

## 4.1: User Input Controls

- Interaction design for user input
- Using buttons
- Using input controls for making choices
- Text input
- Using dialogs and pickers
- Recognizing gestures

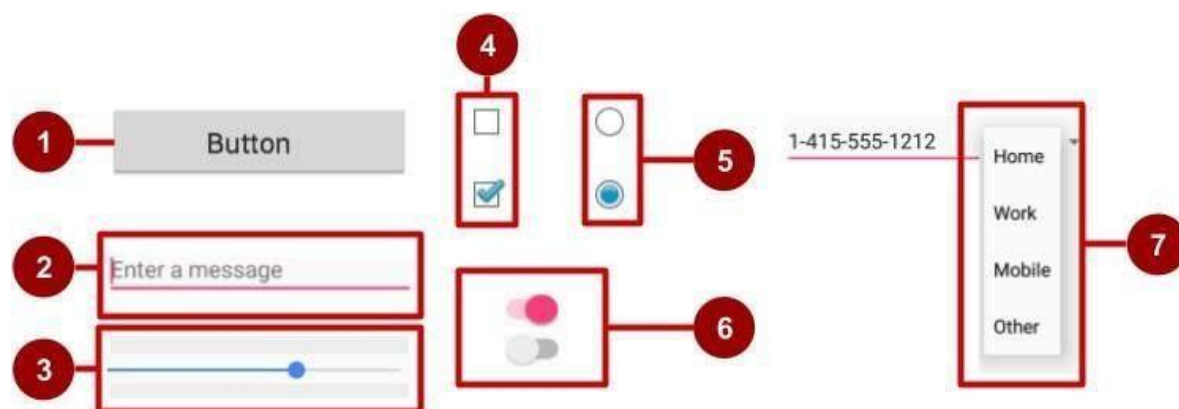
### Interaction design for user input

The purpose of designing an app is to offer a specific function to the user, and for this purpose, a means of interaction must be established. In the case of an Android app, interaction typically involves actions like tapping, pressing, typing, or even voice commands and feedback. To facilitate this interaction, the app framework provides various user interface (UI) elements such as buttons, menus, keyboards, text input fields, and a microphone.

In the following chapter, you will explore into the art of designing an app's user interaction. This encompasses creating buttons to trigger actions and implementing text input fields for user data entry. A critical aspect of your design is the ability to anticipate and accommodate user needs, ensuring that the UI elements are readily accessible, understandable, and user-friendly. When your app requires user input, it should be both effortless and intuitive. Provide users with the necessary guidance, like predicting the data source, minimizing the need for excessive gestures such as taps and swipes, and offering pre-filled forms when feasible.

Your app should strive for intuitiveness; it should behave in a manner that aligns with users' expectations. Just as you expect a car's steering wheel, gear shift, headlights, and indicators to be in certain locations when you rent a car, or a light switch in a specific spot when entering a room, an app's interface should meet user expectations. Users anticipate that buttons will be clickable, spinners will reveal drop-down menus, and text input fields will activate the onscreen keyboard when tapped. Deviating from established user expectations can make it challenging for users to navigate and utilize your app effectively.

**Note:** Android users have grown accustomed to the behavior of UI elements conforming to specific patterns. It's crucial to maintain consistency with the experiences of other Android apps and be predictable in your design choices and their arrangement. This approach contributes to the creation of apps that effectively meet your customers' expectations. This chapter serves as an introduction to Android's input controls, which are the interactive components within your app's user interface. Your UI can incorporate a diverse array of input controls, including text fields, buttons, checkboxes, radio buttons, toggle buttons, spinners, and numerous others.



In the above figure:

1. Button

2. Text field
3. Seek bar
4. Checkboxes
5. Radio buttons
6. Toggle
7. Spinner

## Input controls and view focus

Android employs a universal programmatic concept called a "view" as the foundation for all input controls. The `View` class serves as the core component for various UI elements, including input controls. In preceding chapters, we've explored how the `View` class acts as the base for classes supporting interactive UI components like buttons, text fields, and layout managers.

Now, in a scenario where your app hosts numerous UI input components, you might wonder which one takes precedence in receiving user input. For example, when you have multiple `TextView` objects and an `EditText` object in your app, which UI component (or `View`) will be the initial recipient of user-typed text?

The `View` with "focus" is the one that receives user input. `Focus` designates the currently selected view for input. Users activate focus by interacting with a `View`, like a `TextView` or an `EditText`. You can establish a focus sequence, guiding users through UI controls using keys like `Return`, `Tab`, or arrow keys. Additionally, focus can be programmatically managed, allowing a programmer to use the `requestFocus()` method on any focusable `View`.

Another notable attribute of an input control is its "clickable." When set to `true` (a boolean value), the `View` can respond to click events. Similar to focus, the "clickable" attribute can be programmatically managed. It's important to distinguish between "clickable" and "focusable." Clickable denotes a view's capacity to be clicked or tapped, while focusable signifies a view's ability to gain focus from an input device, such as a keyboard. Input devices like keyboards rely on the view that currently holds focus to determine where to send their input events.

Android device input methods have become increasingly diverse, including directional pads, trackballs, touch screens, keyboards, and more. Some devices, such as tablets and smartphones, primarily rely on touch navigation. In contrast, devices like Google TV lack touch screens and depend on input devices like directional pads (d-pad). When users navigate a user interface using input methods like directional keys or a trackball, it is essential to:

- 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.

Most of the time, you won't need to manually control focus, unless you're creating a set of text input fields and want users to navigate between them by tapping the `Return` or `Tab` key. Android introduces "touch mode" for touch-enabled devices like smartphones and tablets. When users initially engage with the interface by touching it, only `Views` with the attribute `isFocusableInTouchMode()` set to `true` can gain focus, such as text input fields. Other touch-sensitive `Views`, like buttons, don't acquire focus when touched. If the user uses directional keys or scrolls with a trackball, the device exits "touch mode" and identifies a `View` to focus on.

Focus movement is based on an algorithm that finds the nearest neighbor in a given direction:

- When the user touches the screen, the topmost view under the touch is in focus, providing touch-access for the child views of the topmost view.
- If you set an `EditText` view to a single-line, the user can tap the `Return` key on the 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 directional pad (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
    android:orientation="vertical"
    ... >
<Button android:id="@+id/top"
        android:nextFocusUp="@+id/bottom"
        ... />
<Button android:id="@+id/bottom"
        android:nextFocusDown="@+id/top"
        ... />
</LinearLayout>
```

Ordinarily in a vertical `LinearLayout`, navigating up from the first `Button` would not go anywhere, nor would navigating down from the second `Button`. But in the above example, the top `Button` has defined the bottom `button` as the `nextFocusUp` (and vice versa), so the navigation focus will cycle from top-to-bottom and bottom-to-top.

If you'd like 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" with `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 will be notified when the view gains or loses focus, use `setOnFocusChangeListener`.

Understanding focus with respect to input controls is essential for understanding how the on-screen keyboard works with text editing views. For example, depending on which attributes you use with an `EditText` view, tapping the Return key in the keyboard can either enter a new line, or advance the focus to the next view. You'll learn more about focus with text editing views later in this chapter.

## Using buttons

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

You can make a `Button` using:

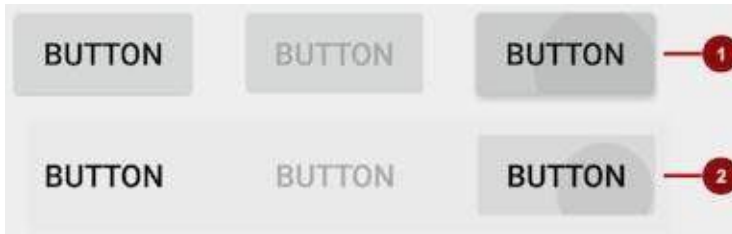
- Only text, as shown on the left side of the figure below.
- Only an icon, as shown in the center of the figure below.
- Both text and an icon, as shown on the right side of the figure below.

When touched or clicked, a button performs an action. The text and/or icon provides a hint of that action. It is also referred to as a "push-button" in Android documentation.



A button is a rectangle or rounded rectangle, wider than it is tall, with a descriptive caption in its center. Android buttons follow the guidelines in the the Android Material Design Specification—you will learn more about that in a later lesson.

Android offers several types of buttons, including raised buttons and flat buttons as shown in the figure below. These buttons have three states: normal, disabled, and pressed.



In the above figure:

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

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

1. In your **build.gradle (Module: app)** file, add the newest appcompat library to the `dependencies` section:

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

In the above, `x.x.x.` 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.

2. Make your activity extend `android.support.v7.app.AppCompatActivity` :

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

3. Use the `Button` element in the layout file. There is no need for an additional attribute, as a raised button is the default style.

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    ... />
```

Use raised buttons to give more prominence to actions in layouts with a lot of varying content. Raised buttons add dimension to a flat layout—they emphasize functions on busy or wide spaces. Raised buttons show a background shadow



when touched (pressed) or clicked, as shown below.

In the above figure:

1. Normal state: In its normal state, the button looks like a raised button.
2. Disabled state: When the button is disabled, it is grayed out and it's 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 `OnClick` attribute), the callback is called when the button is in this state.

## Creating a raised button with text

Some raised buttons are best designed as text, without an icon, such as a "Save" button, because an icon by itself might not convey an obvious meaning. To create a raised button with text, use the `Button` class, which extends the `TextView` class.

To create a raised button with just text, use the `Button` class in your XML layout as follows:

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

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

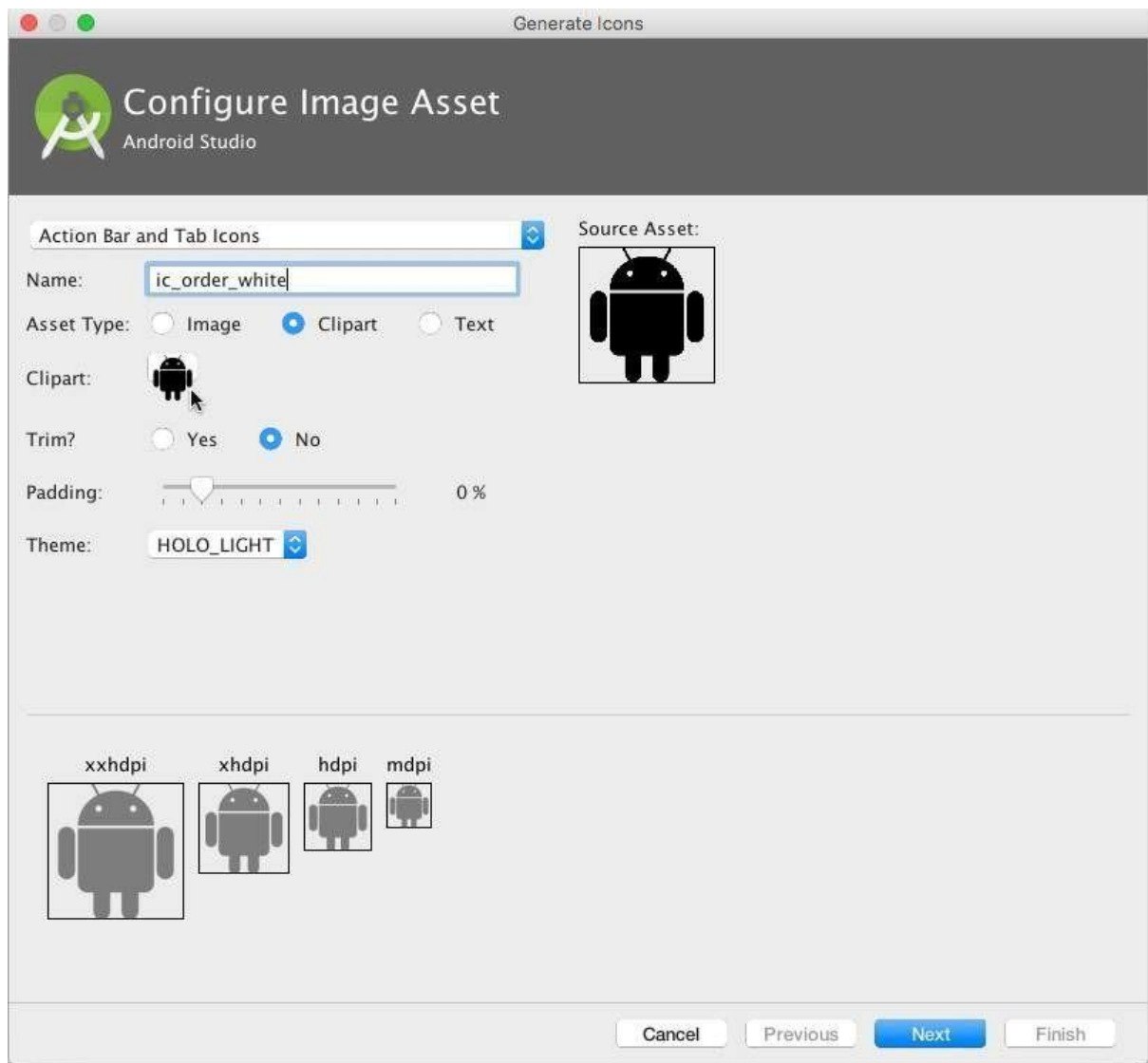
## Creating a raised button with an icon and text

While a button usually displays text that tells the user what the button is for, raised buttons can also display icons 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 view, and right-click (or Command-click) **drawable**.
2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.



3. Choose **Action Bar and Tab Items** in the drop-down menu of the Configure Image Asset dialog (see Image Asset Studio for a complete description of this dialog.)
4. Click the **Clipart**: image (the Android logo) to select a clipart image as the icon. A page of icons appears as shown below. Click the icon you want to use.





5. You may want to make the following adjustments:
  - Choose **HOLO\_DARK** from the Theme drop-down menu to sets the icon to be white against a dark-colored (or black) background.
  - 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.
6. 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 view, and right-click (or Command-click) **drawable**.
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 spec (see Add Multi-Density Vector Graphics for a complete description of this dialog).
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"
    ... />
```



## 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"
    ... />
```

## 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 above declaration, 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 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.

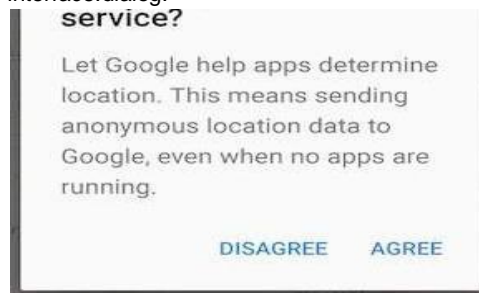
If you're not already familiar with Android's style and theme system, you should read [Styles and Themes](#). The blog post "[Holo Everywhere](#)" provides information about using the Holo theme while supporting older devices.



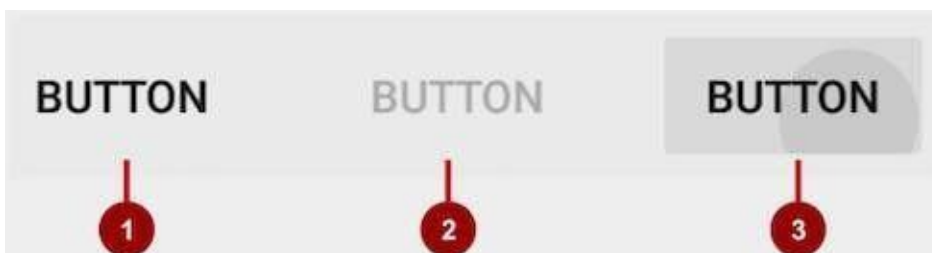
For a guide on styling and customizing buttons using XML, see Buttons (in the "User Interface" section of the Android developer guide). For a comprehensive guide to designing buttons, see "Components - Buttons" in the Material Design Specification.

## Designing flat buttons

A flat button, often referred to as a borderless button, is a text-only button that presents a clean, shadow-free appearance on the screen. The primary advantage of flat buttons is their simplicity, as they reduce visual distractions and seamlessly blend with the content. Flat buttons are particularly valuable in scenarios like the dialog shown in the figure below, where user input or interaction is needed. In such cases, maintaining consistent font and style with the surrounding text ensures a cohesive look and feel across all elements within the interface.



Flat buttons, shown below, resemble basic buttons except that they have no borders or background, but still change appearance during different states. A flat button shows an ink shade around it when pressed (touched or clicked).



In the above figure:

1. Normal state: In its normal state, the button looks just like ordinary text.
2. Disabled state: When the text is grayed out, the button is not active in the app's context.
3. Pressed state: The pressed state, with a 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.

**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:

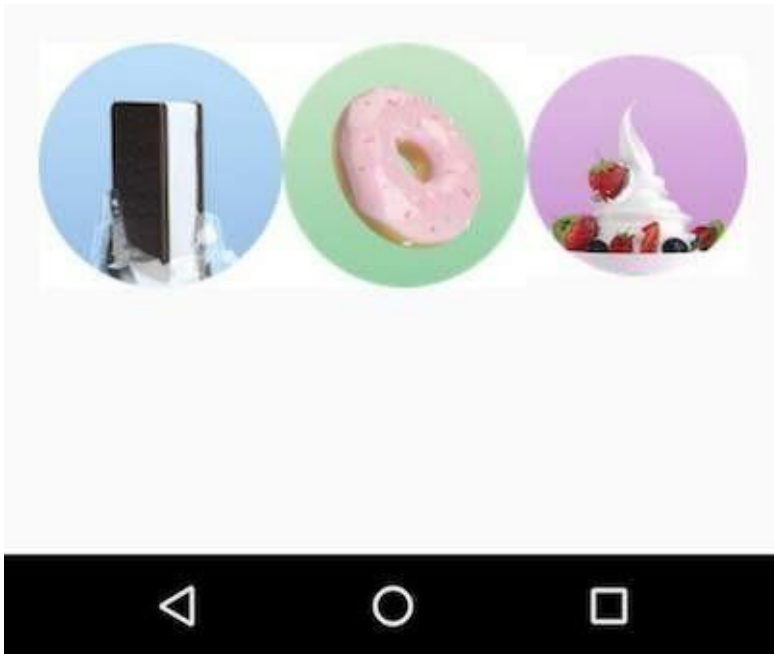
```
<Button
    android:id="@+id/button_send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage"
    style="?android:attr/borderlessButtonStyle" />
```

## Designing images as buttons

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 **drawables** folder of your project.

**Note:** To bring images into your Android Studio project, create or save the image in JPEG format, and copy the image file into the **app > src > main > res > drawables** folder of your project. For more information about drawable resources, see [Drawable Resources](#) in the App Resources section of the [Android Developer Guide](#).

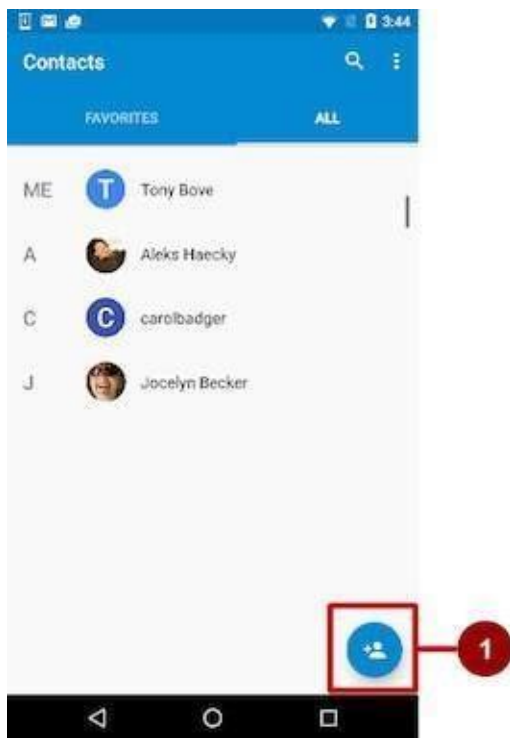
If you are using multiple images as buttons, arrange them in a viewgroup so that they are grouped together. For example, the following images in the drawable folder (`icecream_circle.jpg`, `donut_circle.jpg`, and `froyo_circle.jpg`) are defined for `ImageView`s that are grouped in a `LinearLayout` set to a horizontal orientation so that they appear side-by-side:



```
<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>
```

## Designing a floating action button

A floating action button, 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's 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".

To use a floating action button in your Android Studio project, you must add the following statement to your **build.gradle (Module: app)** file in the `dependencies` section:

```
compile 'com.android.support:design:23.4.0'
```

**Note:** The version number at the end of the statement may change; use the newest version suggested by Android Studio. To create a floating action button, 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.

## Responding to button-click events

Use an *event listener* called `OnClickListener`, which is an interface in the `View` class, to respond to the click event that occurs when the user taps or clicks a clickable object, such as a `Button`, `ImageButton`, or `FloatingActionButton`. For more information on event listeners, or other types of UI events, read the [Input Events](#) section of the [Android Developer Documentation](#).

## Adding `onClick` to the layout element

To set up an `OnClickListener` for the clickable object in your Activity code and assign a callback method, use the

`android:onClick` attribute with the clickable object's element in the XML layout. For example:

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

In this case, when a user clicks the button, the Android system calls the Activity's `sendMessage()` method:

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

The method you declare as the `android:onClick` attribute must be `public`, return `void`, and define a `View` as its only parameter (this will be the view that was clicked). Use the method to perform a task or call other methods as a response to the button click.

## Using the button-listener design pattern

You can also handle the click event programmatically using the button-listener design pattern (see 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 = (Button) findViewById(R.id.button_send);
```

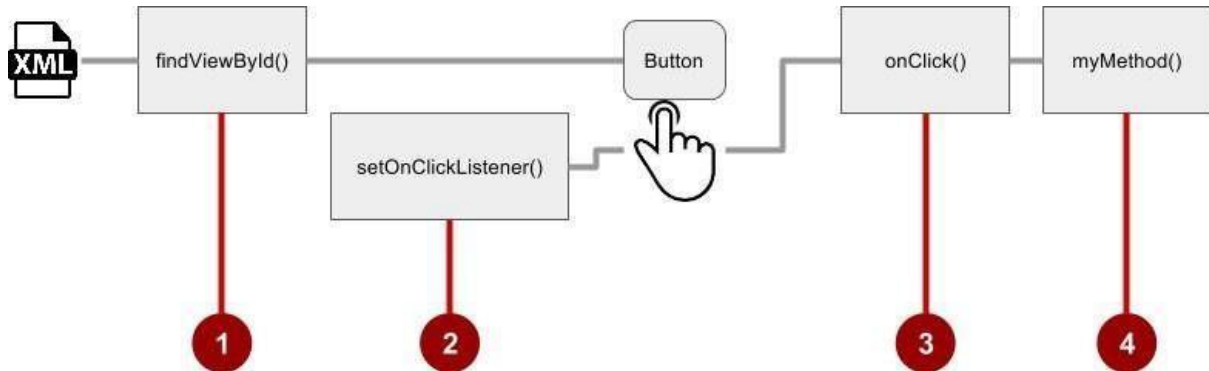
2. Get a new `View.OnClickListener` object and register it to the button by calling the `setOnClickListener()` method. The argument to `setOnClickListener()` takes an object that implements the `View.OnClickListener` interface, which has one method: `onClick()`.

```
button.setOnClickListener(new View.OnClickListener() {
    ...
})
```

3. Define the `onClick()` method to be `public`, return `void`, and define a `View` as its only parameter:

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

4. Create a method to do something in response to the button click, such as perform an action.



To set the click listener programmatically instead of with the `onClick` attribute, customize the `View.OnClickListener` class and override its `onClick()` handler to perform some action, as shown below:

```
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        // Add a new word to the wordList.
    }
});
```

## Using the event listener interface for other events

Additional events may arise with UI elements, and you can employ the callback methods provided within the event listener interfaces to manage them. These methods are automatically invoked by the Android framework when a view, for which the listener has been registered, is activated through user interaction. Consequently, it's essential to assign the relevant listener to make use of the method. Here are some of the listeners available in the Android framework, along with their associated callback methods:

- `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 the touch over a view for an extended period. This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return `true` to indicate that you have handled the event and it should stop here; return `false` if you have not handled it and/or the event should continue to any other on-click 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 input controls for making choices

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

**Checkboxes:** Select one or more values from a set of values by clicking each value's checkbox.

**Radio buttons:** Select only one value from a set of values by clicking the value's circular "radio" button. If you are providing only two or three choices, you might want to use radio buttons for the choices if you have room in your layout

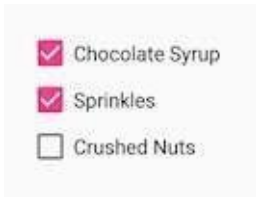
for them.

- Toggle button: Select one state out of two or more states. Toggle buttons usually offer two visible states, such as "on" and "off".
- Spinner: Select one value from a set of values in a drop-down menu. Only one value can be selected. Spinners are useful for three or more choices, and takes up little room in your layout.

## Checkboxes

Use checkboxes when you have a list of options and the user may select *any number* of choices, including no choices. Each checkbox is independent of the other checkboxes in the list, so checking one box doesn't uncheck the others. (If you want to limit the user's selection to only one item of a set, use radio buttons.) A user can also uncheck an already checked checkbox.

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



Each checkbox is a separate instance of the `CheckBox` class. You create each checkbox using a `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 checkboxes when a user touches 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 above `Button` example—must be `public`, return `void`, and define a `View` as a parameter (the view that was clicked). In this method you can determine if a checkbox is selected by using the `isChecked()` method (inherited from `CompoundButton`). The `isChecked()` method will return a (boolean) `true` if there is a checkmark in the box. For example, the following statement assigns the boolean value of `true` or `false` to `checked` depending on whether the checkbox is checked:

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



The following code snippet shows how the `onSubmit()` method might check to see which checkbox is selected, using the resource `id` for the checkbox element:

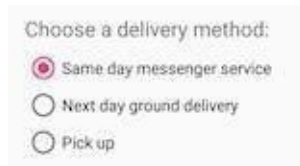
```
public void onSubmit(View view) {
    StringBuffer toppings = new
        StringBuffer().append(getString(R.string.toppings_label));
    if (((CheckBox) findViewById(R.id.checkbox1_chocolate)).isChecked()) {
        toppings.append(getString(R.string.chocolate_syrup_text));
    }
    if (((CheckBox) 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));
    }
    ...
}
```

**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, which must be defined within the activity that hosts this layout.

For more information about checkboxes, see [Checkboxes](#) in the User Interface section of the Android Developer Documentation.

## Radio buttons

Use radio buttons when you have two or more options that are mutually exclusive—the user must select only one of them.



(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 horizontally if the labels are short.

Each radio button is an instance of the `RadioButton` class. Radio buttons are normally used together in a `RadioGroup`. When several radio buttons live inside a radio group, checking one radio button unchecks all the others. You create each radio button using a `RadioButton` element in your XML layout within a `RadioGroup` view group:

```
<RadioGroup
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:layout_below="@id/orderintrotetext">
    <RadioButton
        android:id="@+id/sameday"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/same_day_messenger_service"
        android:onClick="onRadioButtonClicked"/>
    <RadioButton
        android:id="@+id/nextday"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/next_day_ground_delivery"
        android:onClick="onRadioButtonClicked"/>
    <RadioButton
        android:id="@+id/pickup"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/pick_up"
        android:onClick="onRadioButtonClicked"/>
</RadioGroup>
```

Use the `android:onClick` attribute for each radio button to declare the click event handler method for the radio button, which must be defined within the activity that hosts this layout. In the above layout, clicking any radio button calls the same `onRadioButtonClicked()` method in the activity, but you could create separate methods in the activity and declare them in each radio button's `android:onClick` attribute.

The click event handler method must be `public`, return `void`, and define a `View` as its only parameter (the view that was clicked). The following shows one method, `onRadioButtonClicked()`, for all radio buttons, using `switch case` statements to check the resource `id` for the radio button 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();

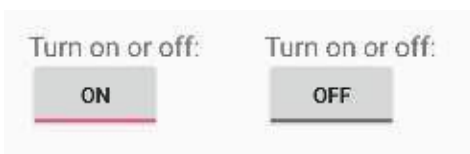
    // Check which radio button was clicked.
    switch(view.getId()) {
        case R.id.sameday:
            if (checked)
                // Same day service
                break;
        case R.id.nextday:
            if (checked)
                // Next day delivery
                break;
        case R.id.pickup:
            if (checked)
                // 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 User Interface section of the Android Developer Documentation.

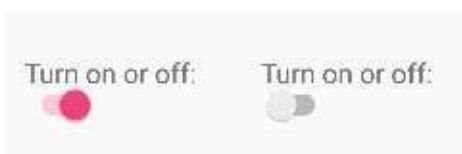
## Toggle buttons and switches

A toggle input control lets the user change a setting between two states. 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 offering two states (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 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 toggle. Create a `CompoundButton.OnCheckedChangeListener` object and assign it to the button 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 = (ToggleButton) 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:

Turn on or off: 

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 button by calling `setOnCheckedChangeListener()`. For example, the `onSwitchClick()` method checks whether the switch is on or off, and displays a toast message:

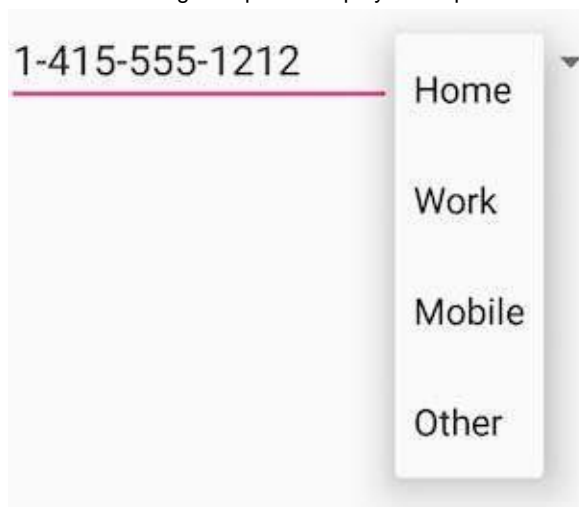
```
public void onSwitchClick(View view) {
    Switch aSwitch = (Switch) findViewById(R.id.my_switch);
    aSwitch.setOnCheckedChangeListener(new
        CompoundButton.OnCheckedChangeListener() {
            public void onCheckedChanged(CompoundButton buttonView,
                boolean isChecked) {
                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 User Interface section of the Android Developer Documentation.

## Spinners

A *spinner* provides a quick way to select one value from a set. Touching the spinner displays a drop-down list with all



available values, from which the user can select one.

If you have a long list of choices, a spinner may extend beyond your layout, forcing the user to scroll it. A spinner scrolls automatically, with no extra code needed. However, scrolling a long list (such as a list of countries) is not recommended as it can be hard to select an item.

To create a spinner, use the `Spinner` class, which creates a view that displays individual spinner values as child views, 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

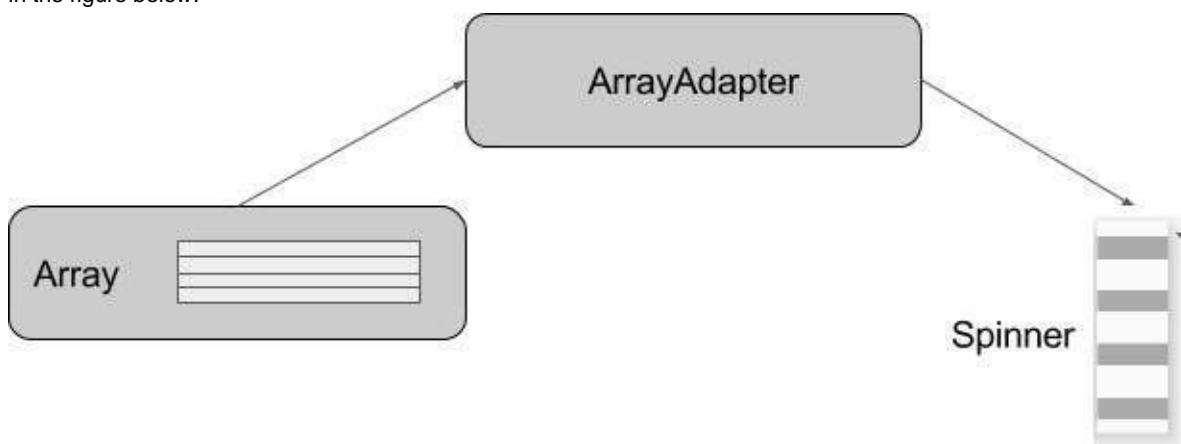
To create a spinner in your XML layout, add a Spinner element, which provides the drop-down list:

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

## Specify the values for the spinner

You 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 spinner-adapter pattern takes the data set you've specified and makes a view for each item in the data set, 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 available 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 could come from a source such as a stored file or a database. You learn more about stored data in another chapter.

## 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 when the view is created in the `onCreate()` method. Follow these steps (refer to the full `onCreate()` method at the end of the steps):

1. Add the code below to the `onCreate()` method, which does the following:
2. Gets the spinner object you added to the layout using `findViewById()` to find it by its `id (label_spinner)`.

3. Sets the `onItemSelectedListener` to whichever activity implements the callbacks ( `this` ) using the `setOnItemSelectedListener()` method.

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

    // Create the spinner.
    Spinner spinner = (Spinner) findViewById(R.id.label_spinner);
    if (spinner != null) {
        spinner.setOnItemSelectedListener(this);
    }
}
```

4. Also in the `onCreate()` method, add a statement that creates the `ArrayAdapter` with the string array:

```
// 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);
```

As shown above, you use the `createFromResource()` method, which takes as arguments:

5. The activity that implements the callbacks for processing the results of the spinner ( `this` )
6. The array ( `labels_array` )
7. The layout for each spinner item ( `layout.simple_spinner_item` ).

**Tip:** You should use the `simple_spinner_item` default layout, unless you want to define your own layout for the items in the spinner.

8. Specify the layout the adapter should use to display the list of spinner choices by calling the `setDropDownViewResource()` method of the `ArrayAdapter` class. For example, you can use

`simple_spinner_dropdown_item` as your layout:

```
// Specify the layout to use when the list of choices appears.
adapter.setDropDownViewResource
    (android.R.layout.simple_spinner_dropdown_item);
```

**Tip:** You should use the `simple_spinner_dropdown_item` default layout, unless you want to define your own layout for the spinner's appearance.

9. Use `setAdapter()` to apply the adapter to the spinner:

```
// Apply the adapter to the spinner.
spinner.setAdapter(adapter);
```

The full code for the `onCreate()` method is shown below:



```

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

    // Create the spinner.
    Spinner spinner = (Spinner) findViewById(R.id.label_spinner);
    if (spinner != null) {
        spinner.setOnItemSelectedListener(this);
    }

    // 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.
    adapter.setDropDownViewResource
        (android.R.layout.simple_spinner_dropdown_item);

    // Apply the adapter to the spinner.
    if (spinner != null) {
        spinner.setAdapter(adapter);
    }
}

```

## 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 MainActivity extends AppCompatActivity implements
    AdapterView.OnItemSelectedListener {

```

Android Studio automatically imports the `AdapterView` widget. Implement the `AdapterView.OnItemSelectedListener` interface in order to have the `onItemSelected()` and `onNothingSelected()` callback methods to use with the spinner object.

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` object 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 menu using the `getItemAtPosition()` method of the `AdapterView` class:

```

public void onItemSelected(AdapterView<?> adapterView, View view, int
    pos, long id) {
    String spinner_item = adapterView.getItemAtPosition(pos).toString();
}

```

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

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

4. Implement/override the `onNothingSelected()` callback method of the `AdapterView.OnItemSelectedListener` interface to do something if nothing is selected.

For more information about spinners, see Spinners in the "User Interface" section of 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.

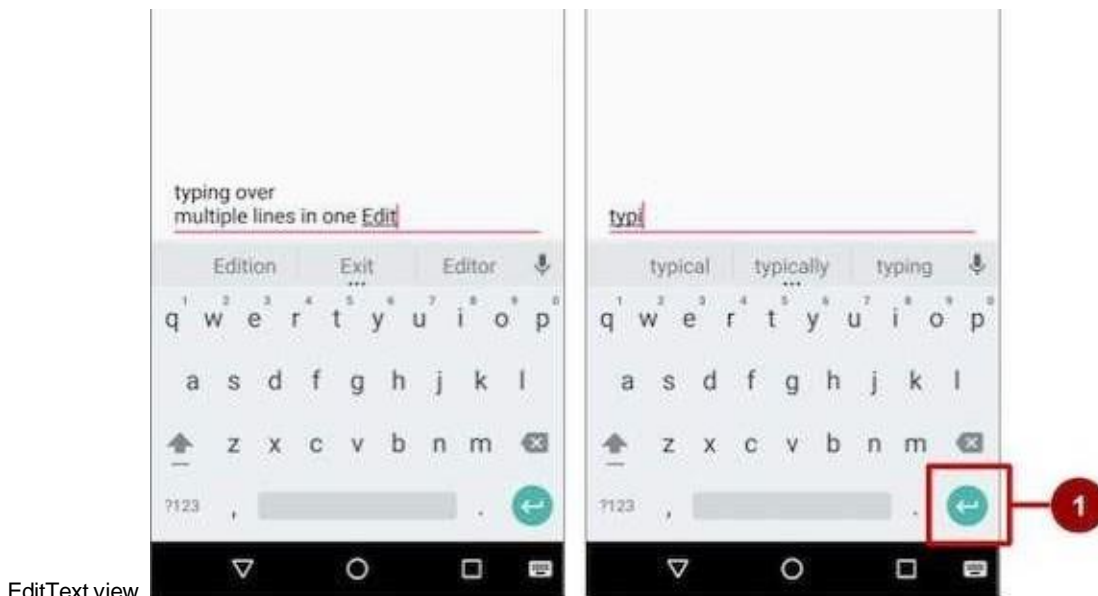
### Customizing an `EditText` object for user input

In the Layout Manager of Android Studio, 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">
</EditText>
```

### Enabling multiple lines of input

By default, the `EditText` view allows multiple lines of input as shown in the figure below, and suggests spelling corrections. Tapping the Return (also known as Enter) key on the on-screen keyboard ends a line and starts a new line in the same



EditText view.

**Note:** In the above figure, #1 is the Return (also known as Enter) key.

### Enabling Return to advance to the next view

If you add the `android:inputType` attribute to the `EditText` view with a value such as `"textCapCharacters"` (to change the input to all capital letters) or `"textAutoComplete"` (to enable spelling suggestions as the user types), tapping the Return key closes the on-screen 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` view.

### Attributes for customizing an `EditText` view

Use attributes to customize the EditText view for input. For example:

- `android:maxLines="1"` : Set the text entry to show only one line.
- `android:lines="2"` : Set the text entry to show 2 lines, even if the length of the text is less.
- `android:maxLength="5"` : Set the maximum number of input characters to 5.
- `android:inputType="number"` : Restrict text entry to numbers.
- `android:digits="01"` : Restrict the digits entered to just "0" and "1".
- `android:textColorHighlight="#7cffe8"` : 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

Enabling the user to input text is only useful if you can *use* that text in some way in your app. To use the input, you must first get it from the EditText view in the XML layout. The steps you follow to set up the EditText view and get user input from it are:

1. Create the EditText view element in the XML layout for an 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"
```

2. In the Java code for the same activity, create a method with a `View` parameter that gets the `EditText` object (in the example below, `editText` ) for the `EditText` view, using the `findViewById()` method of the `View` class to find the view by its `id` ( `editText_main` ):

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

3. Use the `getText()` method of the `EditText` class (inherited from the `TextView` class) to obtain the text as a character sequence (`CharSequence`). You can convert the character sequence into a string using the `toString()` method of the `CharSequence` class, which returns a string representing the data in the character sequence.

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

**Tip:** You can use the `valueOf()` method of the `Integer` class to convert the string to an integer if the input is an integer.

## Changing keyboards and input behaviors

The Android system shows an on-screen keyboard—known as a *soft* input method—when a text field in the UI receives focus. To provide the best user experience, you can specify characteristics about the type of input the app expects, such as whether it's a phone number or email address. You can also specify how the input method should behave, such as whether or not it shows spelling suggestions or provides capital letters for the beginning of a sentence. You can change the soft input method to a numeric keypad for entering only numbers, or even a phone keypad for phone numbers.

Android also provides an extensible framework for advanced programmers to develop and install their own Input Method Editors (IME) for speech input, specific types of keyboard entry, and other applications.

Declare the input method by adding the `android:inputType` attribute to the EditText view. For example, the following attribute sets the on-screen keyboard to be a phone keypad:

```
android:inputType="phone"
```

Use the `android:inputType` attribute with the following values:

- `textCapSentences` : Set the keyboard to capital letters at the beginning of a sentence.
- `textAutoCorrect` : Enable spelling suggestions as the user types.
- `textPassword` : Turn each character the user enters into a dot to conceal an entered password.
- `textEmailAddress` : For email entry, show an email keyboard with the "@" symbol conveniently located next to the space key.
- `phone` : For phone number entry, show a numeric phone keypad.

**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 [Specifying the Input Method Type](#) in the developer documentation. For a complete list of constant values for `android:inputType`, see the "android:inputType" section of the [TextView documentation](#).

## Changing the "action" key in the keyboard

On Android devices, the "action" key is the **Return** key. This key is normally used to enter another line of text for an `EditText` element that allows multiple lines. If you set an `android:inputType` attribute for the `EditText` view with a value such as `"textCapCharacters"` (to change the input to all capital letters) or `"textAutoComplete"` (to enable spelling suggestions as the user types), the **Return** key closes the on-screen keyboard and advances the focus to the next view.

If you want the user to enter something other than text, such as a phone number, you may want to change the "action" key to an icon for a **Send** key, and change the action to be dialing a phone number. Follow these steps:

1. Use the `android:inputType` attribute to set an input type for the keyboard:

```
<EditText
    android:id="@+id/phone_number"
    android:inputType="phone"
    ... >
</EditText>
```

The `android:inputType` attribute, in the above example, sets the keyboard type to `phone`, which forces one line of input (for a phone number).

2. Use `setOnEditorActionListener()` to set the listener for the `EditText` view to respond to the use of the "action" key:

```
EditText editText = (EditText) findViewById(R.id.phone_number);
editText.setOnEditorActionListener(new
    TextView.OnEditorActionListener() {
    // Add onEditorAction() method
    }
```

3. Use the `IME_ACTION_SEND` constant in the `EditorInfo` class for the `actionId` to show a **Send** key as the "action" key, and create a method to respond to the pressed **Send** key (in this case, `dialNumber` to dial the entered phone number):

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

**Note:** For help setting the listener, see "Specifying the Input ActionSpecifying Keyboard Actions" in [Text Fields](#). For

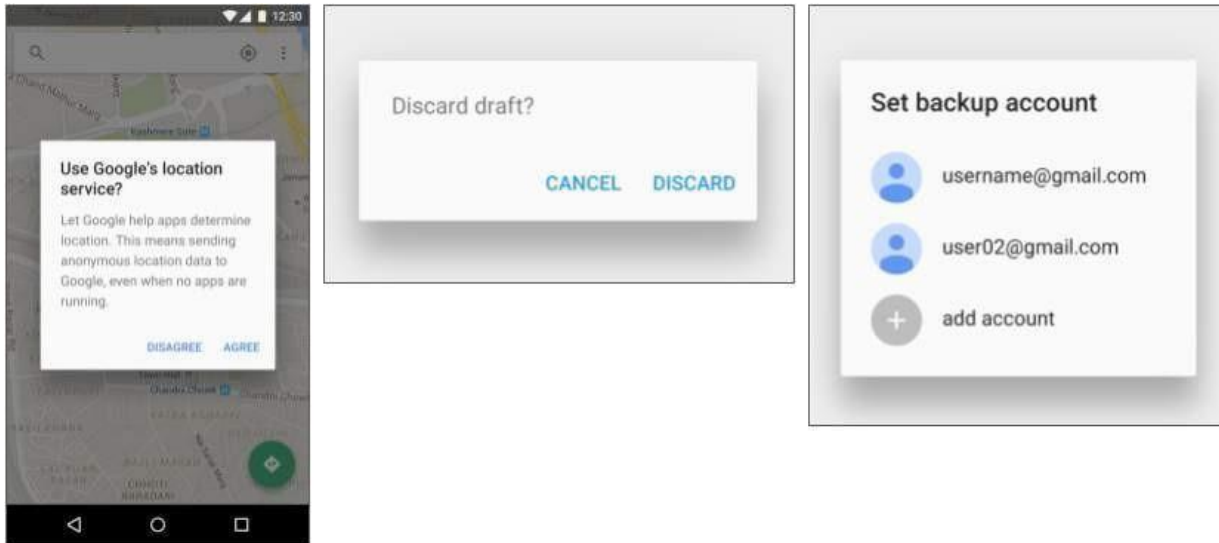
more information about the EditorInfo class, see the EditorInfo documentation.

## Using dialogs and pickers

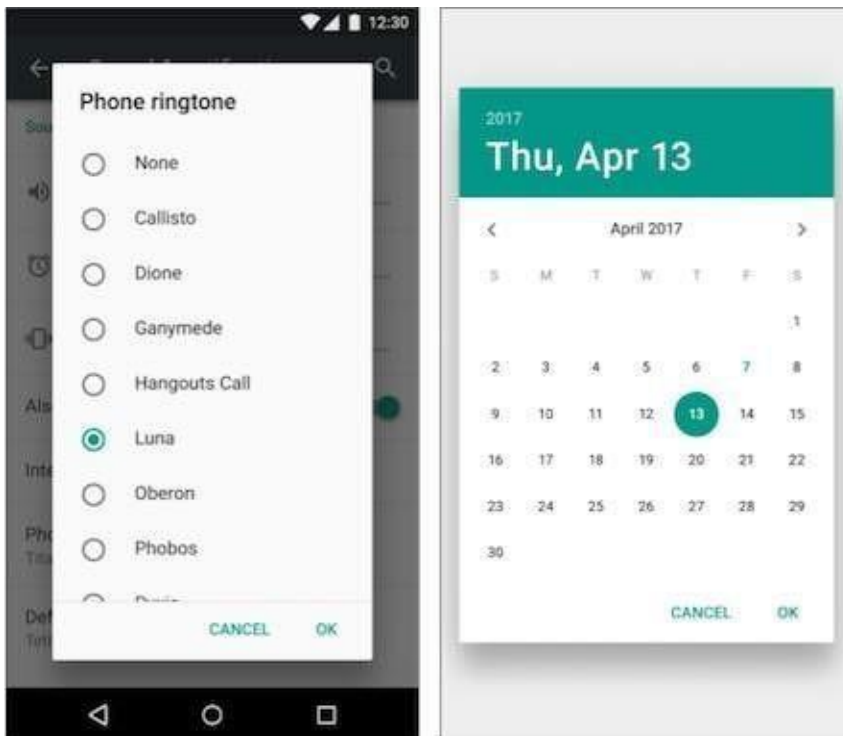
A dialog is a window that overlays or occupies the entire display, momentarily pausing the ongoing activity. Dialogs serve to apprise users of a specific task, conveying essential information, soliciting decisions, or managing multiple tasks.

For example, you would typically use a dialog to show an alert that requires users to tap a button make a decision, such as **OK** or **Cancel**. In the figure below, the left side shows an alert with **Disagree** and **Agree** buttons, and the center shows an alert with **Cancel** and **Discard** buttons.

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 work flow. Read the Dialogs design guide for additional best design practices, and Dialogs in the Android developer documentation for code examples.

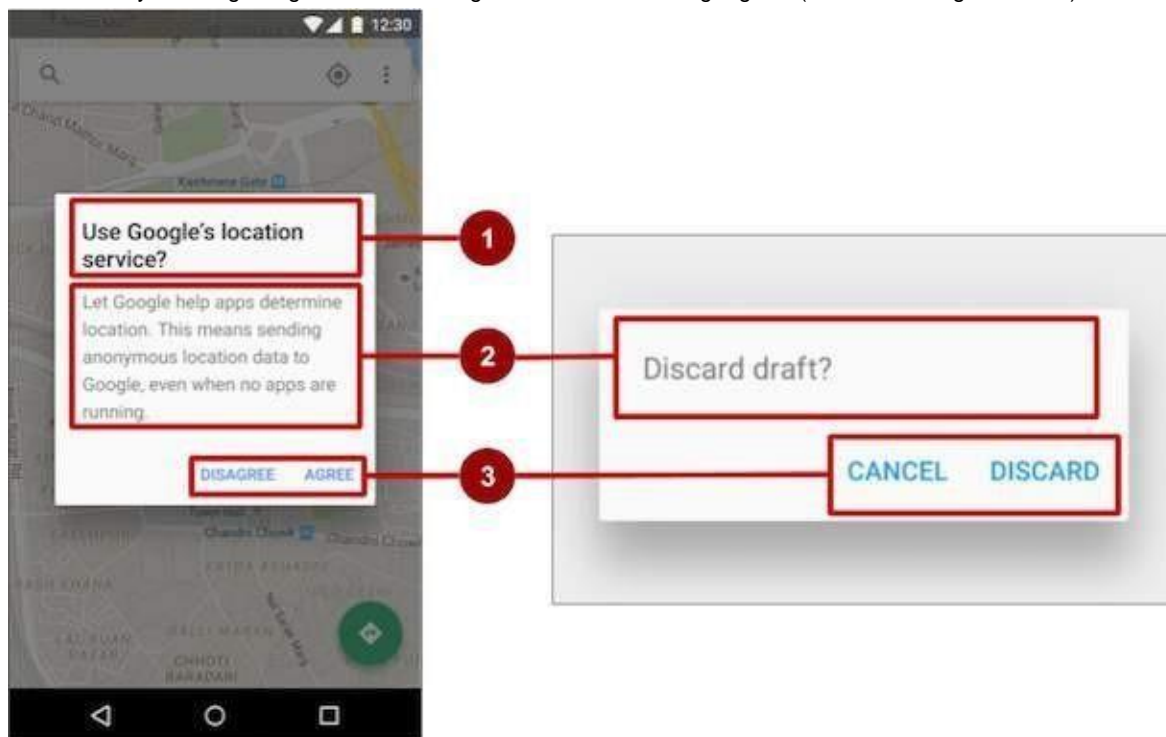
The Dialog class is the base class for dialogs, but you should avoid instantiating Dialog directly unless you are creating a custom dialog. For standard Android dialogs, use one of the following subclasses:

- AlertDialog: A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.
- DatePickerDialog or TimePickerDialog: A dialog with a pre-defined UI that lets the user select a date or 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 and set attributes on the dialog. Use `setTitle()` to set its title, `setMessage()` to set its message, and `setPositiveButton()` and `setNegativeButton()` to set its buttons.

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

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

The following creates the dialog object ( `myAlertBuilder` ) and sets the title (the string resource called `alert_title` ) and message (the string resource called `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 of the `AlertDialog.Builder` class to set the button actions for the alert dialog. These methods require a title for the button (supplied by a string resource) 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.
    }
});
myAlertBuilder.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int which) {
        // User clicked the CANCEL button.
    }
});
```

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.

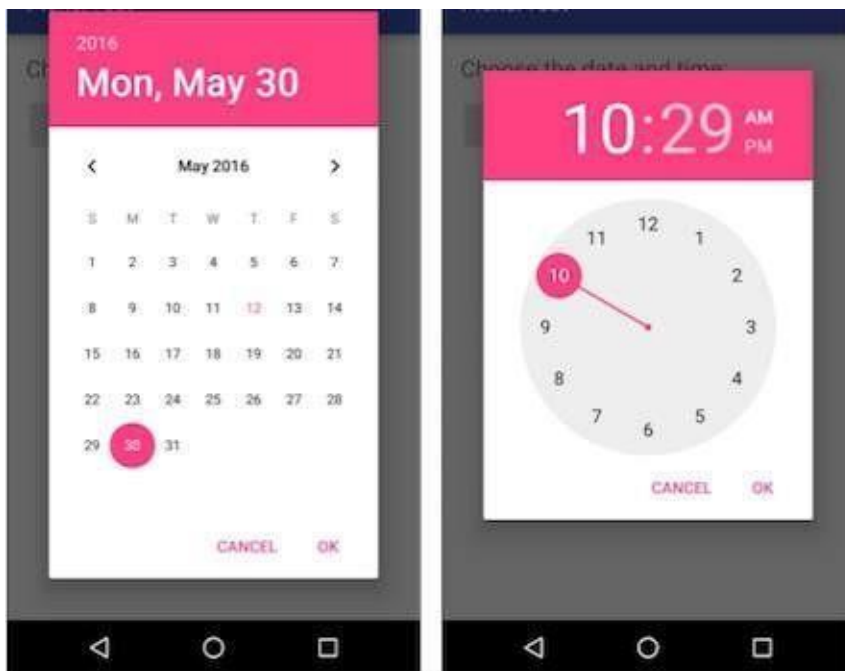
## Displaying the dialog

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 implementing a picker, it's best to utilize a `DialogFragment`, a subclass of `Fragment`. `DialogFragment` presents a floating dialog window above the activity's window. A fragment represents a distinct behavior or section of the user interface within an activity, akin to a self-contained "mini-activity" with its own lifecycle. Fragments can handle their input events and be added or removed while the main activity is running. You can incorporate multiple fragments in a single activity to create a multi-pane user interface or reuse a fragment in various activities. For further details on fragments, refer to the Fragments section in the API Guide.

Utilizing fragments for pickers offers the advantage of segregating the code sections responsible for managing the date and time after users make their selections. Additionally, `DialogFragment` can effectively oversee the dialog's lifecycle.

**Tip:** Another advantage of employing fragments for pickers is the flexibility to accommodate diverse layout configurations. This allows you to present a straightforward dialog on smaller, handset-sized displays, or seamlessly integrate it as a component within a larger layout on larger displays.

## Adding a fragment

To add a fragment for the date picker, create a blank fragment (**DatePickerFragment**) without a layout XML, and without factory methods or interface callbacks:

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

## 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:

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

Android Studio automatically adds the following in the import block at the top:

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

2. Android Studio also shows a red light bulb icon in the left margin, prompting you to implement methods. Click the icon and, with `onDateSet` already selected and the "Insert `@Override`" option checked, click **OK** to create the empty `onDateSet()` method.

Android Studio then automatically adds the following in the import block at the top:

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

3. Replace `onCreateView()` with `onCreateDialog()` :

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

When you extend `DialogFragment`, you should override the `onCreateDialog()` callback method, rather than `onCreateView`. You use your version of the callback method to initialize the `year`, `month`, and `day` for the date picker.

## Setting the defaults and returning the picker

To set the default date in the picker and return it as an object you can use, follow these steps:

1. Add the following code to the `onCreateDialog()` method to set the default date for the picker:

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

As you enter `Calendar`, you are given a choice of which Calendar library to import. Choose this one:

```
import java.util.Calendar;
```

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`, `HOURL`, and so on. `Calendar` is locale-sensitive, and its class method `getInstance()` returns a `Calendar` object whose calendar fields have been initialized with the current date and time.

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

```
return new DatePickerDialog(getActivity(), this, year, month, day);
```

## Showing the picker

In the Main Activity, you need to create a method to show the date picker. Follow these steps:

1. Create a method to instantiate the date picker dialog fragment:

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

2. You can then use `showDatePickerDialog()` with the `android:onClick` attribute for a button or other input control:

```
<Button
    android:id="@+id/button_date"
    ...
    android:onClick="showDatePickerDialog"/>
```

## Processing the user's picker choice

The `onDateSet()` method is automatically called when the user makes a selection in the date picker, so you can use this method to do something with the chosen date. Follow these steps:

1. To make the code more readable, change the `onDateSet()` method's parameters from `int i`, `int i1`, and `int i2` to `int year`, `int month`, and `int day`:

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

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

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

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

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

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

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

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

5. Add the following after the above statement to display a Toast message:

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

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

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

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

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

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

## Using the same procedures for the time picker

Follow the same procedures outlined above for a date picker:

1. Add a blank fragment called **TimePickerFragment** that extends `DialogFragment` and implements

```
TimePickerDialog.OnTimeSetListener :
```

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

2. Add with `@Override` a blank `onTimeSet()` method:

Android Studio also shows a red light bulb icon in the left margin, prompting you to implement methods. Click the icon and, with **onTimeSet** already selected and the "Insert `@Override`" option checked, click **OK** to create the empty `onTimeSet()` method. Android Studio then automatically adds the following in the import block at the top:

```
import android.widget.TimePicker;
```

3. Use `onCreateDialog()` to initialize the time and return the dialog:

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

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

4. Show the picker: Open **MainActivity** and create a method to instantiate the date picker dialog fragment:

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

Use `showDatePickerDialog()` with the `android:onClick` attribute for a button or other input control:

```
<Button
    android:id="@+id/button_date"
    ...
    android:onClick="showDatePickerDialog"/>
```

5. Create the `processTimePickerResult()` method in **MainActivity** to process the result of choosing from the time picker:

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

6. Use `onTimeSet()` to get the time and pass it to the `processTimePickerResult()` method in `MainActivity`:

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

You can read all about setting up pickers in [Pickers](#).

## Recognizing gestures

A touch gesture takes place when a user positions one or more fingers on the touch screen, and your app interprets this pattern of touches as a specific gesture, like a long touch, double-tap, fling, or scroll.

Android offers a range of classes and methods to facilitate both the creation and detection of these gestures. While your app should not rely on touch gestures for fundamental functionalities (as they may not be accessible to all users in all situations), integrating touch-based interaction can significantly enhance the utility and attractiveness of your app.

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.

1. 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());
    }
    ...
}
```

When you instantiate a `GestureDetectorCompat` object, one of the parameters it takes is a class you must create

— `MyGestureListener` in the above snippet—that does one of the following:

- implements the `GestureDetector.OnGestureListener` interface, to detect all standard gestures, or
- extends the `GestureDetector.SimpleOnGestureListener` class, which you can use to process only a few gestures by overriding the methods you need. Some of the methods it provides include `onDown()`, `onLongPress()`, `onFling()`, `onScroll()`, and `onSingleTapUp()`.

1. Create the class `MyGestureListener` as a separate activity (**`MyGestureListener`**) to extend `GestureDetector.SimpleOnGestureListener`, and 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;
    }
}
```

2. To intercept touch events, override the `onTouchEvent()` callback of the `GestureDetectorCompat` class in **`MainActivity`**:

```
@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(s), and ends by capturing the final 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 of every interaction. 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 touch screens, 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.

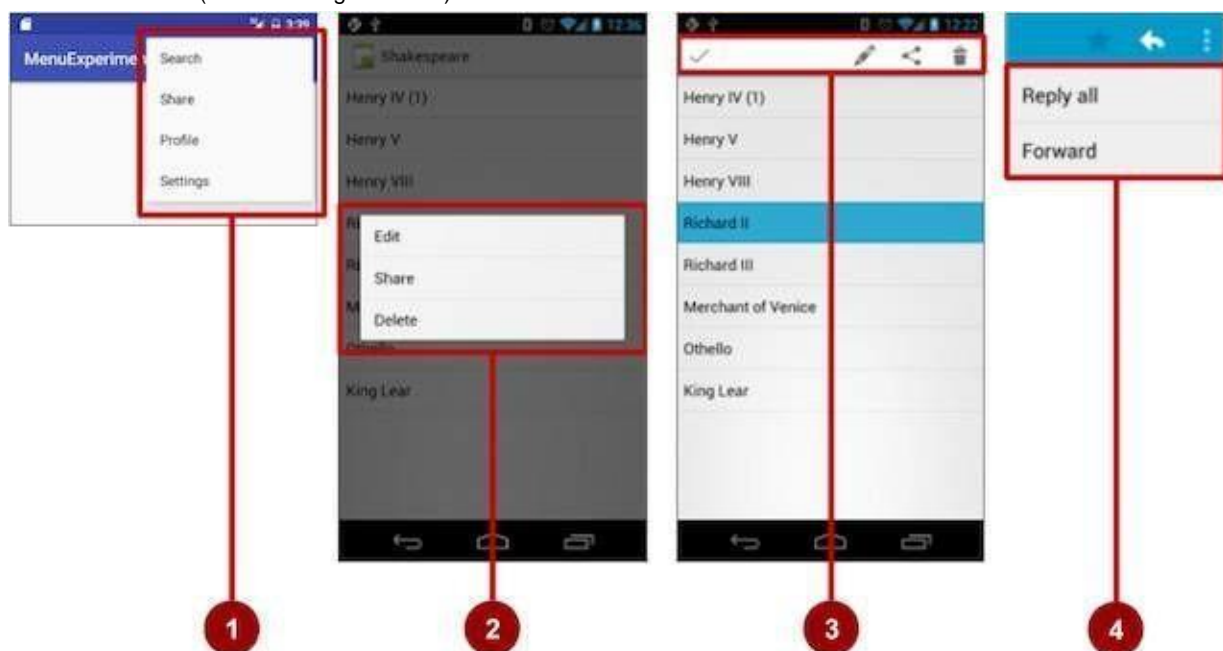
## 4.2: Menus

### Contents:

- Types of menus
- The app bar and options menu
- Contextual menu
- Popup menu

### Types of menus

A menu is a set of options the user can select from to perform a function, such as searching for information, saving information, editing information, or navigating to a screen. Android offers the following types of menus, which are useful for different situations (refer to the figure below):



1. **Options menu:** Appears in the app bar and provides the primary options that affect using the app itself. Examples of menu options: **Search** to perform a search, **Bookmark** to save a link to a screen, and **Settings** to navigate to the Settings screen.
2. **Context 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(s). 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**.

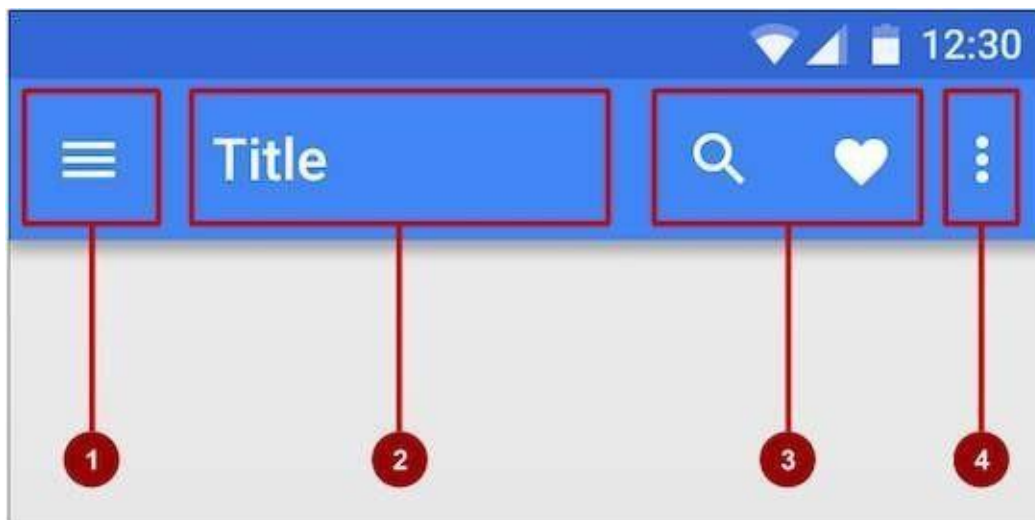
### The app bar and options menu

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 in a CoordinatorLayout root view group at the top of the view hierarchy.

The app bar by default shows the app title, or the name defined in `AndroidManifest.xml` by the `android:label` attribute for the activity. It may also include the **Up** button for navigating up to the parent activity, which is described in the next chapter.

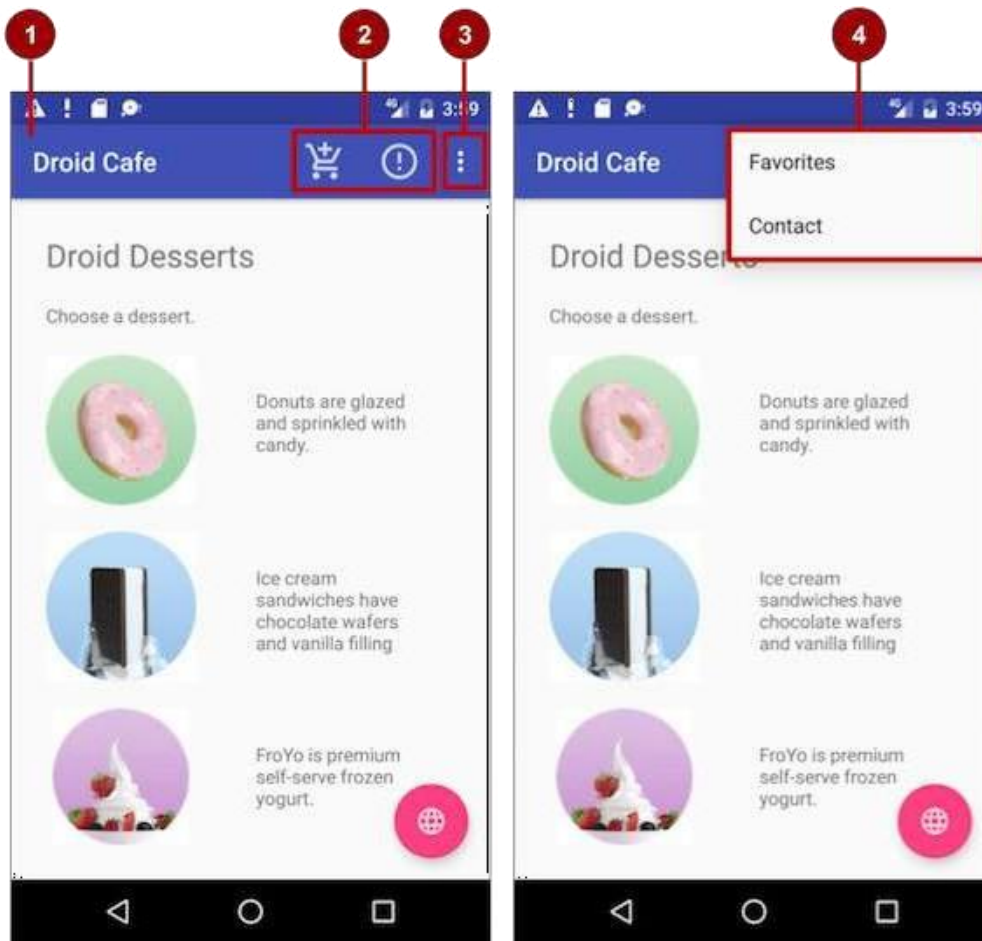
The *options menu* in the app bar provides navigation to other activities in the app, or the primary options that affect using the app itself — but not ones that perform an action on an element on the screen. For example, your options menu might provide the user choices for navigating to other activities,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

The options menu appears in the right corner of the app bar. The app bar is split into four different functional areas that apply to most apps:



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.

However, as features were added to the native ActionBar over various Android releases, the native ActionBar behaves differently depending on the version of Android running on the device. For this reason, if you are adding an options menu, you should use the v7 appcompat support library's Toolbar as an app bar. Using the Toolbar makes it easy to set up an app bar that works on the widest range of devices, and also gives you room to customize your app bar later on as your app develops. Toolbar includes the most recent features, and works for any device that can use the support library.

In order to use Toolbar as the app bar (rather than the default ActionBar) for an activity, do one of the following:

- Start your project with the Basic Activity template, which implements the Toolbar for the activity as well as a rudimentary options menu (with one item, **Settings**). You can skip this section.
- Do it yourself, as shown in this section:
  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 the appcompat support library (current version is v7) for the Toolbar class, and the design library for the NoActionBar themes, to your project:

1. Choose **Tools > Android > SDK Manager** to check that the Android Support Repository is installed; if it is not, 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:

```
compile 'com.android.support:appcompat-v7:23.4.0'
compile 'com.android.support:design:23.4.0'
```

**Note:** Update the version numbers for dependencies if necessary. If the version number you specified is lower than the currently available library version number, Android Studio will warn you ("a newer version of com.android.support:design is available"). 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** in the **res > values** directory.

**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 so that the app bar is visible and stands out against the background. Follow these steps:

1. Open the **styles.xml** file. You should already have the following in the file:

```
<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
    ...
</resources>
```

`AppTheme` "inherits"—takes on all the styles—from a parent them 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**.

2. 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:



```
<resources>
  <!-- Base application theme. -->
  <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    ...
  </style>

  <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" />
  ...
</resources>
```

3. In the `AndroidManifest.xml` file, 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:theme="@style/AppTheme.NoActionBar">
</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's 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:

- `AppBarLayout` must be a direct child within a `CoordinatorLayout` root view group, and `Toolbar` must be a direct child within `AppBarLayout`, as shown below:

```
<android.support.design.widget.CoordinatorLayout
  ... >
  <android.support.design.widget.AppBarLayout
    ...>
    <android.support.v7.widget.Toolbar
      ...
    />
  </android.support.design.widget.AppBarLayout>
  ...
</android.support.design.widget.CoordinatorLayout>
```

- Position the `Toolbar` at the top of the activity's layout, since you are using it as an app bar.
- `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`) as follows:
  - In the same layout file for the activity (as in `activity_main.xml`)
  - In a separate layout file, such as `content_main.xml`, which you can then add to the activity's layout file with an `include` statement:

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

- You need to set the content sibling's scrolling behavior, as shown below with the `RelativeLayout` group, to be an instance of `AppBarLayout.ScrollingViewBehavior`:

```
<RelativeLayout
    ...
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    ... >

</RelativeLayout>
```

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. This string resource represents the following string, which is defined in the `values.xml` file that should not be edited:

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

This behavior is defined by the `AppBarLayout.ScrollingViewBehavior` class. This behavior should be used by Views which can scroll vertically—it supports nested scrolling to automatically scroll any `AppBarLayout` siblings.

## 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 MyActivity extends AppCompatActivity {
    ...
}
```

2. In the activity's `onCreate()` method, call the activity's `setSupportActionBar()` method, and pass the activity's toolbar (assuming the `Toolbar` element's id is `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 = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
}
```

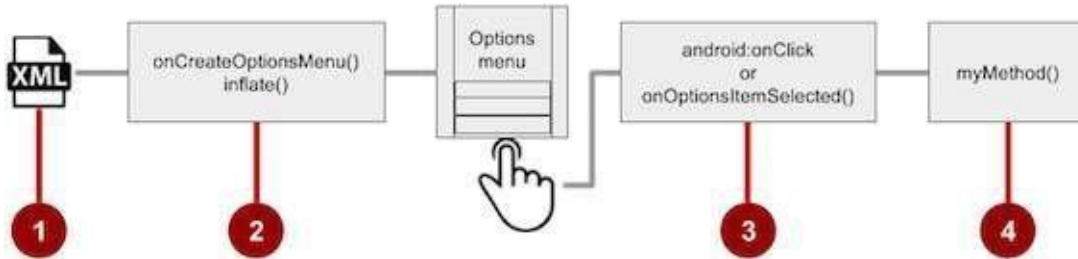
The activity now shows the app bar. By default, the app bar contains just the name of the app.

## Adding the options menu

Android provides a standard XML format to define options menu items. Instead of building the menu in your activity's 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 or fragment.

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 or fragment to inflate the menu.
3. **Handling menu item clicks.** Menu items are views, 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 was clicked, which makes your code easier to understand.
4. **Performing actions.** Create a method to perform an action for each options menu item.

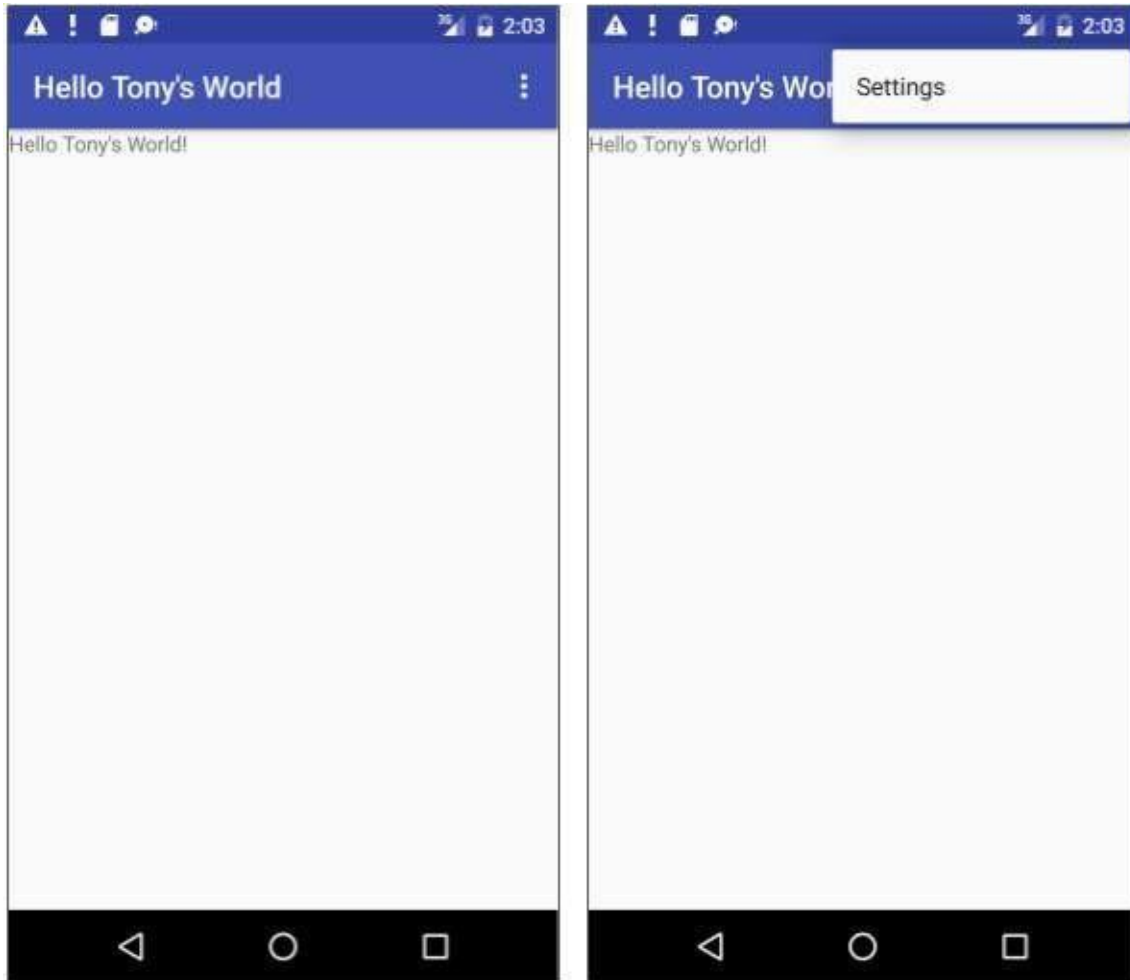
## Creating an XML resource for the menu

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

1. Click the **res** directory, and choose **File > New > Android resource directory**, choose **menu** in the Resource type drop-down menu, and click **OK**.
2. Click the new **menu** directory, and choose **File > New > Menu resource file**, enter the name of the file as **menu\_main** or something similar, and click **OK**. The new `menu_main.xml` file now resides within the menu directory.
3. Open the `menu_main.xml` file (if not already open), and click the **Text** tab next to the Design tab at the bottom of the pane to show the text of the file.
4. Add the first options menu item using the `<item ... />` tag. In this example, the item is **Settings**:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    ...>
    <item
        android:id="@+id/action_settings"
        android:title="@string/settings" />
</menu>
```

After setting up and inflating the XML resource in the activity or fragment code, 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 options menu items are defined: `@string/settings` (**Settings**) and `@string/action_order` (**Order**):

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

Within each `<item ... />` tag you can add the attributes to define how the menu item appears, such as the order of its appearance relative to the other items, and whether the item can appear as an icon in the app bar. Any item you set to *not* appear in the app bar (or that can't fit in the app bar given the display orientation) is placed in order in the overflow menu).

Whenever possible, you want to show the most frequently used actions using icons in the app bar so that the user can click them without having to first click the overflow button.

### Adding icons for menu items

To specify icons for actions, you need to first add the icons as image assets to the **drawable** folder by following these steps (see Image Asset Studio for a complete description):

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

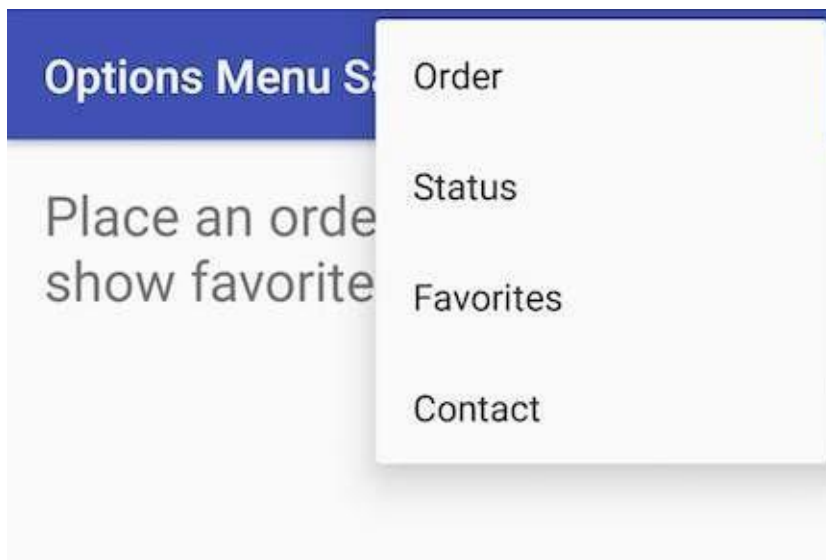
```
<item
    android:id="@+id/action_order"
    android:icon="@drawable/ic_order_white"
    android:title="@string/action_order"/>
```

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



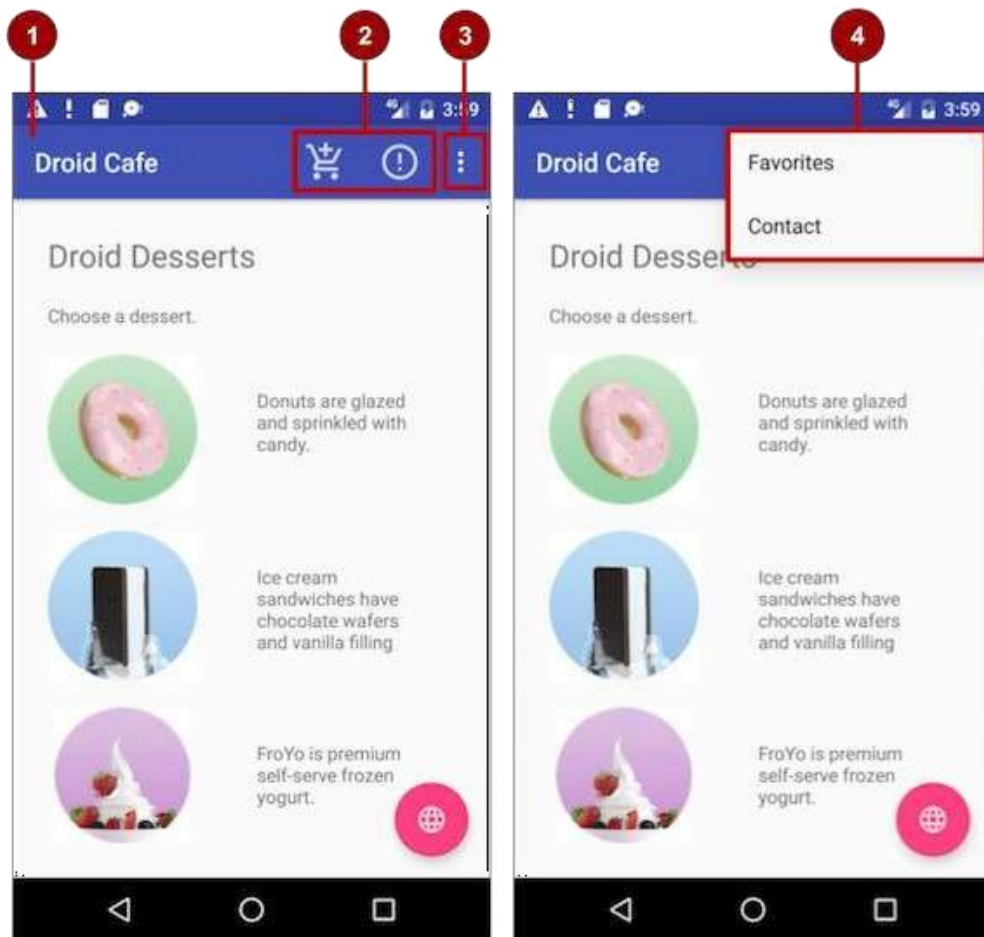
**Note:** While the numbers 1, 2, 3, and 4 would also work in the above example, the numbers 10, 20, 40, and 100 leave room for additional menu items to be added later between them.

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, and 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. The title text appears anyway if the item appears in the overflow menu, so this attribute is used primarily to include the title with the icon in the app bar.

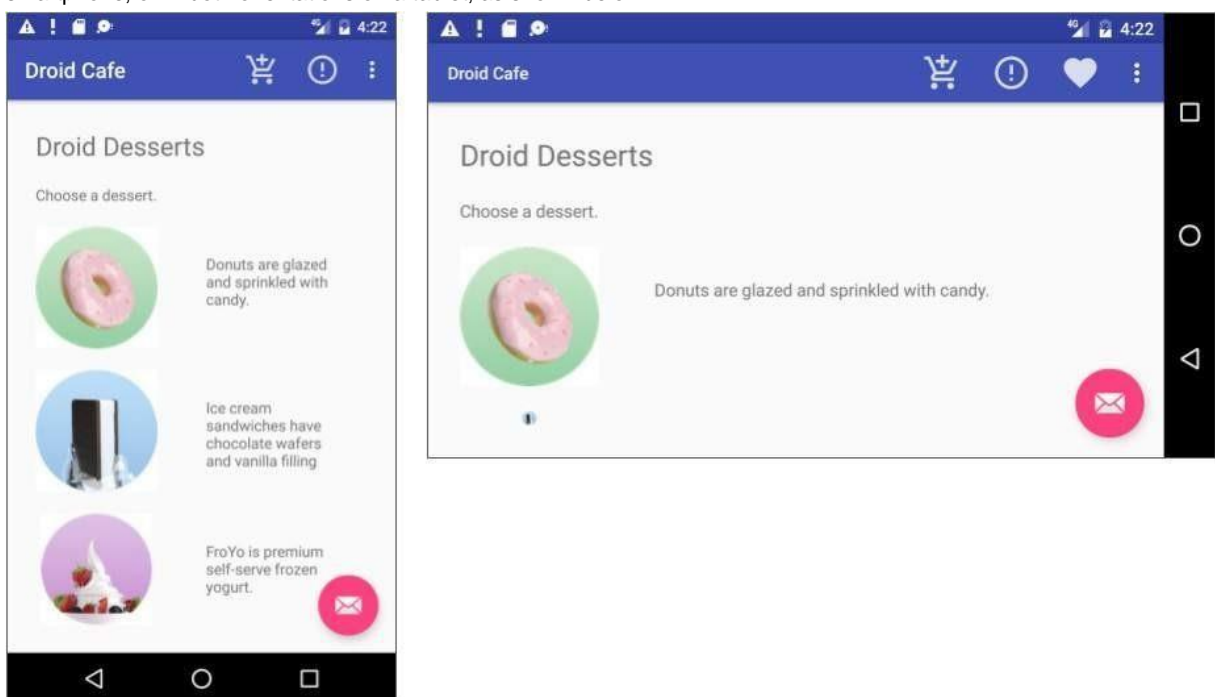
For example, the following menu item's icon appears in the app bar only if there is room for it:

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



In the above figure:

1. **Options menu action icons.** The first two options 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** and **Contact**. **Favorites** (the heart icon) 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 using the `onCreateOptionsMenu()` method (with the `Override` annotation) with 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 as a parameter the resource `id` for an XML layout resource to load ( `R.menu.menu_main` in the following example), and 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 of the Activities class can handle all the menu item clicks in one place, and determine which menu item was clicked, which makes your code easier to understand. The Basic Activity template provides an implementation of the `onOptionsItemSelected()` method with a `switch case` block to call the appropriate method (such as `showOrder`) based on the menu item's `id`, which you can retrieve using the `getItemId()` method of the Adapter class:

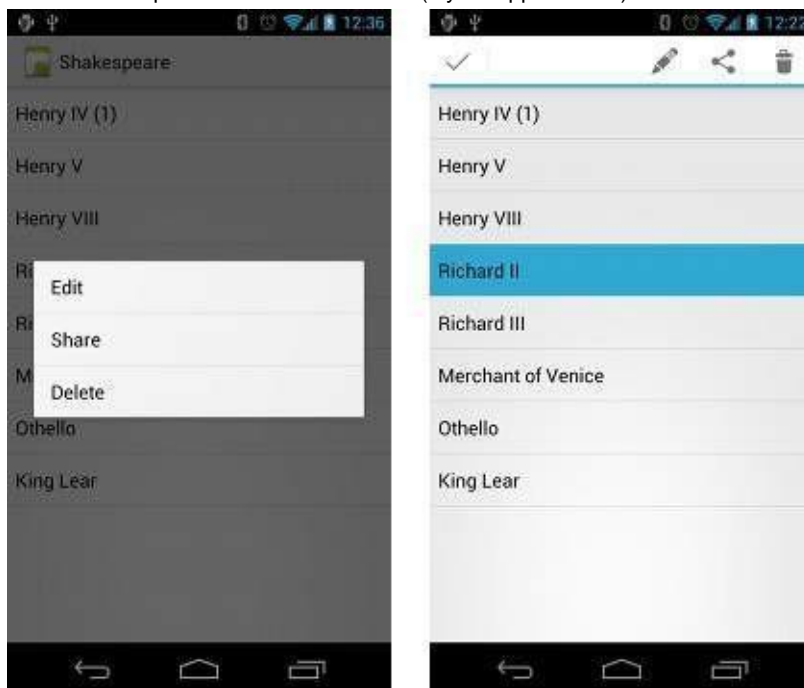
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_order:
            showOrder();
            return true;
        case R.id.action_status:
            showStatus();
            return true;
        case R.id.action_contact:
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

## Contextual menu

Use a *contextual menu* to allow users to take an action on a selected view. You can provide a context menu for any View, but they are most often used for items in a RecyclerView, GridView, or other view collections in which the user can perform direct actions on each item.

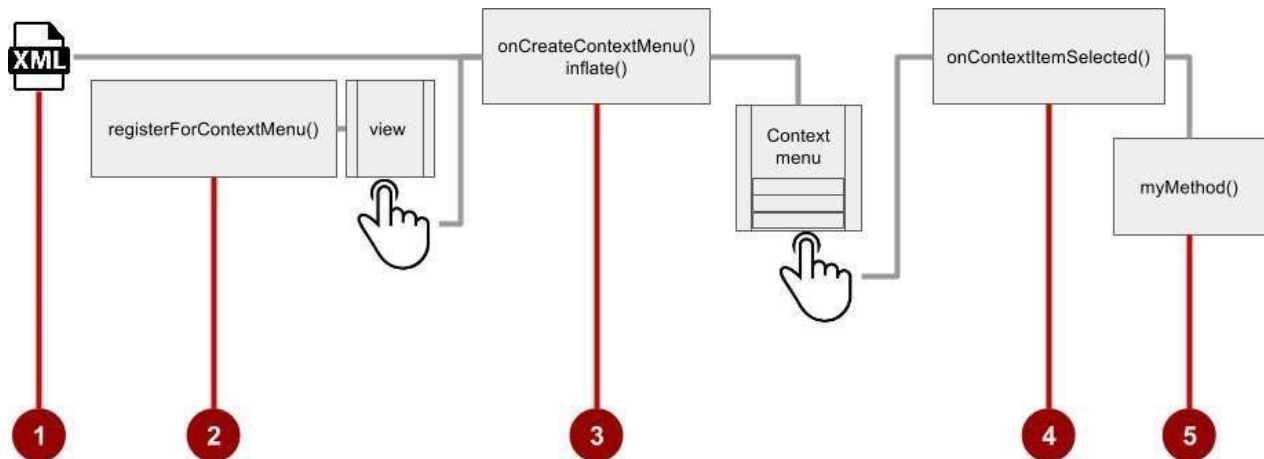
Android provides two kinds of contextual menus:

- 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 element on the screen. It is typically used to modify the view element or use it in some fashion. For example, a context menu might include **Edit** to edit a view element, **Delete** to delete it, and **Share** to share it over social media. Users can perform a contextual action on one view element 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 the selected view element(s). 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 floating context menu, modified to include registering (associating) the context menu with a view:



Follow these steps to create a floating context menu for one or more view elements (refer to figure above):

1. Create an XML menu resource file for the menu items, and assign appearance and position attributes (as described in the previous section).
2. Register a view to the context menu using the `registerForContextMenu()` method of the Activity class.
3. Implement the `onCreateContextMenu()` method in your activity or fragment to inflate the menu.
4. Implement the `onContextItemSelected()` method in your activity or fragment to handle menu item clicks.
5. 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. Use a suitable name for the file, such as `menu_context`. Add the context menu items (in this example, the menu items are **Edit**, **Share**, and **Delete**):

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

<item
    android:id="@+id/context_share"
    android:title="@string/share"
    android:orderInCategory="20"/>

<item
    android:id="@+id/context_delete"
    android:title="@string/delete"
    android:orderInCategory="30"/>
```

## Registering a view to the context menu

Register a view to the context menu by calling the `registerForContextMenu()` method and passing it the view. Registering a context menu for a view sets the `View.OnCreateContextMenuListener` on the view to this activity, so that `onCreateContextMenu()` will be called when it is time to show the context menu. (You implement `onContextItemSelected` in the next section.)

For example, in the `onCreate()` method for the activity, add the `registerForContextMenu()` statement:

```
...
// Registering the context menu to the text view of the article.
TextView article_text = (TextView) 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 or fragment. This 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 above code:

- 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 `v`, which could be a RecyclerView or a GridView. If you are registering a RecyclerView or GridView, you would instantiate a `ContextMenu.ContextMenuInfo` object to provide the 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 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).

## 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 or fragment 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()` below for the **Edit** and **Share** menu items. For example:

```
@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 example 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 text view, the floating context menu appears and the user can click a menu item.

3G  1:39

## Scrolling Text

### Beatles Anthology Vol. 1

#### Behind That Locked Door: Beatles Rarities!

In a vault deep inside Abbey Road Studios in London — protected by an unmarked, triple-locked, police-

a  
u  
1  
fo  
re  
v  
th

Edit

Share

Delete

This volume starts with the first new Beatle song, “Free as a Bird” (based on a John Lennon demo, found only on the bootleg *The Lost Lennon Tapes Vol. 28*, and covers the very earliest historical recordings, outtakes from the first albums, and live recordings from early concerts and BBC Radio sessions.

© 2015 Apple Inc. All rights reserved.



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

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

## 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 the View.

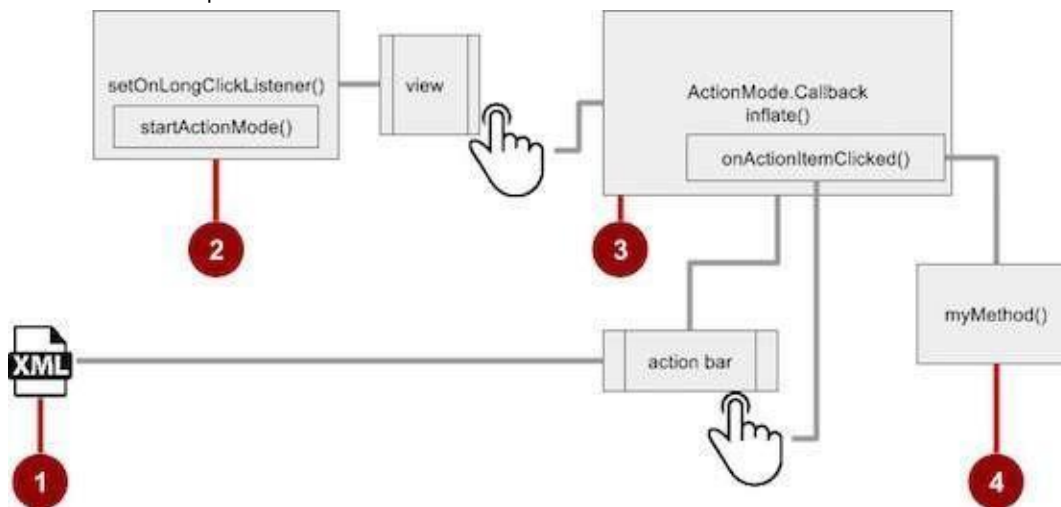
`ActionMode` represents a user interface (UI) mode for providing alternative interaction, replacing parts of the normal UI until finished. For example, text selection is implemented as an `ActionMode`, as are contextual actions that work on a selected item on the screen. Selecting a section of text or long-clicking a view triggers `ActionMode`.

While this mode is enabled, the user can select multiple items (if your app allows it), deselect items, and continue to navigate within the activity. The action mode is disabled and the contextual action bar disappears when the user deselects all items, presses the Back button, or taps **Done** (left-arrow icon) on the left side of the bar.

Follow these steps to create a contextual action bar (refer to 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 to the view that should trigger the contextual action bar using the `setOnLongClickListener()` method. 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 a previous section. Use a suitable name for the file, such as `menu_context`. Add icons for the context menu items (in this example, the menu items are **Edit**, **Share**, and **Delete**). 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="@string/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 class (such as **MainActivity**) using the activity's `onCreate()` method. Follow these steps:

1. Declare the member variable `mActionMode` in the class definition for the activity:

```
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.

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



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    View articleView = findViewById(article);
    articleView.setOnLongClickListener(new View.OnLongClickListener() {
        ...
        // Add method here to start ActionMode after long-click.
        ...
    });
}
```

## 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 action mode 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 class (such as **MainActivity**) to implement the interface:

```
public ActionMode.Callback mActionModeCallback = new
    ActionMode.Callback() {
    ...
    // Add code to create action mode here.
    ...
}
```

2. Add the `onCreateActionMode()` code within the brackets of the above method to create action mode (the full code is provided at the end of this section):

```
@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).

3. Add the `onOptionsItemSelected()` method with your handlers for each menu item:

```
@Override
public boolean onOptionsItemSelected(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 can create in the same activity. After the action is picked, you use the `mode.finish()` method to close the contextual action bar.

4. 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.

5. Review 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.

1. To start `ActionMode`, add the `onLongClick()` method within the brackets of the `setOnLongClickListener` method in

`onCreate()` :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    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`).

2. Review 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);

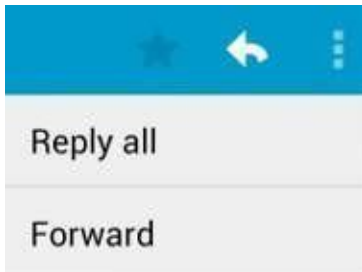
    // set up the contextual action bar listener
    View articleView = findViewById(article);
    articleView.setOnLongClickListener(new View.OnLongClickListener() {
        // Called when the user long-clicks on articleView
        public boolean onLongClick(View view) {
            if (mActionMode != 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.

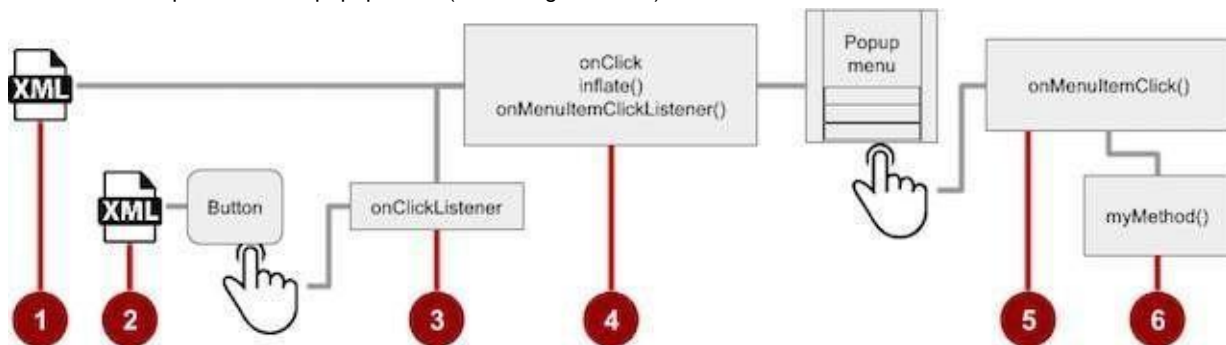
A popup menu is typically used to provide an overflow of actions (similar to the overflow action icon for the options menu) or the second part of a two-part command. Use a popup menu for extended actions that relate to regions of content in your activity. Unlike a context menu, a popup menu is anchored to a Button (View), 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 action bar.



## Creating a pop-up 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 button.
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/ic_action_popup"/>
```

## Assigning onClickListener to the button

1. Create a member variable ( `mButton` ) in the activity's class definition:

```
public class MainActivity extends AppCompatActivity {  
    private ImageButton mButton;  
    ...  
}
```

2. In the `onCreate()` method for the same activity, assign the `ImageButton` in the layout to the member variable, and assign `onClickListener()` to the button:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    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`:

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

After instantiating a `PopupMenu` object ( `popup` in the above example), the method uses the `MenuInflater` class and its `inflate()` method, which takes as parameters:

- The resource `id` for an XML layout resource to load ( `menu_popup` in the example above)  
The Menu to inflate into ( `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, and finish the method with `popup.show` to show the popup menu:

```

        public boolean onOptionsItemSelected(MenuItem item) {
            // Perform action here
            return true;
        }
    });
    popup.show(); //show the popup menu
}
}); // close the setOnClickListener method

```

Putting these pieces together, the entire `onCreate()` method should now look like this:

```

private ImageButton mButton;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ...
    // popup button setup
    mButton = (ImageButton) findViewById(R.id.button_popup);
    mButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            //Creating the instance of PopupMenu
            PopupMenu popup = new PopupMenu(MainActivity.this, mButton);
            //Inflating the Popup using xml file
            popup.getMenuInflater().inflate(R.menu.menu_popup, popup.getMenu());

            //registering popup with OnMenuItemClickListener
            popup.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
                public boolean onOptionsItemSelected(MenuItem item) {
                    // Perform action here
                    return true;
                }
            });
            popup.show();//show the popup menu
        }
    }); //close the setOnClickListener method
}

```



## 4.3: Screen 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

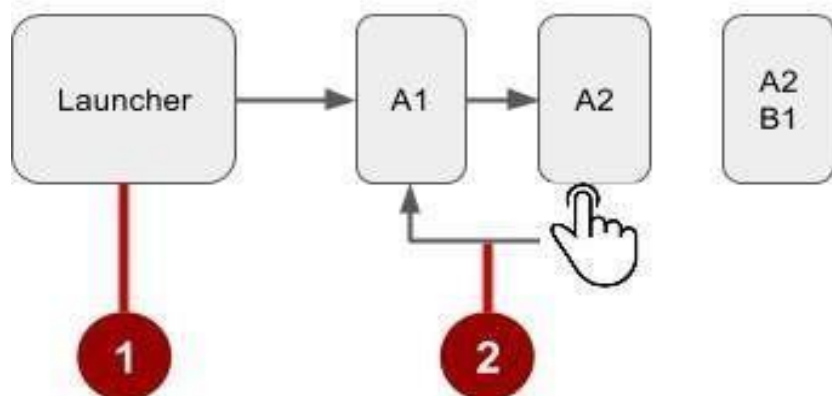
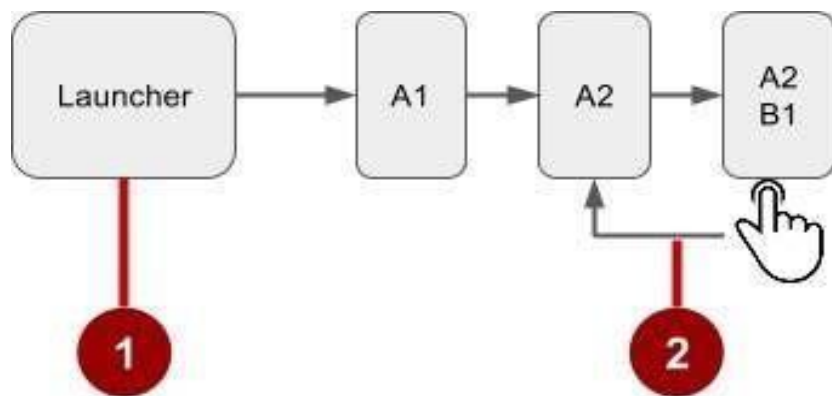
### Providing users with a path through your app

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

In many cases you will need several different paths through your app that offer the following types of navigation:

- *Back* navigation: Users can navigate back to the previous screen using the Back button.
- *Hierarchical* navigation: Users can navigate through a hierarchy of screens organized with a *parent* screen for every set of *child* screens.

### Back-button navigation





In the above figure:

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.

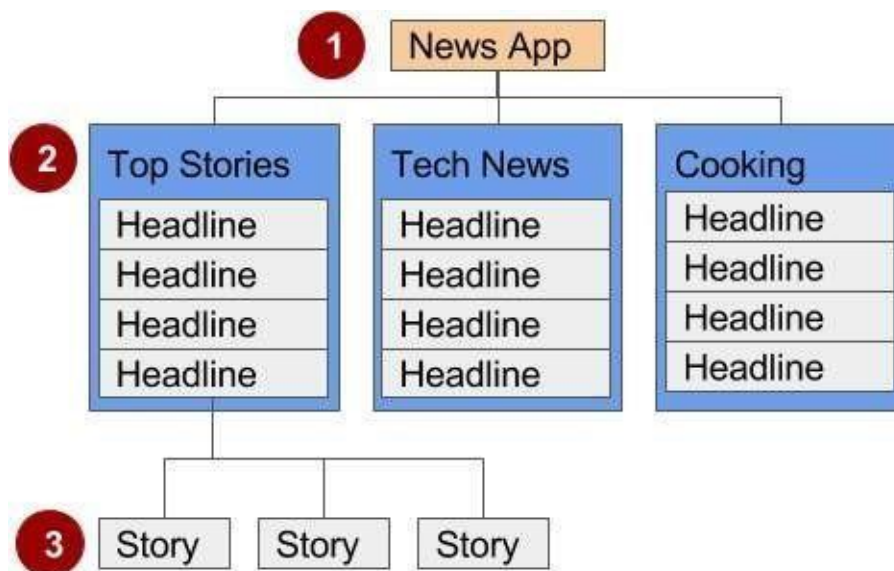
The `onBackPressed()` method of the Activity class is called whenever the activity detects the user's press of the Back key. The default implementation simply finishes the current activity, but you can override this to do something else:

```
@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.** 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.
2. **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.
3. **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 a 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 a main activity (as a parent screen) and then other activities or fragments to implement a hierarchy of screens within an app.

## Main activity with other activities

If the first-level child screen siblings have another level of child screens under them, you should implement the first-level screens as activities, so that their lifecycles are 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, and you implement a navigation pattern in the main activity to enable the user to go to other activities or fragments. 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 learned previously, the Android system initiates code in an Activity instance with callback methods that manage the activity's lifecycle for you. (A previous lesson covers the activity lifecycle; for more information, see "Managing the Activity Lifecycle" in the Training section of the Android Developer Develop guide.)

The hierarchy of parent and child activities is defined in the AndroidManifest.xml file. For example, the following defines

```
OrderActivity as a child of the parent MainActivity :
```

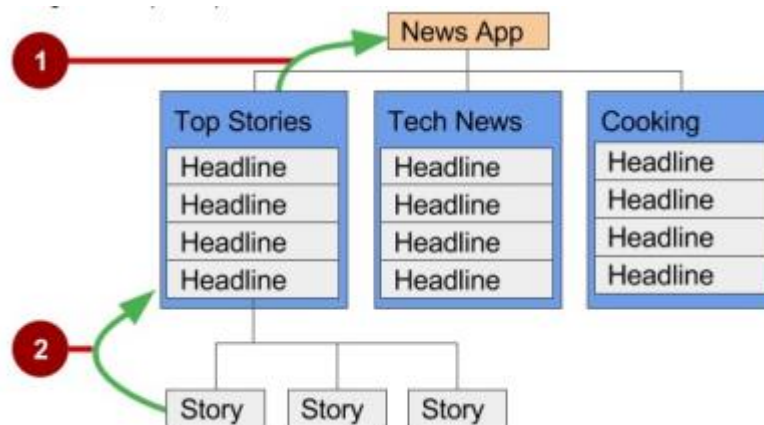
## Main activity with fragments

If the child screen siblings do *not* have another level of child screens under them, you can implement them as fragments. A Fragment represents a behavior or portion of a user interface 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 combine multiple fragments 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 showing 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 above figure:

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.

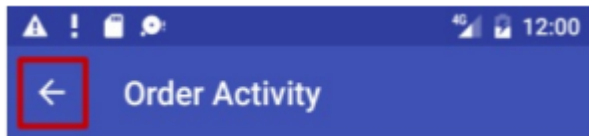
**Tip:** The **Back** button below the screen differs from the **Up** button. The **Back** button provides navigation to whatever screen you viewed previously. If you have several children screens that the user can navigate through using a lateral navigation pattern (as described later in this chapter), 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 "Menus" concept chapter for details on how to implement the app bar. To provide the **Up** button for a child screen activity, declare the activity's parent to be MainActivity in the AndroidManifest.xml file. You can also set the to a title for the activity screen, such as "Order Activity" (extracted into the string resource in the code below). Follow these steps to declare the parent in AndroidManifest.xml:

1. Open **AndroidManifest.xml**.
2. Change the activity element for the child screen activity (in this example, `OrderActivity`) to the following:

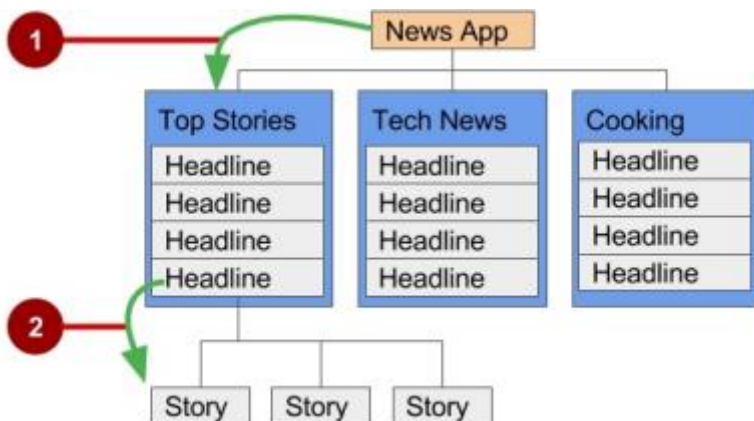
```
<activity android:name=".OrderActivity"
    android:label="@string/title_activity_order"
    android:parentActivityName=
        "com.example.android.optionsmenuorderactivity.MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity"/>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

The child ("Order Activity") 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.



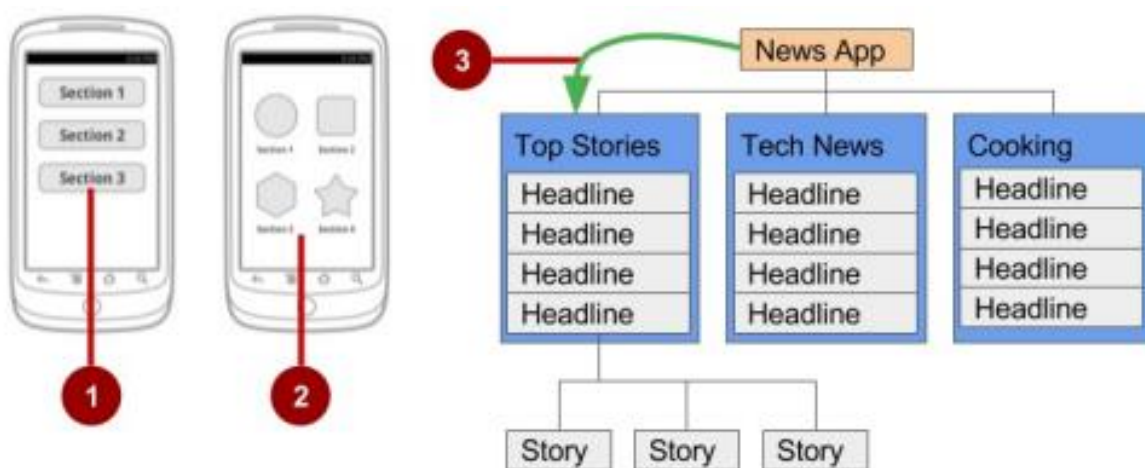
In the above figure:

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.



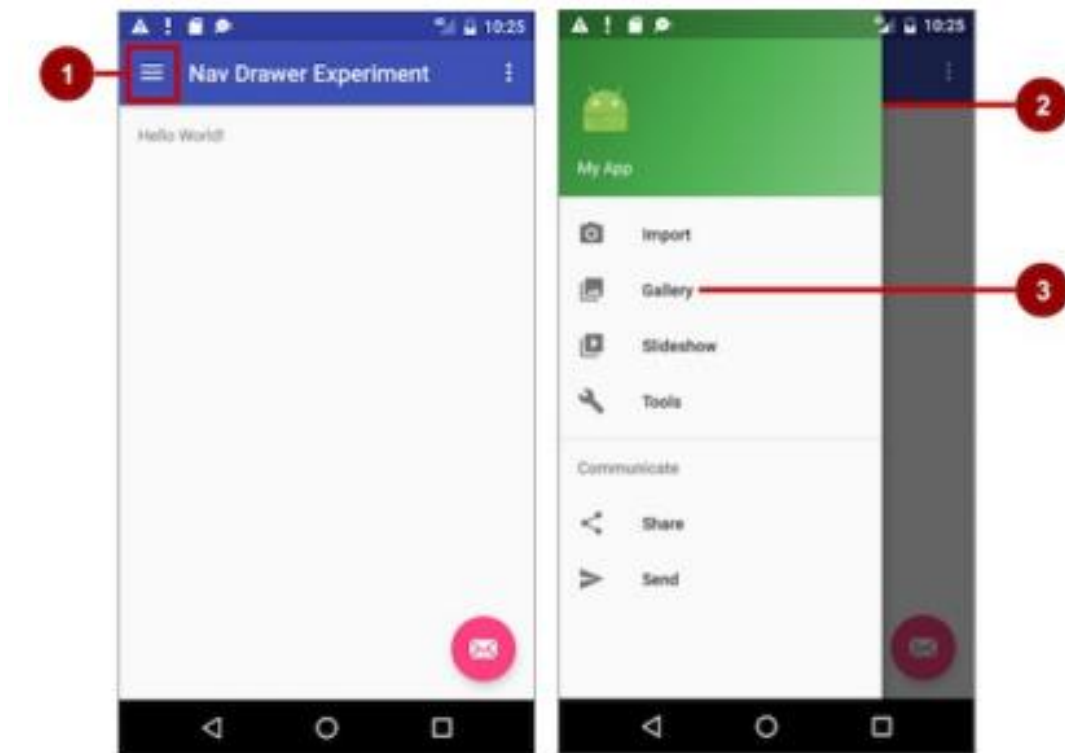
In the figure above:

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 above figure:

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, labelled 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 activities or fragments in an app. It can display many navigation targets at once—for example, it could 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 do the following:

1. Create the following layouts:
  - A navigation drawer as the activity layout's root view.
  - A navigation view for the drawer itself.
  - An app bar layout that will include a navigation icon button.
  - A content layout for the activity that displays the navigation drawer.
  - A layout for the navigation drawer header.
2. Populate the navigation drawer menu with item titles and icons.
3. Set up the navigation drawer and item listeners in the activity code.
4. Handle the navigation menu item selections.

## 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 view of your activity's layout. Inside the `DrawerLayout`, add one view that contains the main content for the screen (your primary layout when the drawer is hidden) and another view, typically a `NavigationView`, that contains the contents of the navigation drawer.

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

For example, the following layout uses:

A `DrawerLayout` as the root of the activity's layout in **activity\_main.xml**.

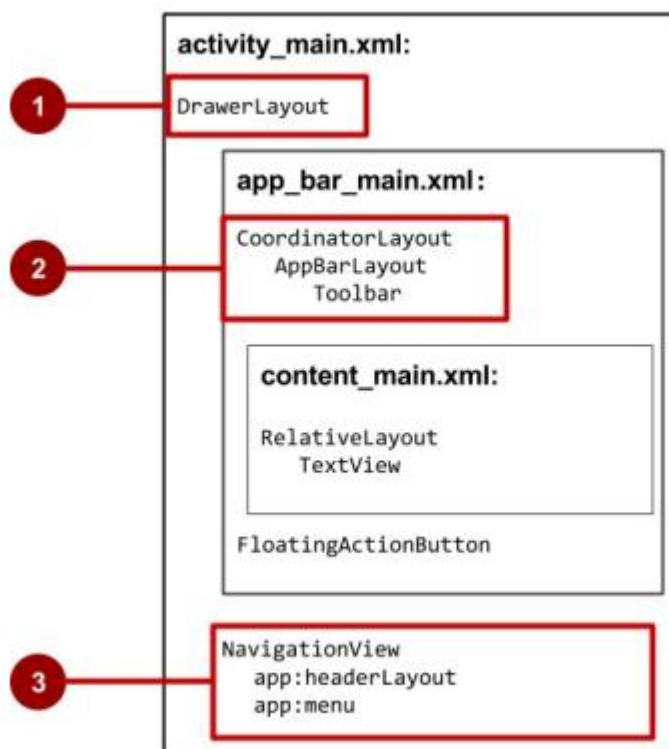
The main content of screen defined in the **app\_bar\_main.xml** layout file.

A `NavigationView` that represents a standard navigation menu that can be populated by a menu resource XML file.

Refer to the figure below that corresponds to this layout:

**activity\_main.xml:**

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





In the above figure:

1. DrawerLayout is the root view of the activity's layout.
2. The included `app_bar_main` (**`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 view 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 (RTL) languages, the drawer appears on the right rather than the left side.

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 view), 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 your content does not overlap with system-provided UI components.

## The navigation drawer header

The `app:headerLayout` 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 Views.

**Tip:** The header's height should be 160dp, which you should extract into a dimension resource (`nav_header_height`).

**`nav_header_main.xml`:**

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

**app\_bar\_main.xml:**

```
<?xml version="1.0" encoding="utf-8"?>
<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` layout uses a `CoordinatorLayout` as its root, and includes the `content_main` layout.
- The `app_bar_main` 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 above layout uses an `include` tag to include the `content_main` layout, which defines the layout of the main activity screen (`content_main.xml`). In the example layout below, the main activity screen shows a `TextView` that displays the string "Hello World!":

**content\_main.xml:**

```
<?xml version="1.0" encoding="utf-8"?>
<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` 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` layout is included in the `app_bar_main` layout, which is the first child.
- The `content_main` layout uses a `RelativeLayout` view group 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. This behavior is defined by the `AppBarLayout.ScrollingViewBehavior` class. This behavior should be used by Views which can scroll vertically—it supports nested scrolling to automatically scroll any `AppBarLayout` siblings.

## Populating the navigation drawer menu

The `NavigationView` in the `activity_main` 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**. The `<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 checked. Use for radio buttons.
all : All items can be checked. Use for checkboxes.
none : No items are checkable.
```

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

```
public class MainActivity extends AppCompatActivity
    implements NavigationView.OnNavigationItemSelectedListener {
    ...
}
```

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

2. Click the red warning bulb, choose **Implement methods**, and choose the **`onNavigationItemSelectedListener(item:MenuItem):boolean`** method.

Android Studio adds a stub for the method:

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

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

- Before setting up the navigation item listener, add code to the activity's `onCreate()` method to instantiate the `DrawerLayout` and `NavigationView` objects ( `drawer` and `navigationView` in the code below):

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    . . .
    DrawerLayout drawer = (DrawerLayout)
        findViewById(R.id.drawer_layout);

    ActionBarDrawerToggle toggle =
        new ActionBarDrawerToggle(this, drawer, toolbar,
            R.string.navigation_drawer_open,
            R.string.navigation_drawer_close);

    if (drawer != null) {
        drawer.addDrawerListener(toggle);
    }
    toggle.syncState();

    NavigationView navigationView = (NavigationView)
        findViewById(R.id.nav_view);
    if (navigationView != null) {
        navigationView.setNavigationItemSelectedListener(this);
    }
}
```

The above code instantiates an `ActionBarDrawerToggle`, which substitutes a special drawable for the activity's **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.appcompat`, 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 above code 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`'s 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 above code 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 following 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

Write code in the `onNavigationItemSelectedListener()` method stub to handle menu item selections. This method is called when an item in the navigation drawer menu is tapped.

It uses `if` 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).

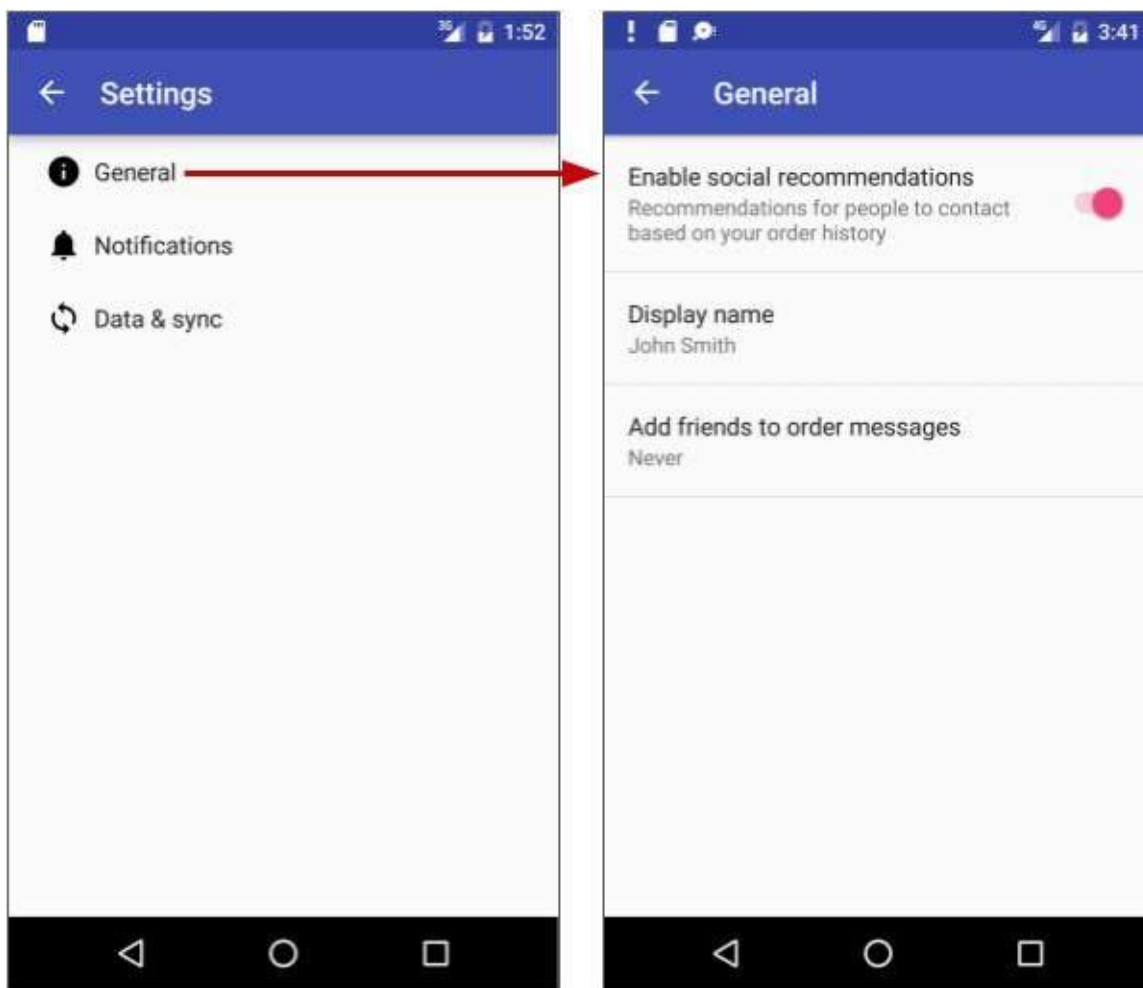
You learn all about `RecyclerView` in the next chapter.

## 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 a specific item. Descendant navigation is usually implemented by one of the following:

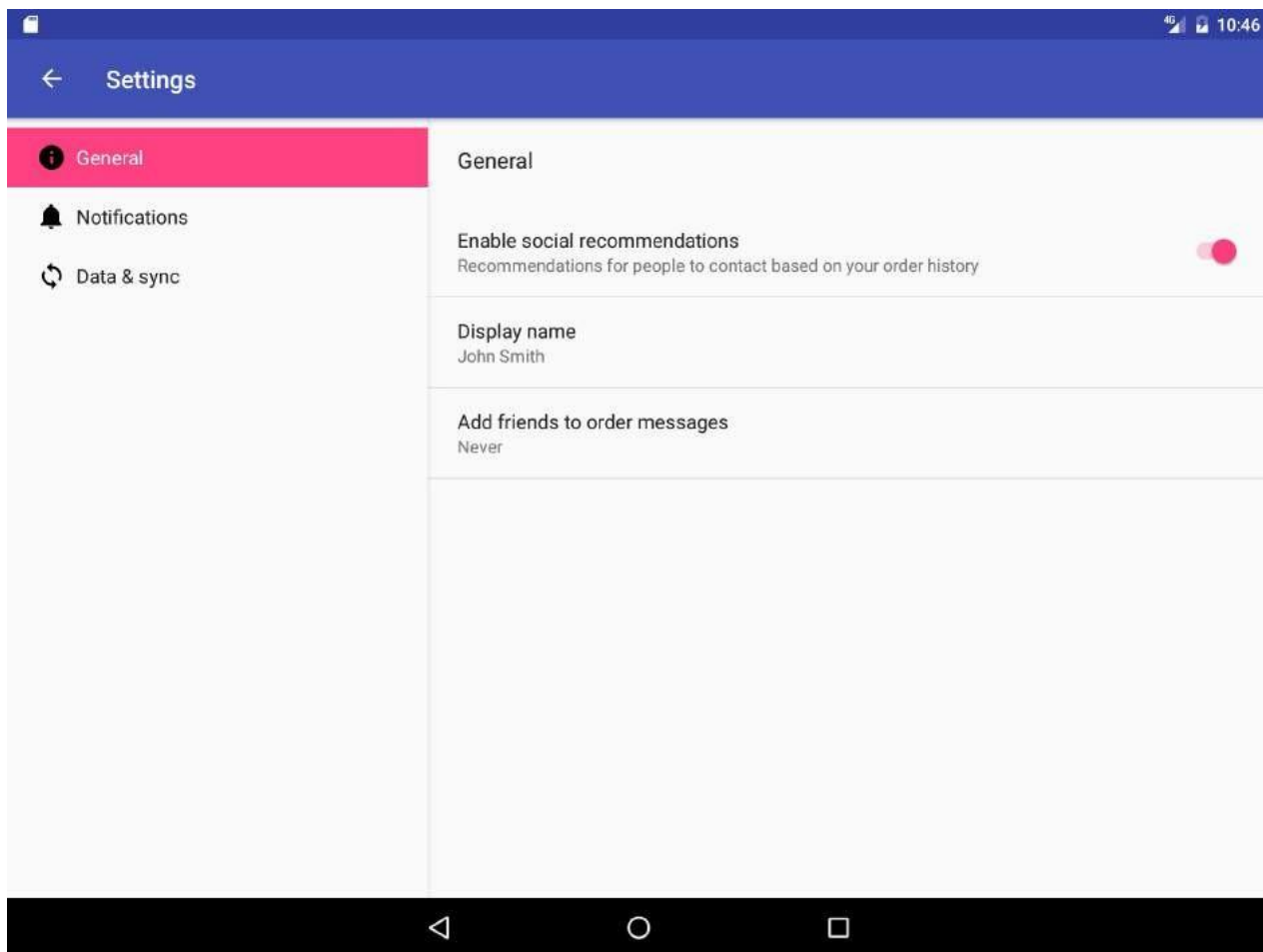
- Using an Intent that starts an activity representing the detail screen. For more information about Intents, see Intents and Intent Filters in the Android Developer Guide.
- When adding a Settings Activity, you can extend `PreferenceActivity` to create a two-pane master/detail layout to support large screens, and include fragments within the activity to replace the activity's 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 a subsequent chapter. For more information about using fragments, see Fragments in the Android Developer Guide.

Smartphones are best suited for displaying one screen at a time—such as 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.



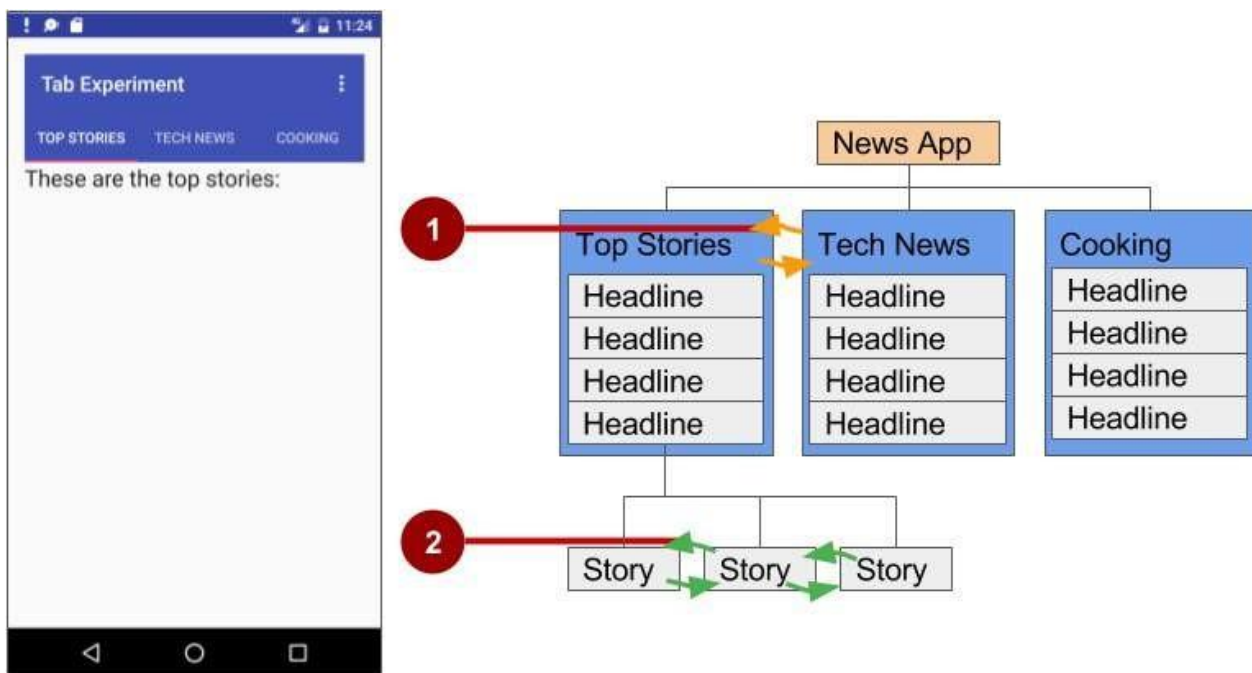


## Options menu in the app bar

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 icon for ancestral navigation, a nav icon for opening a navigation drawer, and a filter icon to filter page views. You learned how to set up the options menu and the app bar in a previous 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 above figure:

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 above figure, 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:

- Since 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:

1. Defining 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. Implementing a `Fragment` for each tab content screen. A *fragment* is a behavior or a portion of user interface within an activity. It's like a mini-activity within the main activity, with its own lifecycle. One benefit of using fragments for the tab content is that you can isolate the code for managing the tab content in the fragment. To learn about fragments, see [Fragments in the API Guide](#).

3. Adding 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. Creating an instance of the tab layout, and set the text for each tab.
5. Using `PagerAdapter` to manage screen ("page") views. Each screen is represented by its own fragment.
6. Setting a listener to determine which tab is tapped.

There are standard adapters for using fragments with the `ViewPager`:

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

## Defining tab layout

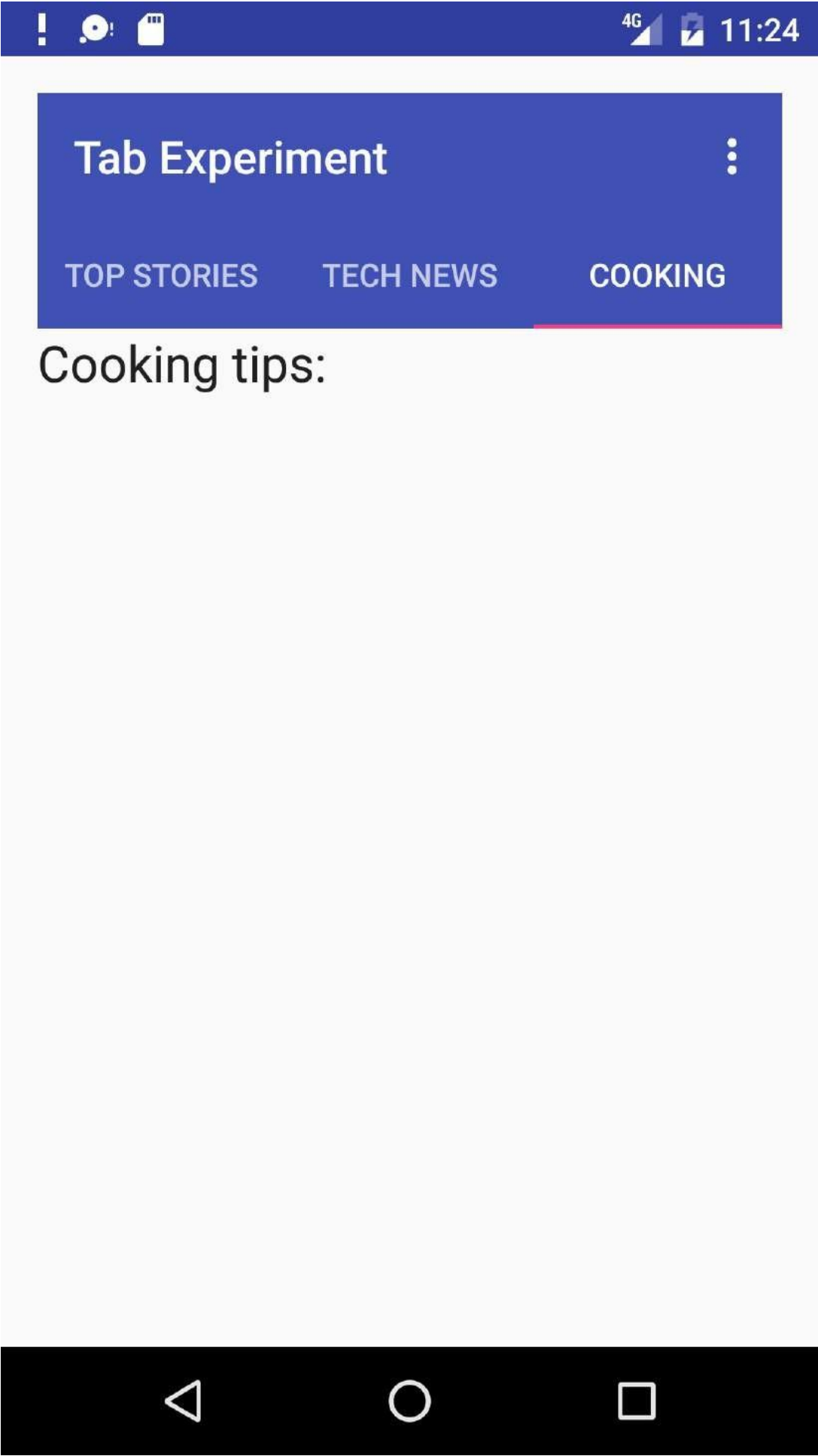
To use a `TabLayout`, you can design the main activity's layout to use a `Toolbar` for the app bar, a `TabLayout` for the tabs below the app bar, and a `ViewPager` within the root layout to switch child views. The layout should look similar to the following, assuming each child view fills the screen:

```
<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 file such as `tab_fragment1.xml`, `tab_fragment2.xml`, `tab_fragment3.xml`, and so on.



## Implementing each fragment

A *fragment* is a behavior or a portion of user interface within an activity. It's like a mini-activity within the main activity, with its own own lifecycle. To learn about fragments, see [Fragments in the API Guide](#).

Add a class for each fragment (such as `TabFragment1.java`, `TabFragment2.java`, and `TabFragment3.java`) representing a screen the user can visit by clicking a tab. Each class should extend `Fragment` and inflate the layout associated with the screen ( `tab_fragment1` , `tab_fragment2` , and `tab_fragment3` ). For example, `TabFragment1.java` looks like this:

```
public class TabFragment1 extends Fragment {

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

## Adding a pager adapter

Add a `PagerAdapter` that extends `FragmentStatePagerAdapter` and:

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

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

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

    @Override
    public Fragment getItem(int position) {

        switch (position) {
            case 0:
                return new TabFragment1();
            case 1:
                return new TabFragment2();
            case 2:
                return new TabFragment3();
            default:
                return null;
        }
    }

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

## 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) {
    ...
    // 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.
    ...
}
```

Extract string resources for the tab text set by `setText()` :

- "Top Stories" to `tab_label1`
- "Tech News" to `tab_label2`
- "Cooking" to `tab_label3`

## Managing screen views in fragments and set a listener

Use `PagerAdapter` in the main activity's `onCreate()` method to manage screen ("page") views in the fragments. 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) {
    ...
    // Use PagerAdapter to manage page views in fragments.
    // Each page is represented by its own fragment.
    // This is another example of the adapter pattern.
    final ViewPager viewPager = (ViewPager) findViewById(R.id.pager);
    final PagerAdapter adapter = new PagerAdapter
        (getSupportFragmentManager(), tabLayout.getTabCount());
    viewPager.setAdapter(adapter);
    // Setting a listener for clicks.
    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)

The `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.