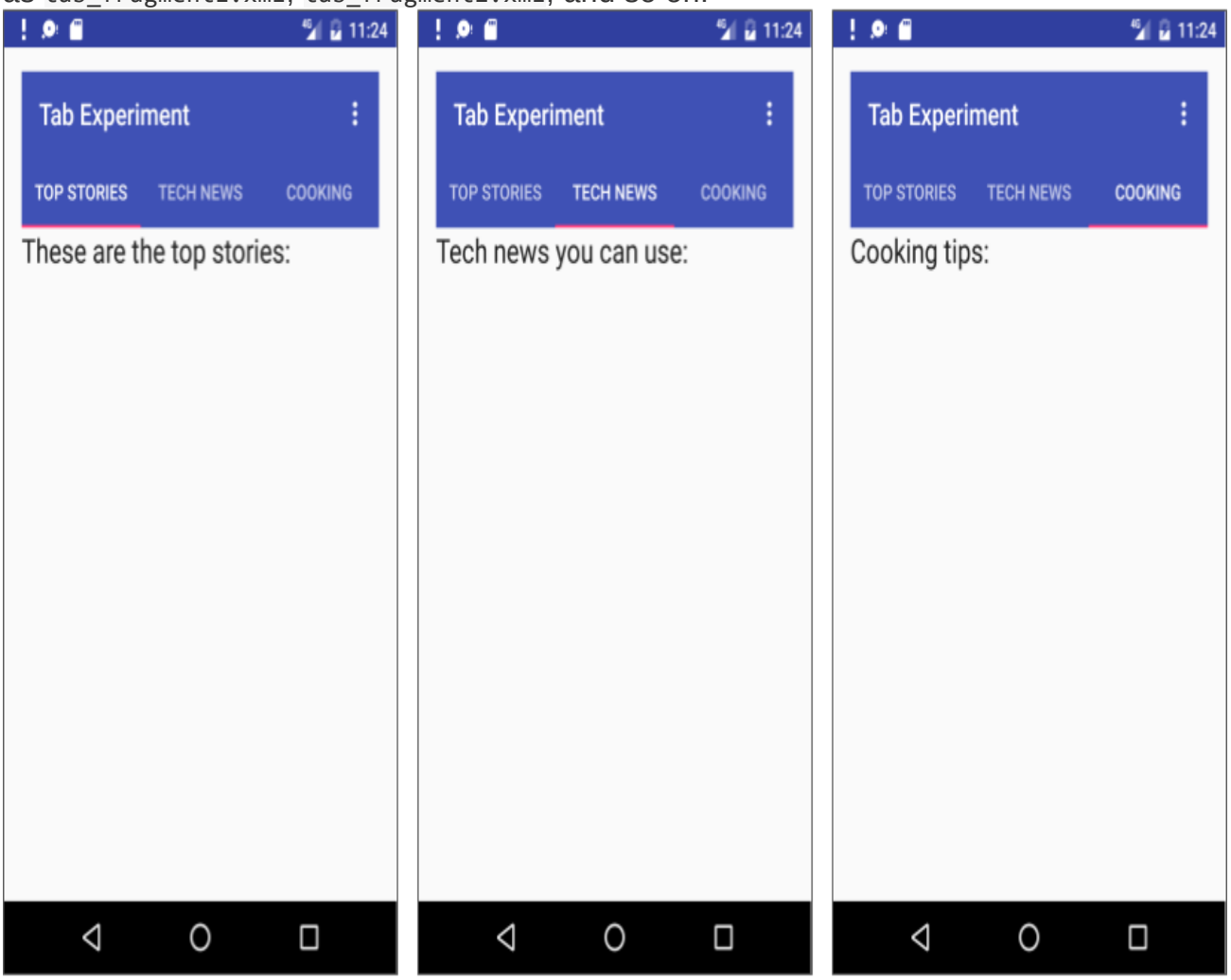
To use a `TabLayout`, you can design the main `Activity` layout to use a `Toolbar` for the app bar, a `TabLayout` for the tabs below the app bar, and a `ViewPager` within the root layout to switch child elements. The layout should look similar to the following, assuming each child element fills the screen:

```
<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"/>
```

For each child view, create a layout for each `Fragment` such as `tab_fragment1.xml`, `tab_fragment2.xml`, and so on.



Implementing each fragment

A `Fragment` 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. To learn about `Fragment`, see Fragments in the API Guide.

Add a class for each `Fragment` (such as `TabFragment1.java`, `TabFragment2.java`, and `TabFragment3.java`) representing each screen the user can visit by clicking a tab.

Each class should extend `Fragment` and inflate the layout associated with the screen (`tab_fragment1.xml`, `tab_fragment2.xml`, and `tab_fragment3.xml`). For example, `TabFragment1.java` looks like this:

```
public class TabFragment1 extends Fragment {

   @Override
   public View onCreateView(LayoutInflater inflater,
                            ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.tab_fragment1, container, false);
   }
}
```

Adding a pager adapter

Add a `PagerAdapter` that extends `FragmentStatePagerAdapter`. The code should do the following:

1. Define the number of tabs.
2. Use the `getItem()` method of the `Adapter` class to determine which tab is clicked.
3. Use a `switch case` block to return the screen (page) to show based on which tab is clicked.

The following is an example:

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

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

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

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

Creating an instance of the tab layout

In the `onCreate()` method of the main `Activity`, create an instance of the tab layout from the `tab_layout` element in the layout, and set the text for each tab using `addTab()`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   // ... Rest of onCreate code
   // Create an instance of the tab layout from the view.
   TabLayout tabLayout = (TabLayout) findViewById(R.id.tab_layout);
   // Set the text for each tab.
   tabLayout.addTab(tabLayout.newTab().setText("Top Stories"));
   tabLayout.addTab(tabLayout.newTab().setText("Tech News"));
   tabLayout.addTab(tabLayout.newTab().setText("Cooking"));
   // Set the tabs to fill the entire layout.
   tabLayout.setTabGravity(TabLayout.GRAVITY_FILL);
   // Use PagerAdapter to manage page views in fragments.
}
```

Managing screen views in fragments with a listener

Use `PagerAdapter` in the `onCreate()` method of the main `Activity` to manage screen ("page") views in each `Fragment`. Each screen is represented by its own `Fragment`. You also need to set a listener to determine which tab is tapped. The following code should appear after the code from the previous section in the `onCreate()` method:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   // ... Rest of onCreate code
   // Use PagerAdapter to manage page views in fragments.
   final ViewPager viewPager = (ViewPager) findViewById(R.id.pager);
   final PagerAdapter adapter = new PagerAdapter
               (getSupportFragmentManager(), tabLayout.getTabCount());
   viewPager.setAdapter(adapter);
   // Setting a listener for clicks.
   viewPager.addOnPageChangeListener(new
               TabLayout.TabLayoutOnPageChangeListener(tabLayout));
   tabLayout.addOnTabSelectedListener(new
                                  TabLayout.OnTabSelectedListener() {
      @Override
      public void onTabSelected(TabLayout.Tab tab) {
         viewPager.setCurrentItem(tab.getPosition());
      }

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

      }

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

Using ViewPager for swipe views (horizontal paging)

`ViewPager` is a layout manager that lets the user flip left and right through "pages" (screens) of content. `ViewPager` is most often used in conjunction with `Fragment`, which is a convenient way to supply and manage the lifecycle of each "page". `ViewPager` also provides the ability to swipe "pages" horizontally.
In the previous example, you used a `ViewPager` within the root layout to switch child screens. This provides the ability for the user to swipe from one child screen to another. Users are able to navigate to sibling screens by touching and dragging the screen horizontally in the direction of the desired adjacent screen.
Swipe views are most appropriate where there is some similarity in content type among sibling pages, and when the number of siblings is relatively small. In these cases, this pattern can be used along with tabs above the content region to indicate the current page and available pages, to aid discoverability and provide more context to the user.

Tip: It's best to avoid horizontal paging when child screens contain horizontal panning surfaces (such as maps), as these conflicting interactions may deter your screen's usability.

Related practical

The related practical is 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

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

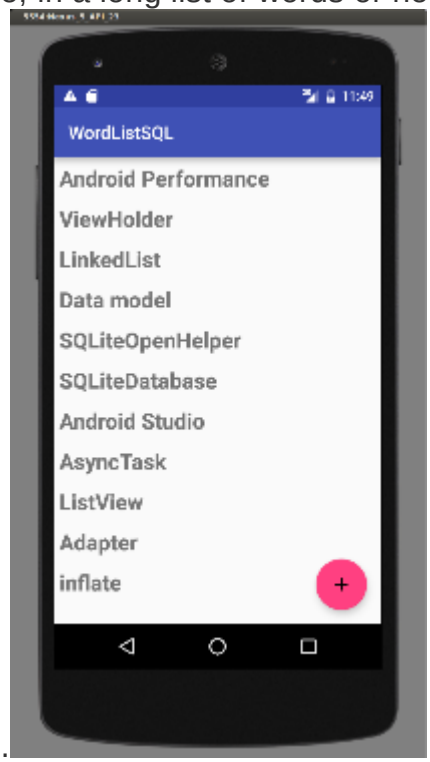
4.5: RecyclerView

Contents:

- RecyclerView components
- Implementing a RecyclerView
- Related practical
- Learn more

About RecyclerView

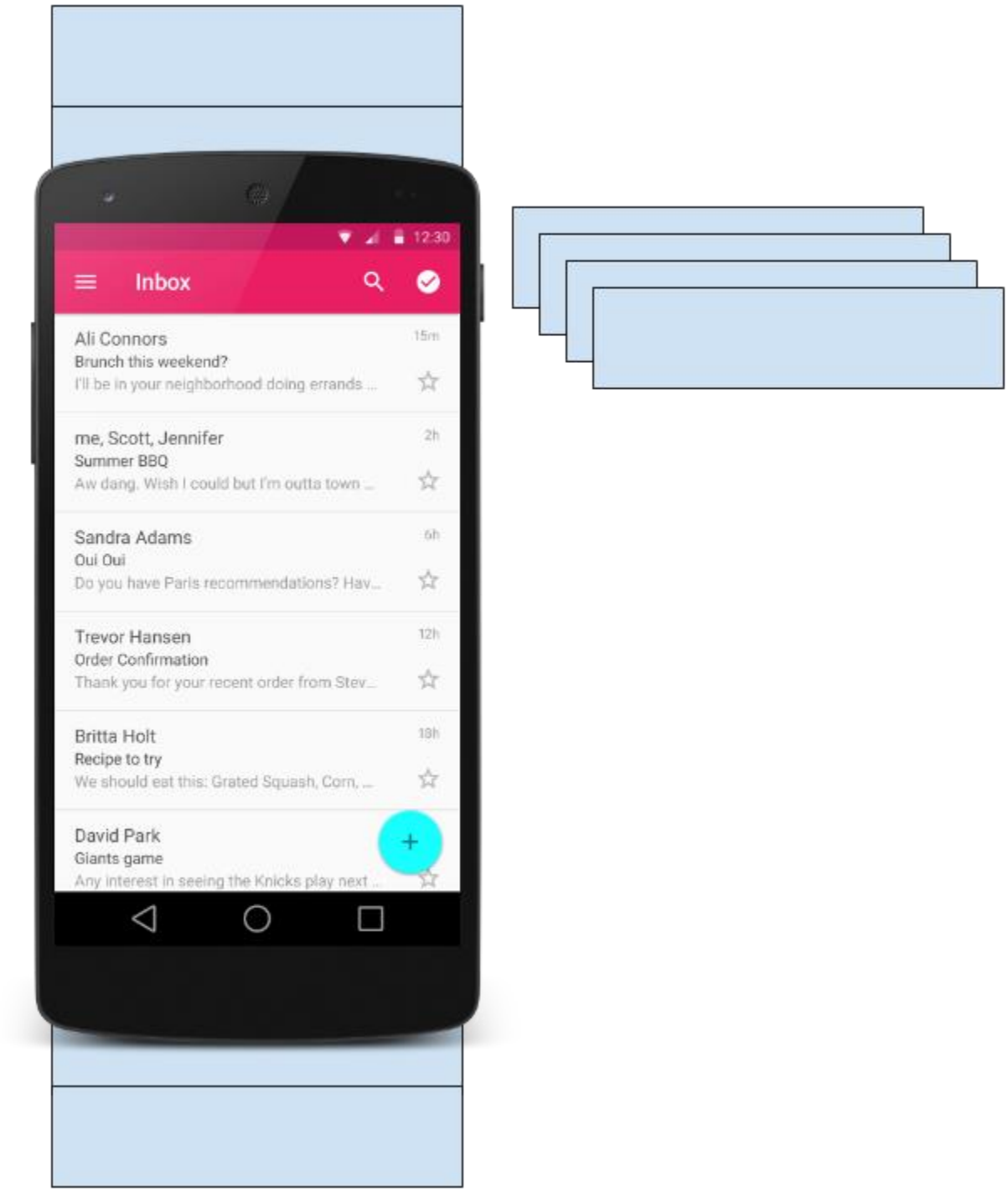
When you display a large number of items in a scrollable list, most of the items aren't visible. For example, in a long list of words or news headlines, the user only sees a few items at a time.



Or you may have a dataset that changes as the user interacts with it. If you create a new `View` every time the data changes, that's a lot of `View` items, even for a small dataset.
From a performance perspective, you want to conserve memory and save time:

- To conserve memory, minimize the number of `View` items that exist at any given point.
- To save time, minimize the number of `View` items you have to create.

To accomplish both these goals, create more `View` items than the user can see on the screen and cache the created `View` items. Then reuse the `View` items with different data as list items scroll in and out of the display.



The `RecyclerView` class is a more advanced and flexible version of `ListView`. It's a container for displaying large, scrollable data sets efficiently by maintaining a limited number of `View` items.

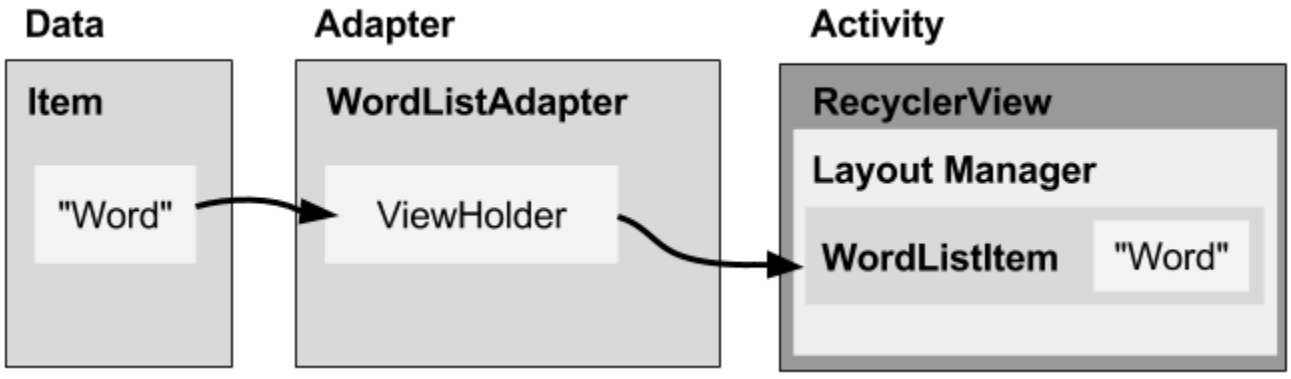
Use `RecyclerView` when you need to display a large amount of scrollable data, or data collections whose elements change at runtime based on user action or network events.

RecyclerView components

To display data in a `RecyclerView`, you need the following (refer to the figure below):

- **Data.** It doesn't matter where the data comes from. You can create the data locally, as you do in the practical, get it from a database on the device as you will do in a later practical, or pull it from the cloud.
- **A RecyclerView.** The scrolling list that contains the list items. An instance of `RecyclerView` as defined in the `Activity` layout file to act as the container for the `View` items.
- **Layout for one item of data.** All list items look the same, so you can use the same layout for all of them. The item layout has to be created separately from the `Activity` layout, so that one `View` item at a time can be created and filled with data.
- **A layout manager.** The layout manager handles the organization (layout) of user interface components in a `View`. Each `ViewGroup` has a layout manager. For `LinearLayout`, the Android system handles the layout for you. `RecyclerView` requires an explicit layout manager to manage the arrangement of list items contained within it. This layout could be vertical, horizontal, or a grid. The layout manager is an instance of `Recyclerview.LayoutManager` to organize the layout of the items in the `RecyclerView`.
- **An adapter.** Use an extension of `RecyclerView.Adapter` to connect your data to the `RecyclerView`. It prepares the data and how will be displayed in a `ViewHolder`. When the data changes, the adapter updates the contents of the respective list item view in the `RecyclerView`.
- **A ViewHolder.** Use an extension of `RecyclerView.ViewHolder` to contain the information for displaying one `View` item using the item's layout.

The diagram below shows the relationship between these components.



Data

Any displayable data can be shown in a `RecyclerView`.

- Text
- Images
- Icons

Data can come from any source.

- Created by the app. For example, scrambled words for a game.
- From a local database. For example, a list of contacts.
- From cloud storage or the internet. For example news headlines.

RecyclerView

A `RecyclerView` is a `ViewGroup` for a scrollable container. It is Ideal for long lists of similar items.
A `RecyclerView` uses a limited number of `View` items that are reused when they go off-screen. This saves memory and makes it faster to update list items as the user scrolls through data, because it is not necessary to create a new `View` for every item that appears.
In general, the `RecyclerView` keeps as many `View` items as can fit on the screen, plus a few extra at each end of the list to make sure that scrolling is fast and smooth.

Item Layout

The layout for a list item is kept in a separate XML layout file so that the adapter can create `View` items and edit their contents independently from the layout of the `Activity`.

Layout Manager

A layout manager positions `View` items inside a `ViewGroup`, such as the `RecyclerView`, and determines when to reuse `View` items that are no longer visible to the user. To reuse (or recycle) a `View`, a layout manager may ask the adapter to replace the contents of the `View` with a different element from the dataset. Recycling `View` items in this manner improves performance by avoiding the creation of unnecessary `View` items or performing expensive `findViewById()` lookups.
`RecyclerView` provides these built-in layout managers:

- `LinearLayoutManager` shows items in a vertical or horizontal scrolling list.
- `GridLayoutManager` shows items in a grid.
- `StaggeredGridLayoutManager` shows items in a staggered grid.

To create a custom layout manager, extend the `RecyclerView.LayoutManager` class.

Animations

Animations for adding and removing items are enabled by default in `RecyclerView`. To customize these animations, extend the `RecyclerView.ItemAnimator` class and use the `RecyclerView.setItemAnimator()` method.

Adapter

An *adapter* helps two incompatible interfaces to work together. In a `RecyclerView`, the adapter connects data with `View` items. It acts as an intermediary between the data and the `View`. The adapter receives or retrieves the data, does any work required to make it displayable in a `View`, and places the data in a `View`.
For example, the adapter may receive data from a database as a `Cursor` object, extract the the word and its definition, convert them to strings, and place the strings in a `View` item that has two `TextView` elements—one for the word and one for the definition. You will learn more about cursors in a later chapter.
The `RecyclerView.Adapter` implements a `ViewHolder`, and must override the following callbacks:

- `onCreateViewHolder()` inflates a `View` item and returns a new `ViewHolder` that contains it. This method is called when the `RecyclerView` needs a new `ViewHolder` to represent an item.
- `onBindViewHolder()` sets the contents of a `View` item at a given position in the `RecyclerView`. This is called by the `RecyclerView`, for example, when a new `View` item scrolls onto the screen.
- `getItemCount()` returns the total number of items in the data set held by the adapter.

ViewHolder

A `RecyclerView.ViewHolder` describes a `View` item and metadata about its place within the RecyclerView. Each `ViewHolder` holds one set of data. The adapter adds data to each `ViewHolder` for the layout manager to display.
You define your `ViewHolder` layout in an XML resource file. It can contain (almost) any type of `View`, including clickable elements.

Implementing a RecyclerView

Implementing a `RecyclerView` requires the following steps:

1. Add the `RecyclerView` dependency if needed (depending on which template is used for the `Activity`).
2. Add the `RecyclerView` to the `Activity` layout.
3. Create a layout XML file for one `View` item.
4. Extend `RecyclerView.Adapter` and implement the `onCreateViewHolder()` and `onBindViewHolder()` methods.
5. Extend `RecyclerView.ViewHolder` to create a `ViewHolder` for your item layout. You can add click behavior by overriding the `onClick()` method.
6. In the `Activity`, inside the `onCreate()` method, create a `RecyclerView` and initialize it with the adapter and a layout manager.

Adding the dependency

The `RecyclerView` library (`android.support.v7.widget.RecyclerView`) is part of the Support Library. Some `Activity` templates, such as the Basic Activity template, already include the Support Library dependency in the app's `build.gradle (Module: app)` file.
If Android Studio doesn't suggest `<android.support.v7.widget.RecyclerView` when entering **<RecyclerView** in the layout editor, you need to add the Support Library dependency for `RecyclerView` to the `dependencies` section of the app's `build.gradle (Module: app)` file:

```
compile 'com.android.support:recyclerview-v7:26.1.0'
```

If Android Studio highlights the above dependency and suggests a newer version, enter the version numbers for the newer version.

Adding a RecyclerView to the Activity layout

Add the `RecyclerView` to the `Activity` layout file:

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

Use the `RecyclerView` from the Support Library (`android.support.v7.widget.RecyclerView`) to be compatible with older versions of Android. The only required attributes are the `id`, the `layout_width`, and the `layout_height`. For customizing with more attributes, add the attributes to the items, not to this `RecyclerView`, which is a `ViewGroup`.

Creating the layout for one item

Create an XML resource file and specify the layout of one item. The adapter uses this code to create the `ViewHolder`.

```
<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>
```

The `TextView` has a `@style` element. A style is a collection of properties that specifies the look of a `View`. You can use styles to share display attributes with multiple `View` elements. An easy way to create a style is to extract the style of a `View` element that you already created. For example, after styling a `TextView`:

1. Right-click (or Control-click) the `TextView`.
2. Choose **Refactor > Extract > Style**.
3. Name your style, leave all other options selected, and select the L**aunch 'Use Style Where Possible**' option. Then click **OK**.
4. When prompted, apply the style to the **Whole Project**.

You learn more about styles in another lesson.

Creating an adapter with a ViewHolder

Extend RecyclerView.Adapter and implement the `onCreateViewHolder()` and `onBindViewHolder()` methods.
Create a new Java class with the following signature:

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

In the constructor, get an inflater from the current context, and your data.

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

For this adapter, you have to implement three methods:

1. `onCreateViewHolder()` creates a `View` and returns it.

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

11. `onBindViewHolder()` associates the data with the `ViewHolder` for a given position in the `RecyclerView`.

```
 @Override
 public void onBindViewHolder(
                     WordViewHolder holder, int position) {
         // Retrieve the data for that position
         String mCurrent = mWordList.get(position);
         // Add the data to the view
         holder.wordItemView.setText(mCurrent);
 }
```

20. `getItemCount()` returns to number of data items available for displaying.

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

Implementing the ViewHolder class

The `ViewHolder` class for your item layout is usually defined as an inner class to the adapter. Extend `RecyclerView.ViewHolder` to create the `ViewHolder`. You can add click behavior by overriding the `onClick()`method.

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

If you want to add click handling, you need to implement `View.onClickListener`. One way to do this is to have the `ViewHolder` implement the `View.onClickListener` methods.

```
// Extend the signature of WordViewHolder to implement a click listener.
class WordViewHolder extends RecyclerView.ViewHolder
                                        implements View.OnClickListener {
```

In its constructor, the `ViewHolder` has to inflate its layout, associate with its adapter, and, if applicable, set a click listener.

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

And, if you implementing `View.onClickListener`, you also have to implement `onClick()`.

```
@Override
public void onClick(View v) {
    wordItemView.setText ("Clicked! "+ wordItemView.getText());
}
```

If you want to attach click listeners to other elements of the `ViewHolder`, do that dynamically in `onBindViewHolder()` (you will do this in another practical).

Creating the RecyclerView

Finally, to tie it all together, add to the `Activity` the following:

1. Declare a `RecyclerView`.

```
 private RecyclerView mRecyclerView;
```

3. In the `Activity` `onCreate()` method, get a handle to the `RecyclerView` in the layout:

```
 mRecyclerView = findViewById(R.id.recyclerview);
```

5. Create an adapter and supply the data to be displayed.

```
 mAdapter = new WordListAdapter(this, mWordList);
```

7. Connect the adapter with the `RecyclerView`.

```
 mRecyclerView.setAdapter(mAdapter);
```

9. Give the `RecyclerView` a default layout manager.

```
 mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
```

`RecyclerView` is an efficient way to display scrolling list data. It uses the adapter pattern to connect data with list item views. To implement a `RecyclerView`, you need to create an adapter and a `ViewHolder`. You also need to create the methods that take the data and add it to the list items.

Related practical

The related practical 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)