Mobile programming

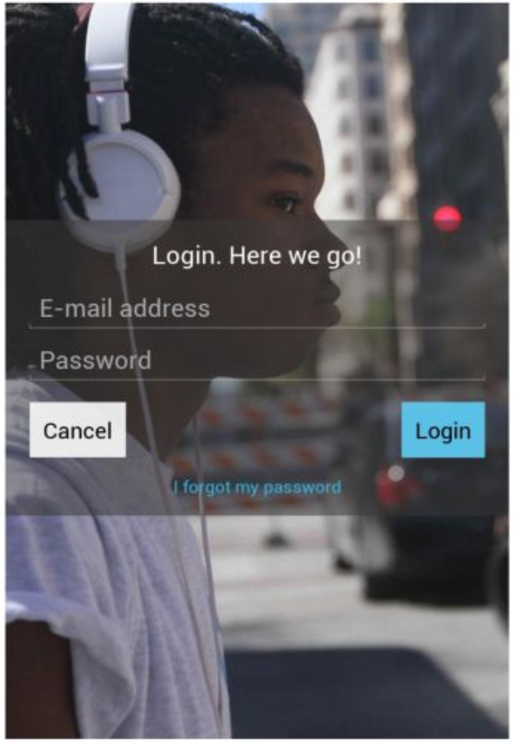
Omid Jafarinezhad

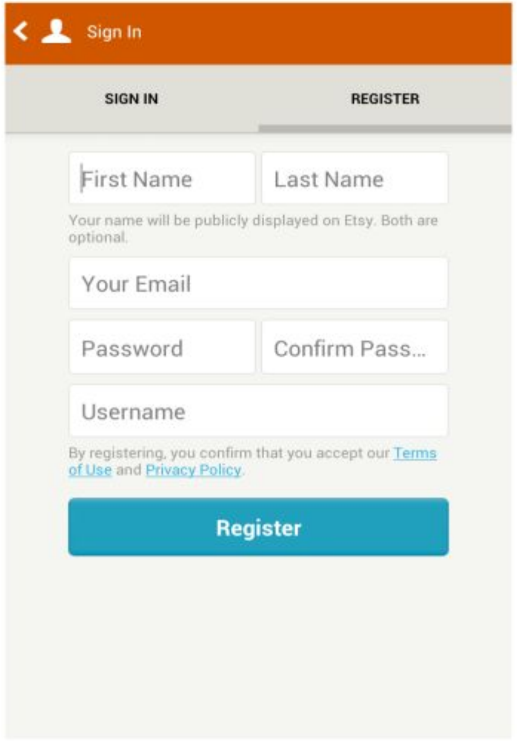
Android 02

BEFORE SMARTPHONES
HOW OLD IS BARBRA STREISAND?
I THINK SHE'S 71
REALLY? WOW!
WHO'S BARBRA STREISAND?

AFTER SMARTPHONES

Common app views – Login, Register

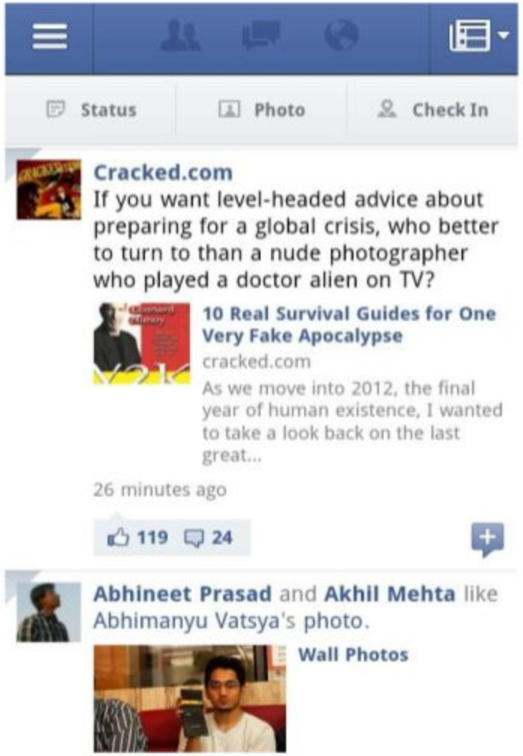




Your name will be publicly displayed on Etsy. Both are optional.

By registering, you confirm that you accept our Terms of Use and Privacy Policy.

Common app views – Stream, Detail



Status Photo Check In

Cracked.com
If you want level-headed advice about preparing for a global crisis, who better to turn to than a nude photographer who played a doctor alien on TV?

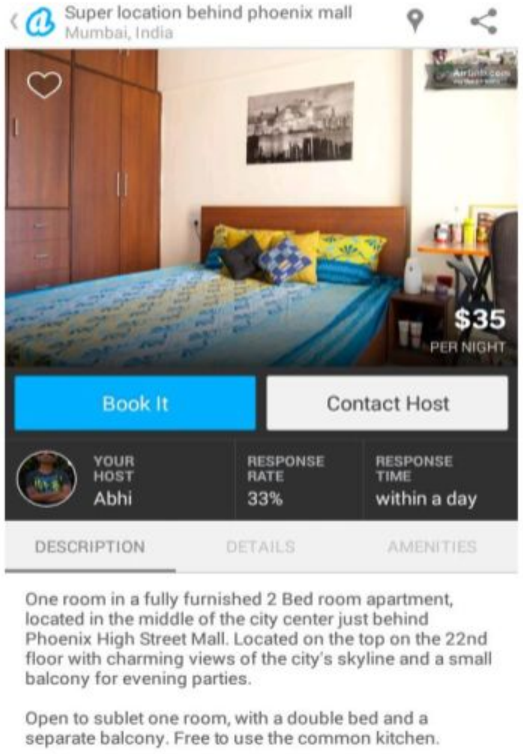
10 Real Survival Guides for One Very Fake Apocalypse
cracked.com
As we move into 2012, the final year of human existence, I wanted to take a look back on the last great...

26 minutes ago

119 24

Abhineet Prasad and **Akhil Mehta** like Abhimanyu Vatsya's photo.

Wall Photos



Super location behind phoenix mall
Mumbai, India

$35
PER NIGHT

Book It Contact Host

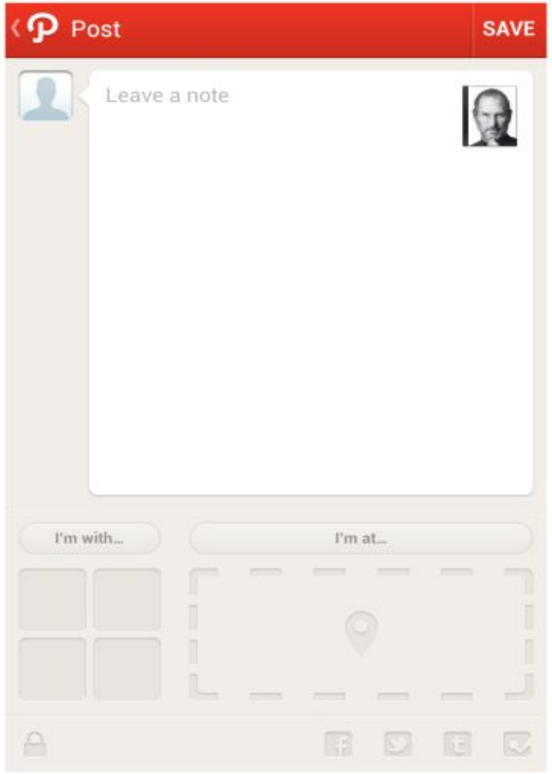
YOUR HOST	RESPONSE RATE	RESPONSE TIME
Abhi	33%	within a day

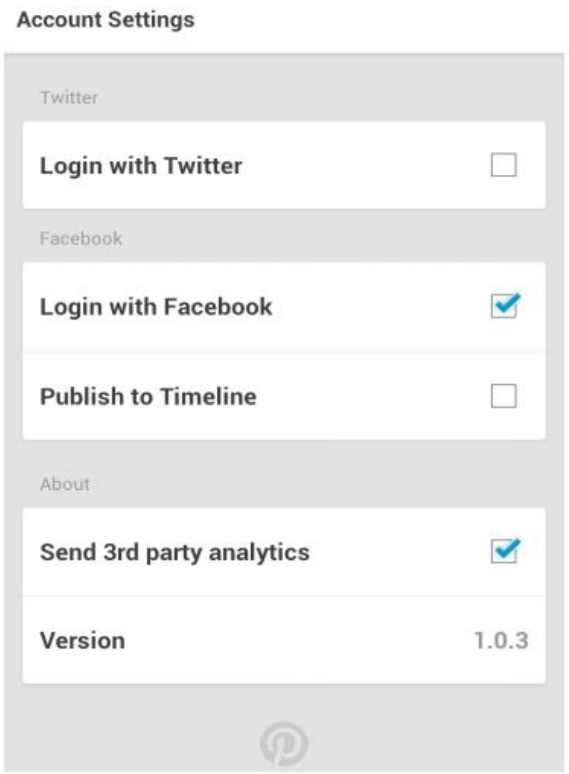
DESCRIPTION DETAILS AMENITIES

One room in a fully furnished 2 Bed room apartment, located in the middle of the city center just behind Phoenix High Street Mall. Located on the top on the 22nd floor with charming views of the city's skyline and a small balcony for evening parties.

Open to sublet one room, with a double bed and a separate balcony. Free to use the common kitchen.

Common app views – Creation, Settings




Account Settings

Twitter

Login with Twitter ☐

Facebook

Login with Facebook ☑

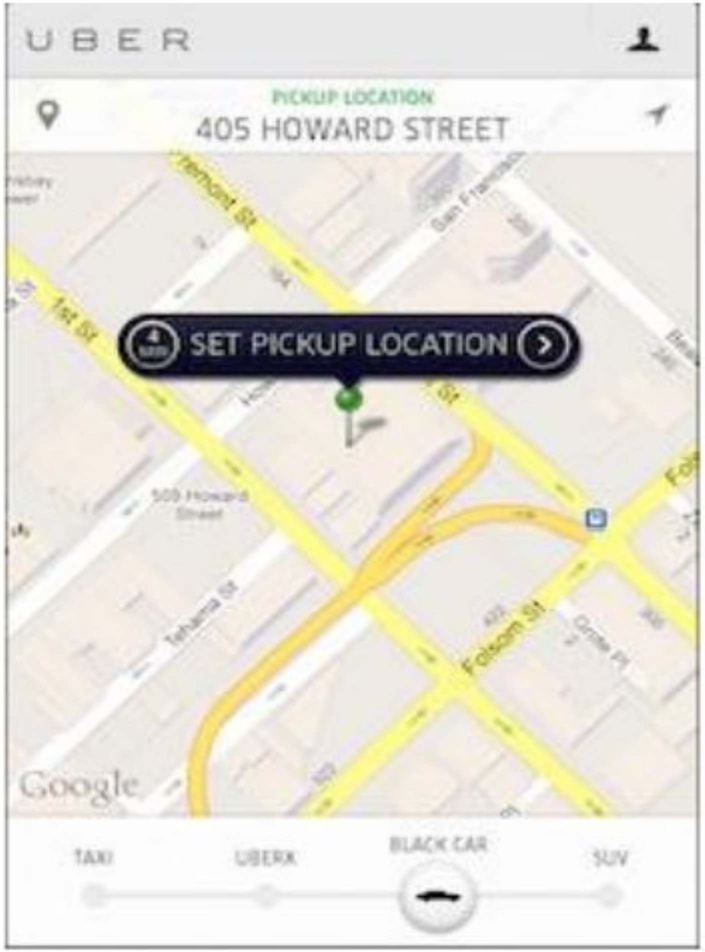
Publish to Timeline ☐

About

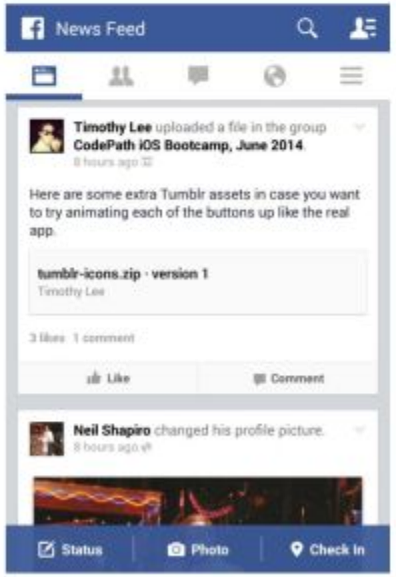
Send 3rd party analytics ☑

Version 1.0.3

Common app views – Maps



Mobile app architecture



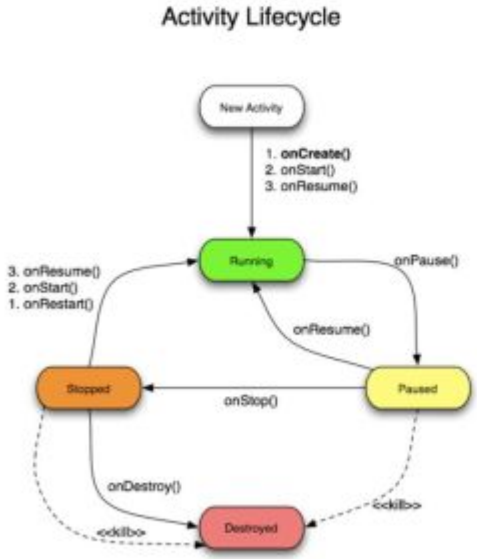
Timothy Lee uploaded a file in the group CodePath iOS Bootcamp, June 2014.
8 hours ago

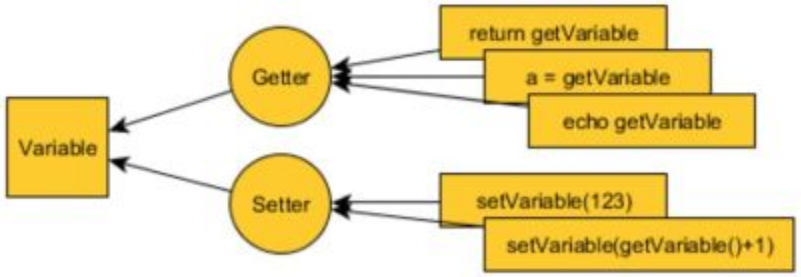
Here are some extra Tumblr assets in case you want to try animating each of the buttons up like the real app.

tumblr-icons.zip · version 1
Timothy Lee

3 likes 1 comment

Neil Shapiro changed his profile picture.
8 hours ago

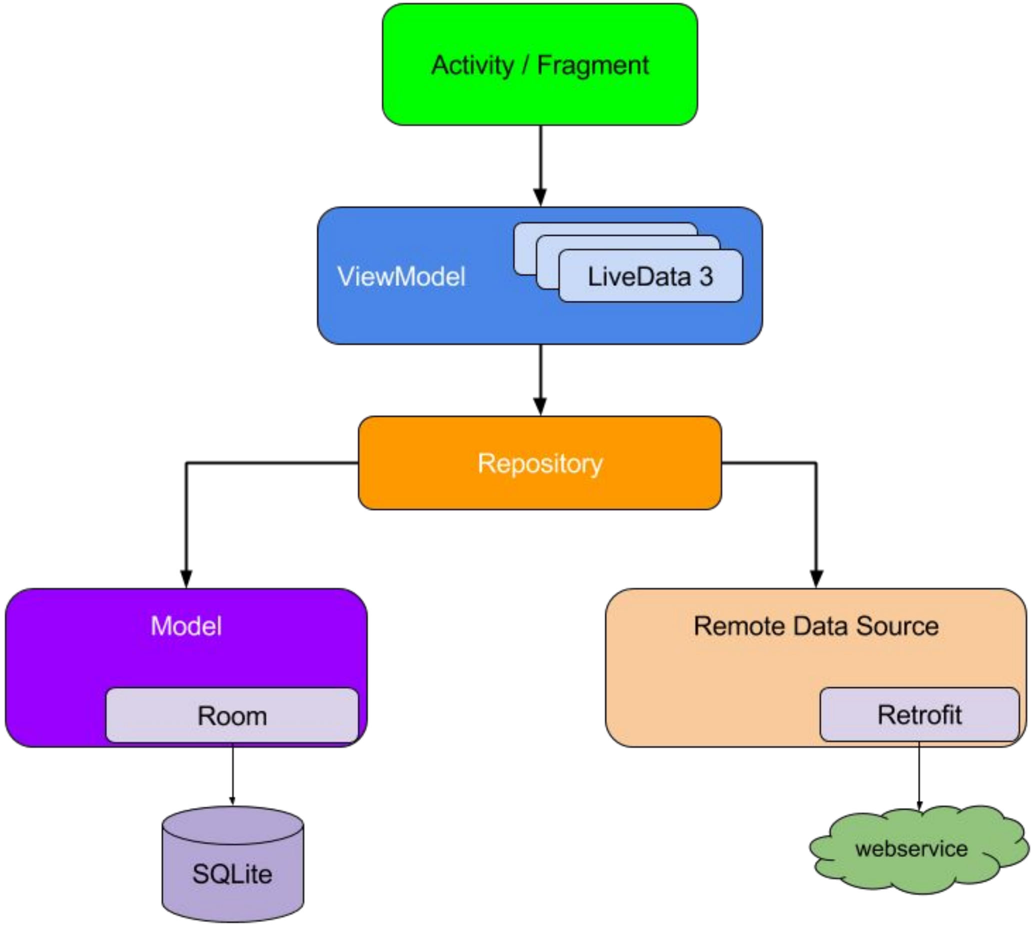




Views (objects)

Controllers (objects)

Models (objects)


Activity / Fragment
ViewModel
LiveData 3
Repository
Model
Room
SQLite
Remote Data Source
Retrofit
webservice

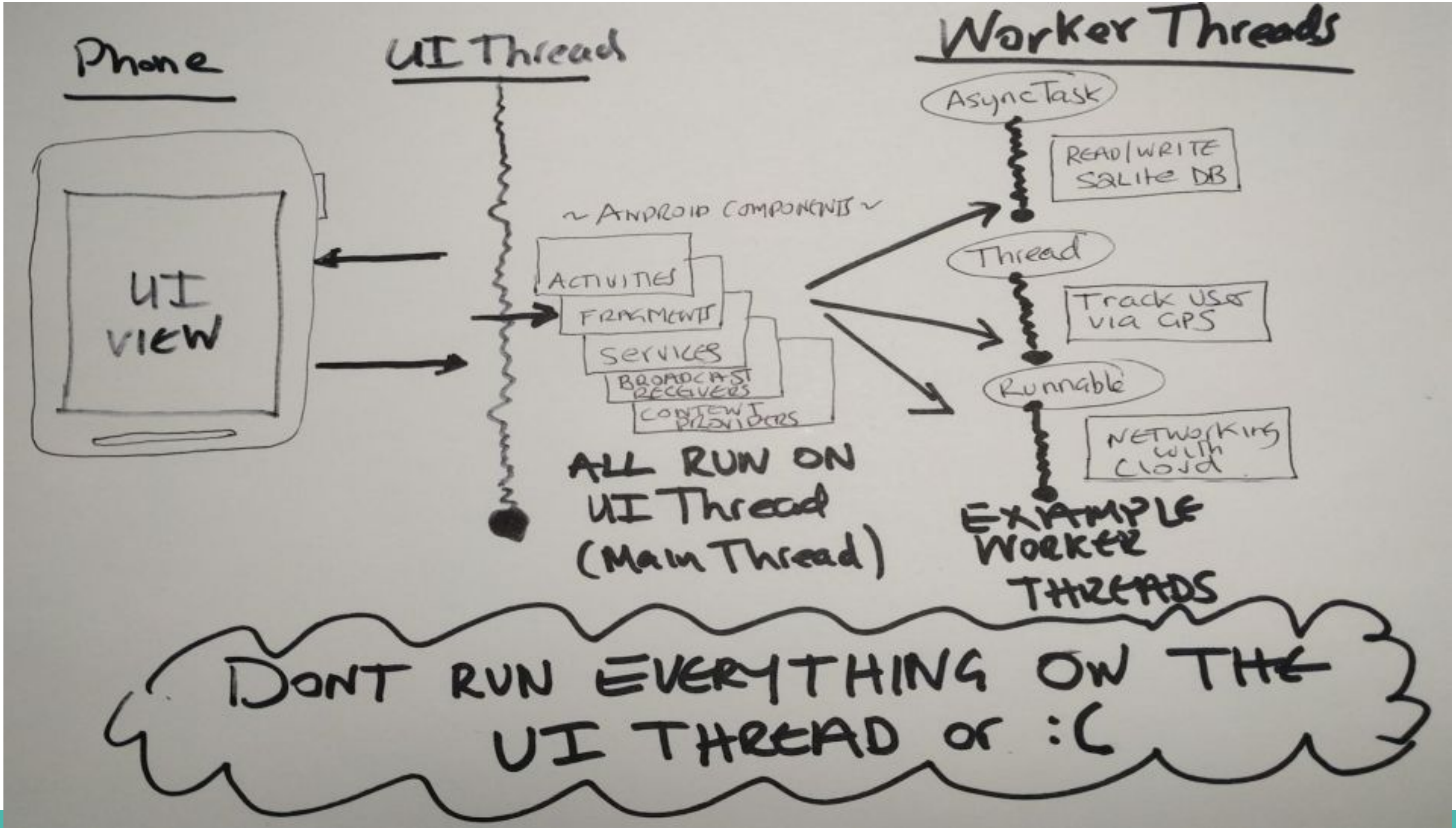
What's in an app: application components

- **Activities**: An activity is the key execution component . You need at least one activity in an app that deals with the UI but you probably will have multiple activity.
- **Fragments**: Fragments are new to Android but very important in programming the UI. I think of them as mini-activities.
- **Services**: Typically services are long running programs that don't need to interact with the UI . Examples of services could be listening to music, updating the location. Typically an activity will control a service -- that is, start it, pause it, bind and get data from it.

What's in an app: application components

- **Content providers**: Apps share data. The nice thing about Android is you can not only call internal apps such as the camera but you can get data from apps that you might need for your application.
- **Broadcast receivers**: If an service has data it can initiate broadcasts (something like *I got the location for anyone interested*). The other end of that are components (e.g., an activity) that are broadcast receivers.

App execution



Android Development Tools

The **Android Software Development Kit (Android SDK)** contains the necessary tools to create, compile and package Android applications.

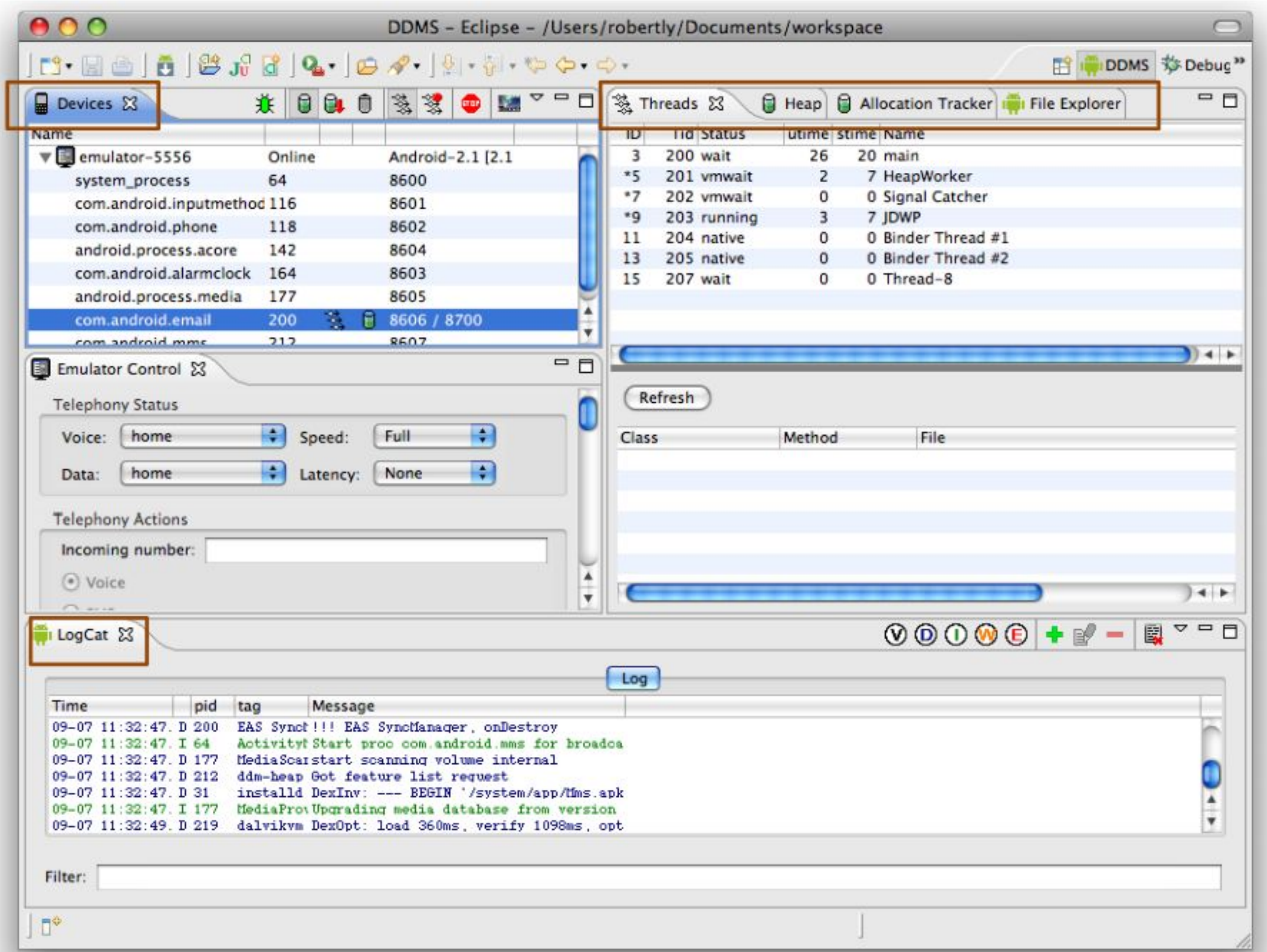
The Android SDK contains the **Android debug bridge (adb)**, which is a tool that allows you to connect to a virtual or real Android device, for the purpose of managing the device or debugging your application.

The Android SDK contains a tool called **dx** which converts Java class files into a .dex (Dalvik Executable) file.

•The Android tooling uses **Gradle as build system**.

Dalvik Debug Monitor Server (DDMS)

Android Studio includes a **debugging tool** called the **Dalvik Debug Monitor Server (DDMS)**, which provides *port-forwarding services, screen capture on the device, thread and heap information on the device, logcat, process, and radio state information, incoming call and SMS spoofing, location data spoofing, and more*.



Name		
emulator-5556	Online	Android-2.1 [2.1
system_process	64	8600
com.android.inputmethod	116	8601
com.android.phone	118	8602
android.process.acore	142	8604
com.android.alarmclock	164	8603
android.process.media	177	8605
com.android.email	200	8606 / 8700
com.android.mms	212	8607

ID	Tid	Status	utime	stime	Name
3	200	wait	26	20	main
*5	201	vmwait	2	7	HeapWorker
*7	202	vmwait	0	0	Signal Catcher
*9	203	running	3	7	JDWP
11	204	native	0	0	Binder Thread #1
13	205	native	0	0	Binder Thread #2
15	207	wait	0	0	Thread-8

Class	Method	File

Telephony Status

Voice: home Speed: Full

Data: home Latency: None

Telephony Actions

Incoming number:

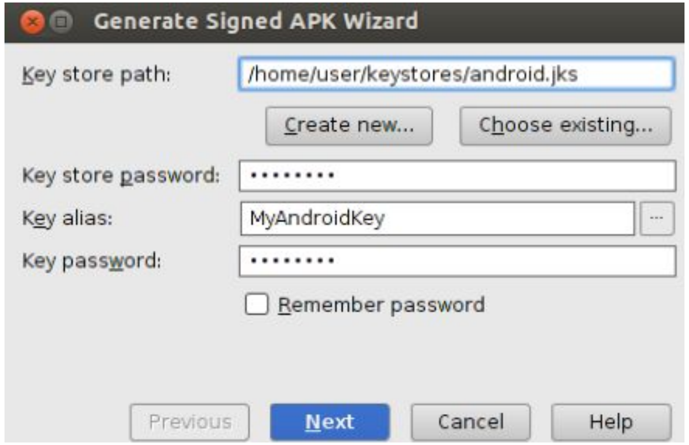
Voice

```
Time                pid  tag       Message
09-07 11:32:47. D 200  EAS Synct !!! EAS SyncManager, onDestroy
09-07 11:32:47. I 64   Activityt Start proc com.android.mms for broadca
09-07 11:32:47. D 177  MediaScar start scanning volume internal
09-07 11:32:47. D 212  ddm-heap Got feature list request
09-07 11:32:47. D 31   installd DexInv: --- BEGIN '/system/app/Mms.apk
09-07 11:32:47. I 177  MediaPro\ Upgrading media database from version
09-07 11:32:49. D 219  dalvikvm DexOpt: load 360ms, verify 1098ms, opt
```

Filter:

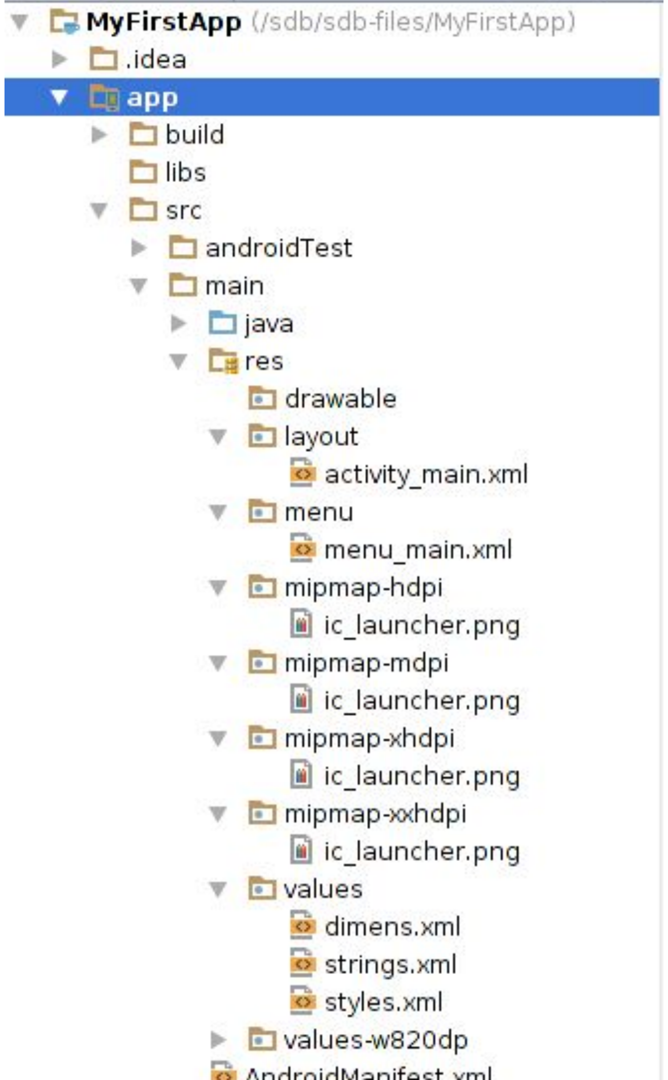
Android Development Tools

Android requires that all apps be **digitally signed** with a certificate before they can be installed. Android uses this certificate to identify the author of an app, and the certificate does not need to be signed by a certificate authority. Android apps often use self-signed certificates. *The app developer holds the certificate's private key*.


Generate Signed APK Wizard

Key store path:	/home/user/keystores/android.jks
	Create new... / Choose existing...
Key store password:	•••••••••
Key alias:	MyAndroidKey
Key password:	•••••••••
	☐ Remember password

Code structure



```
▼ MyFirstApp (/sdb/sdb-files/MyFirstApp)
  ▶ .idea
  ▼ app
    ▶ build
      libs
    ▼ src
      ▶ androidTest
      ▼ main
        ▶ java
        ▼ res
            drawable
          ▼ layout
              activity_main.xml
          ▼ menu
              menu_main.xml
          ▼ mipmap-hdpi
              ic_launcher.png
          ▼ mipmap-mdpi
              ic_launcher.png
          ▼ mipmap-xhdpi
              ic_launcher.png
          ▼ mipmap-xxhdpi
              ic_launcher.png
          ▼ values
              dimens.xml
              strings.xml
              styles.xml
          ▶ values-w820dp
          AndroidManifest.xml
```

java/
: Contains Java code sources.

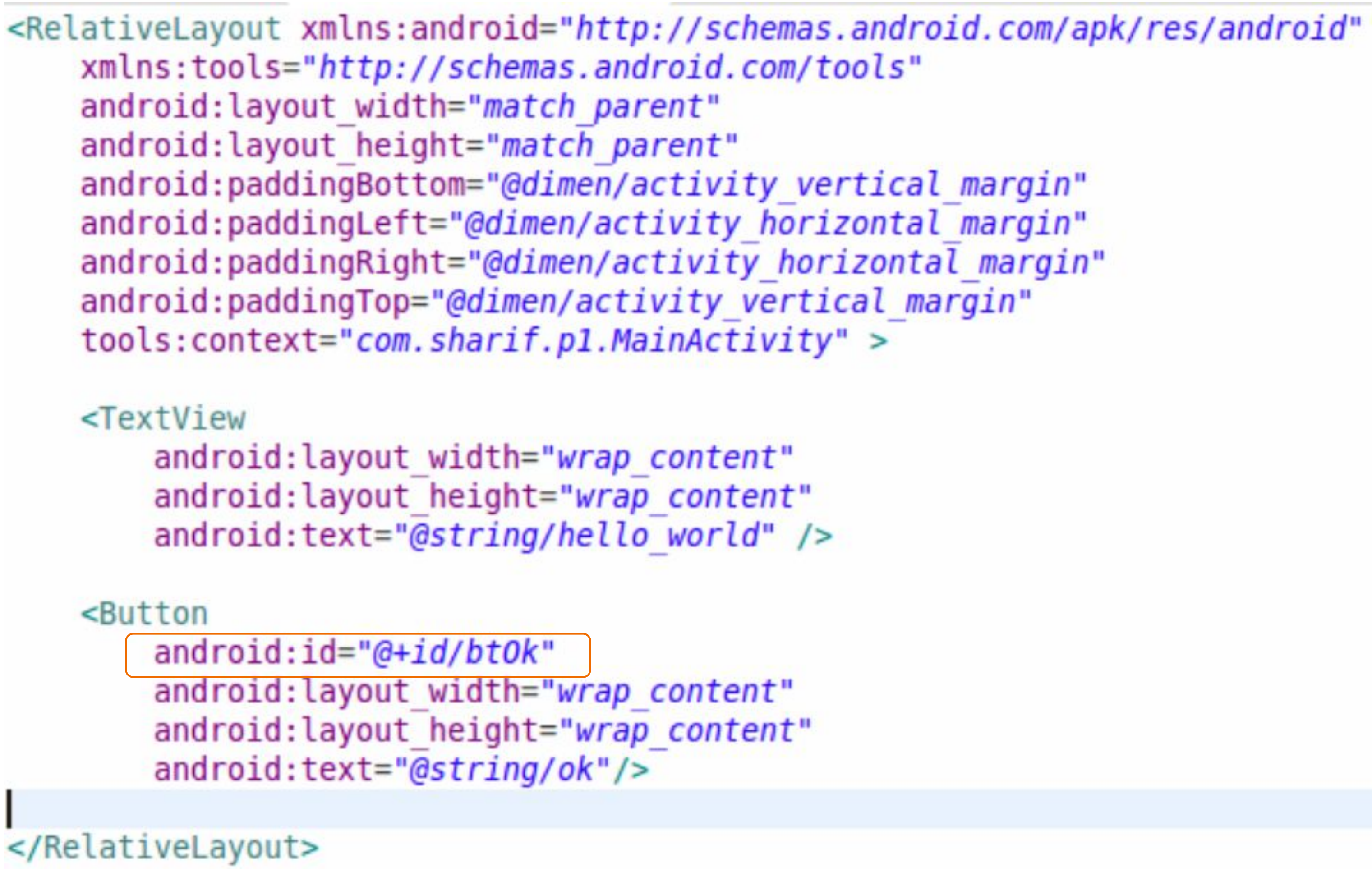
jni/
: Contains native code using the Java Native Interface (JNI). For more information, see the Android NDK documentation.

gen/
: Contains the Java files generated by Android Studio, such as your R.java file and interfaces created from AIDL files.

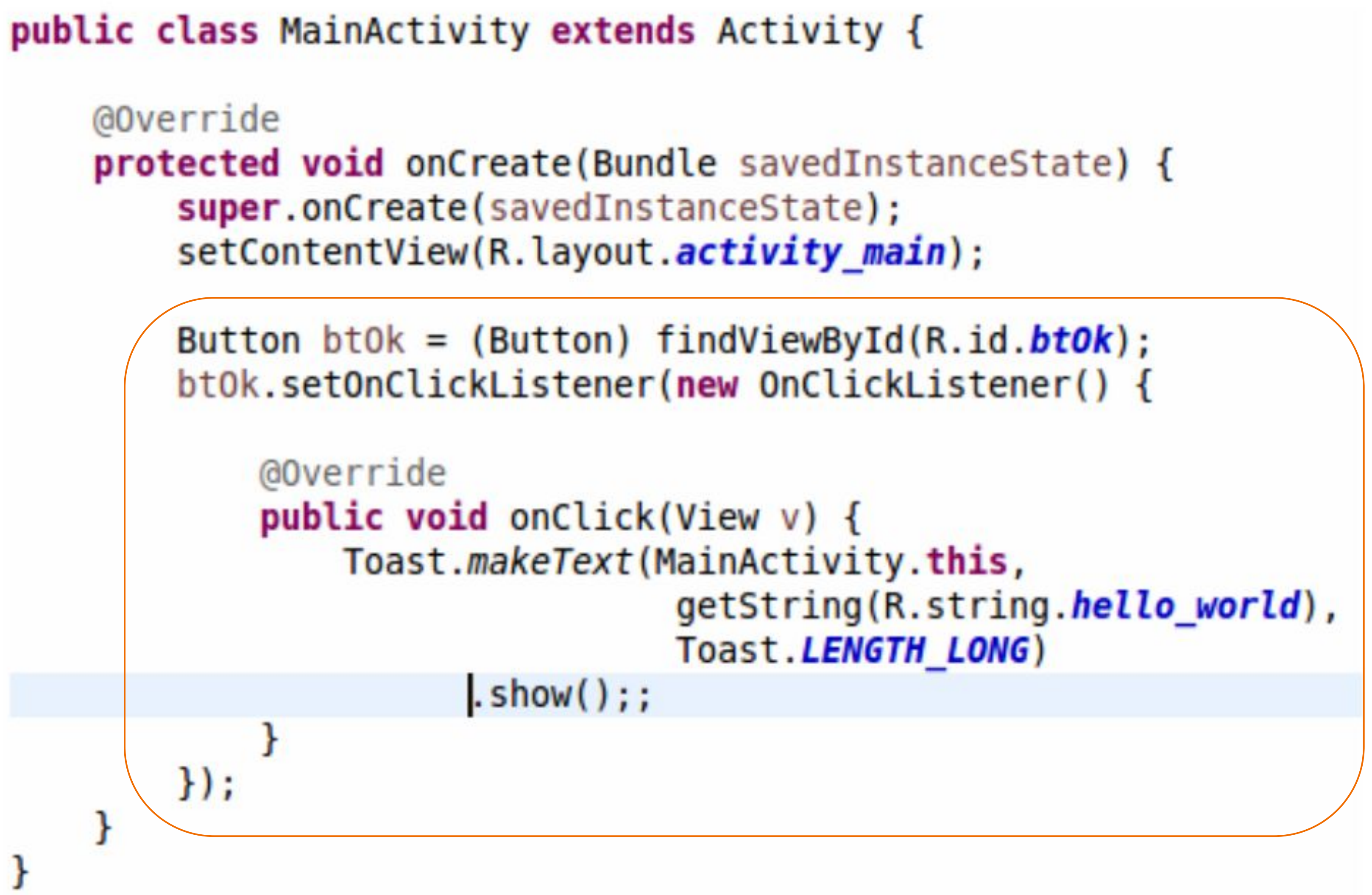
res/
: Contains application resources, such as drawable files, layout files, and UI string. See Application Resources for more information.

assets/
: Contains file that should be compiled into an .apk file as-is. You can navigate this directory in the same way as a typical file system using URIs and read files as a stream of bytes using the AssetManager . For example, this is a good location for textures and game data.

Example



```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.sharif.p1.MainActivity" >

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

    <Button
        android:id="@+id/btOk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/ok"/>

</RelativeLayout>
```



```
public class MainActivity extends Activity {

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

        Button btOk = (Button) findViewById(R.id.btOk);
        btOk.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                Toast.makeText(MainActivity.this,
                                getString(R.string.hello_world),
                                Toast.LENGTH_LONG)
                        .show();;
            }
        });
    }
}
```

User Interface

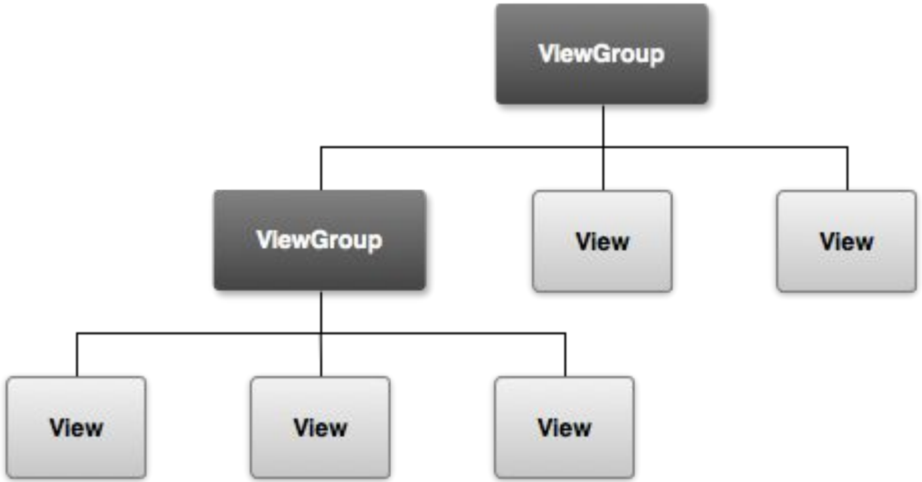
All user interface elements in an Android app are built using **View** and **ViewGroup** objects.

A **View** is an object that **draws something on the screen** that the user can interact with.

A **ViewGroup** is an object that **holds other View** (and ViewGroup) objects in order to define the layout of the interface.

User Interface

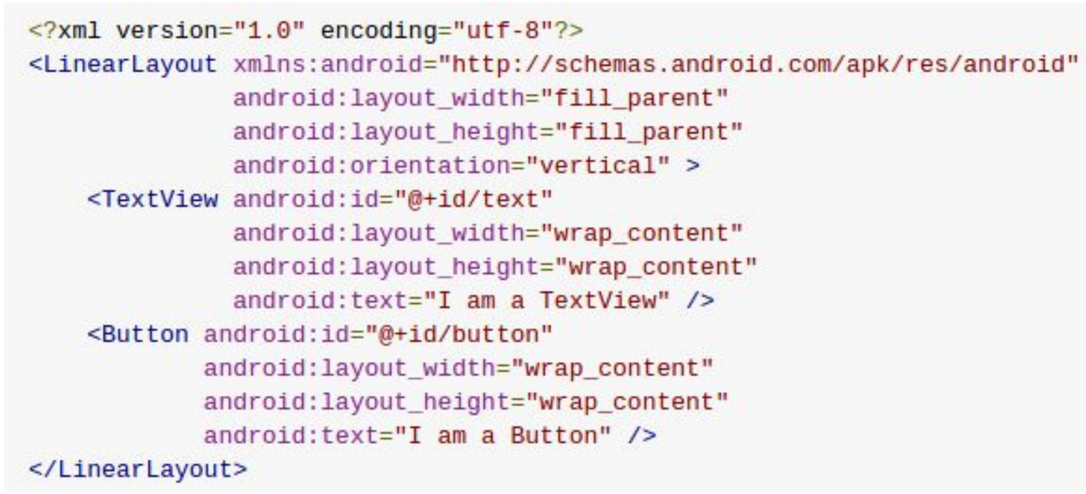
Illustration of a view hierarchy, which defines a UI layout



User Interface Layout

To **declare your layout,** you can **instantiate View objects in code** and start building a tree, **but the easiest and most effective way to define your layout is with an XML file**. XML offers a human-readable structure for the layout, similar to HTML.

The **name of an XML element for a view is respective to the Android class it represents**.
So a <TextView> element creates a TextView widget in your UI, and a <LinearLayout> element creates a LinearLayout viewgroup



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="I am a Button" />
</LinearLayout>
```

Layouts

A layout **defines the visual structure** for a user interface, such as the UI for an activity or app widget. You can declare a layout in two ways:

Declare UI elements in XML. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.

Instantiate layout elements at runtime. Your application can create View and ViewGroup objects (and manipulate their properties) **programmatically.**

Load the XML Resource

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="I am a Button" />
</LinearLayout>
```

created and added to our resources (in the **R.java** file)

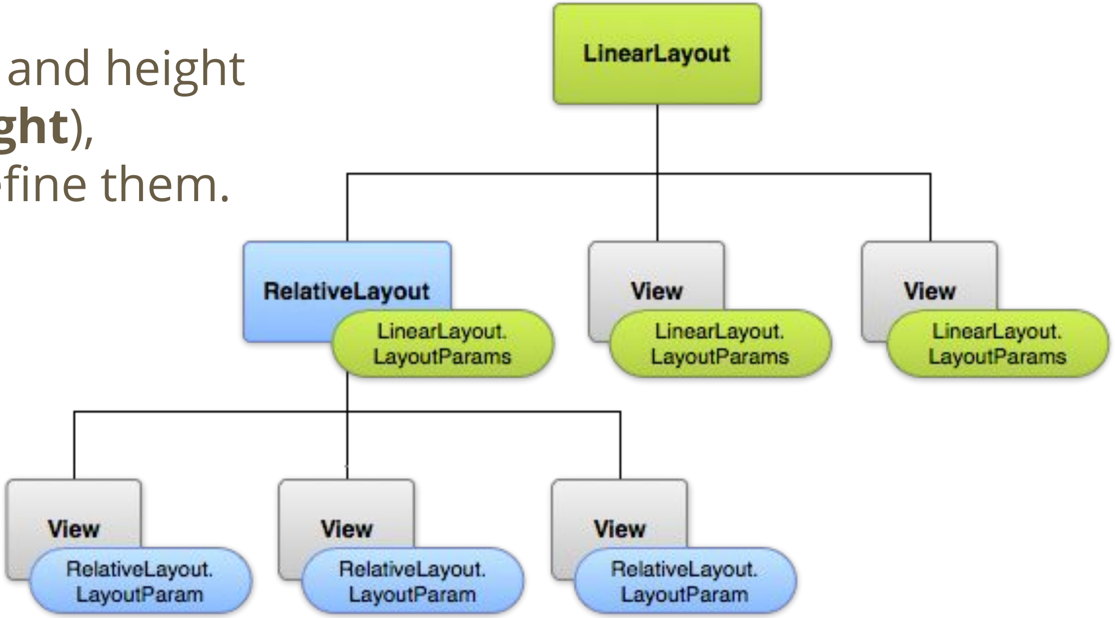
```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

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

Layout Parameters

XML layout attributes named **layout_something** define layout parameters for the View that are appropriate for the ViewGroup in which it resides.

All view groups include a width and height (**layout_width** and **layout_height**), and each view is required to define them.


LinearLayout
RelativeLayout
LinearLayout.LayoutParams
View
LinearLayout.LayoutParams
View
LinearLayout.LayoutParams
View
RelativeLayout.LayoutParam
View
RelativeLayout.LayoutParam
View
RelativeLayout.LayoutParam

layout_width and layout_height

You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will use one of these constants to set the width or height:

wrap_content tells your view **to size itself to the dimensions required by its content**.

match_parent (named **fill_parent before API Level 8**) tells your view to **become as big as its parent view group will allow**.

Layout Position

The **geometry of a view** is that of a **rectangle.** A view has a location, expressed as a pair of **left and top coordinates**, and two dimensions, expressed as a width and a height. The **unit for location and dimensions** is the **pixel**.

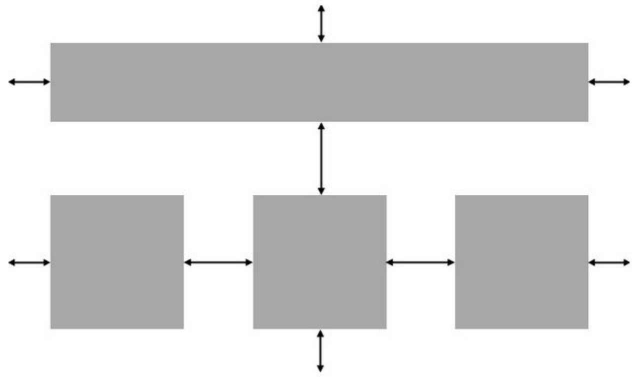
It is possible to retrieve the location of a view by invoking the methods **getLeft()** and **getTop()** (also **getRight()** and **getBottom()**)

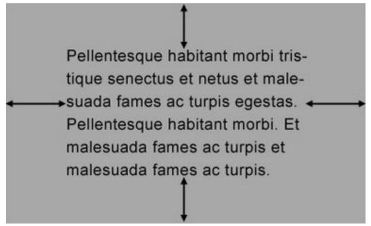
These methods both **return the location of the view relative to its parent**. For instance, when getLeft() returns 20, that means the view is located 20 pixels to the right of the left edge of its direct parent.

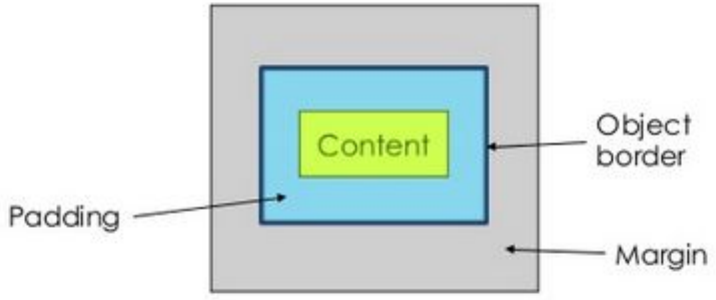
Size, Padding and Margins

The **size** of a view is expressed with a **width** and a **height**

Padding can be used to **offset the content of the view** by a specific number of pixels

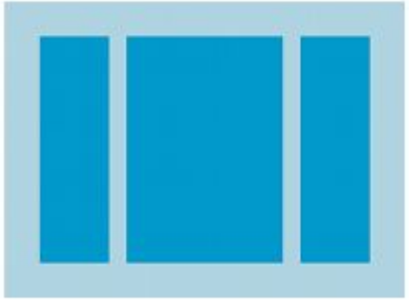



Pellentesque habitant morbi tris-
tique senectus et netus et male-
suada fames ac turpis egestas.
Pellentesque habitant morbi. Et
malesuada fames ac turpis et
malesuada fames ac turpis.



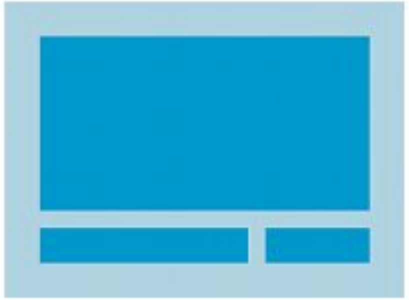
Common Layouts

Linear Layout



A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.

Relative Layout



Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

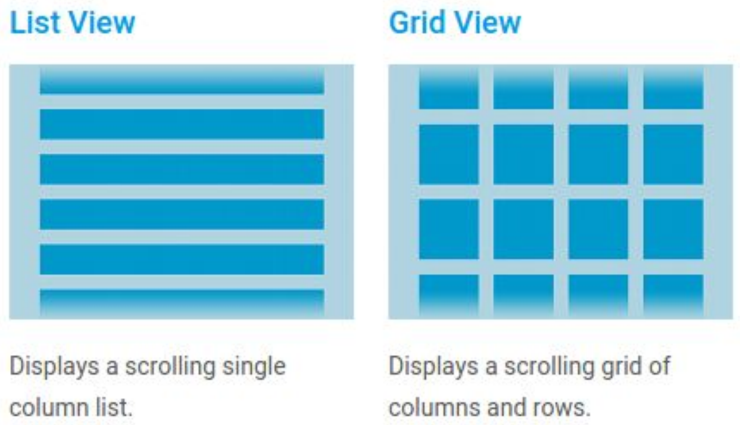
Web View

Displays web pages.

Building Layouts with an Adapter

When the **content for your layout is dynamic or not pre-determined**, you can use a layout that subclasses **AdapterView** to populate the layout with views at runtime. A **subclass** of the **AdapterView** class uses an **Adapter** to bind data to its layout.

The Adapter behaves as a middleman between the data source and the AdapterView layout—the **Adapter retrieves the data** (from a source such as an array or a database query) and **converts each entry into a view** that can be added into the AdapterView layout.


List View
Grid View
Displays a scrolling single column list.
Displays a scrolling grid of columns and rows.

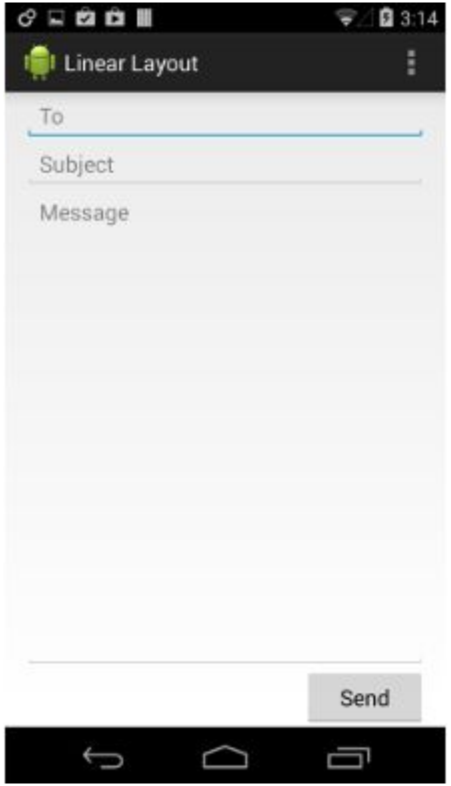
Linear Layout

LinearLayout is a view group that **aligns all children in a single direction**, **vertically or horizontally**. You can specify the layout direction with the **android:orientation** attribute

LinearLayout also supports **assigning a weight to individual children with the android:layout_weight attribute**. This attribute assigns an "importance" value to a view **in terms of how much space it should occupy on the screen**

A **larger weight value** allows it **to expand to fill any remaining space** in the parent view. **Default weight is zero**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.an
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:orientation="vertical" >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/to" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/subject" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/message" />
    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="@string/send" />
</LinearLayout>
```



Layout Weight example

For example, if there are three text fields and two of them declare a weight of 1, while the other is given no weight, the third text field without weight will not grow and will only occupy the area required by its content. The other two will expand equally to fill the space remaining after all three fields are measured. If the third field is then given a weight of 2 (instead of 0), then it is now declared more important than both the others, so it gets half the total remaining space, while the first two share the rest equally.

Relative Layout

RelativeLayout is a view group that **displays child views in relative positions**. The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent RelativeLayout area (such as aligned to the bottom, left or center).

RelativeLayout **lets child views specify their position relative to the parent view or to each other (specified by ID)**.

Positioning Views

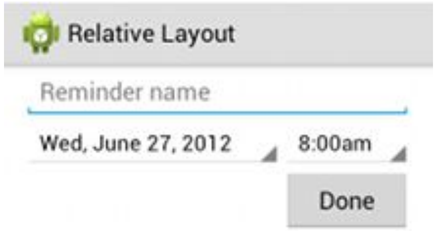
android:**layout_alignParentTop**: If "true", makes the top edge of this view match the top edge of the parent

android:**layout_centerVertical:** If "true", centers this child vertically within its parent

android:**layout_below:** Positions the top edge of this view below the view specified with a resource ID

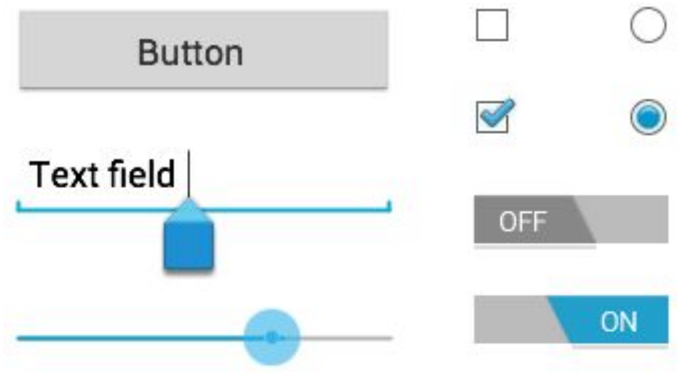
android:**layout_toRightOf:** Positions the left edge of this view to the right of the view specified with a resource ID.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.andr
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
    <EditText
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
    <Spinner
        android:id="@+id/dates"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentLeft="true"
        android:layout_toLeftOf="@+id/times" />
    <Spinner
        android:id="@id/times"
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentRight="true" />
    <Button
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/times"
        android:layout_alignParentRight="true"
        android:text="@string/done" />
</RelativeLayout>
```



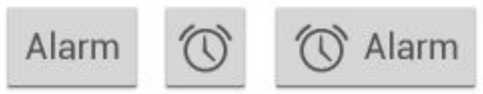
Input Controls

Input controls are the **interactive components** in your app's user interface. Android provides a wide variety of controls you can use in your UI, such as buttons, text fields, seek bars, checkboxes, zoom buttons, toggle buttons, and many more.



Buttons

A button consists of **text or an icon** (or both text and an icon) that communicates what action occurs when the **user touches it**



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

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

Responding to Click Events

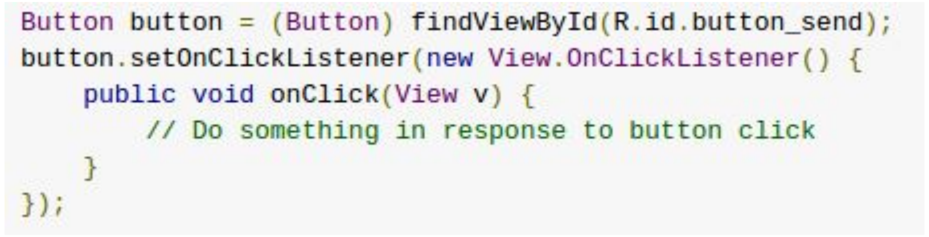
```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/button_send"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage" />
```

Within the Activity that hosts this layout, the following method handles the click event:

```
/** Called when the user touches the button */
public void sendMessage(View view) {
    // Do something in response to button click
}
```

Using an OnClickListener

To declare the event handler **programmatically**, create an **View.OnClickListener** object and assign it to the button by calling **setOnClickListener(View.OnClickListener)**. For example:



```
Button button = (Button) findViewById(R.id.button_send);
button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Do something in response to button click
    }
});
```

Styling Your Button

To create a borderless button, apply the borderlessButtonStyle style to the button. 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"
    style="?android:attr/borderlessButtonStyle" />
```

Custom background

You can define the **state list in an XML file** that defines three different images or colors to use for the different button states.

Create three bitmaps for the button background that represent the **default, pressed, and focused button states**.

Place the bitmaps into the **res/drawable/** directory of your project. Be sure **each bitmap is named properly to reflect the button state** that they each represent, such as button_default.9.png, button_pressed.9.png, and button_focused.9.png

Custom background

Create a new XML file in the **res/drawable/** directory (name it something like **button_custom.xml**). Insert the following XML:

```xml
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/button_pressed"   android:state_pressed="true" />
    <item android:drawable="@drawable/button_focused"   android:state_focused="true" />
    <item android:drawable="@drawable/button_default" />
</selector>
```

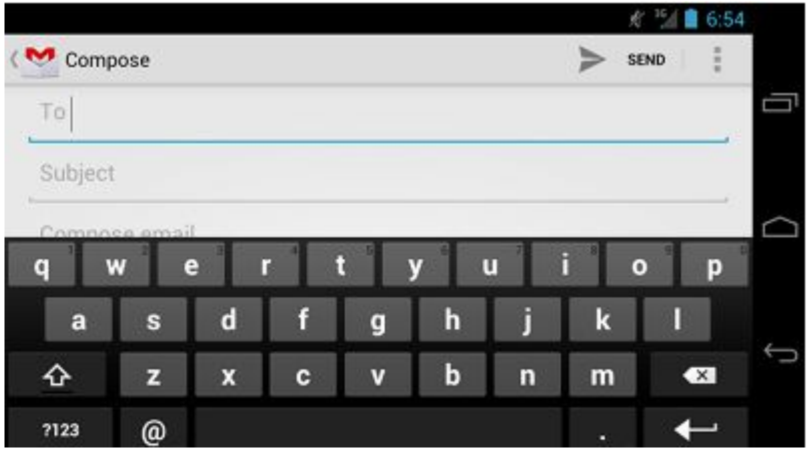
Custom background

The order of the <item> elements is important. When this drawable is referenced, the <item> elements are traversed in-order to determine which one is appropriate for the current button state. Because the default bitmap is last, it is only applied when the conditions android:state_pressed and android:state_focused have both evaluated as false.

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

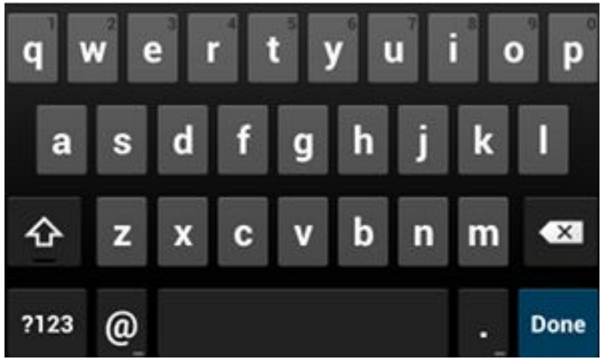
Text Fields

A text field allows the **user to type text into your app**. It can be either single line or multi-line. Touching a text field places the cursor and automatically displays the keyboard

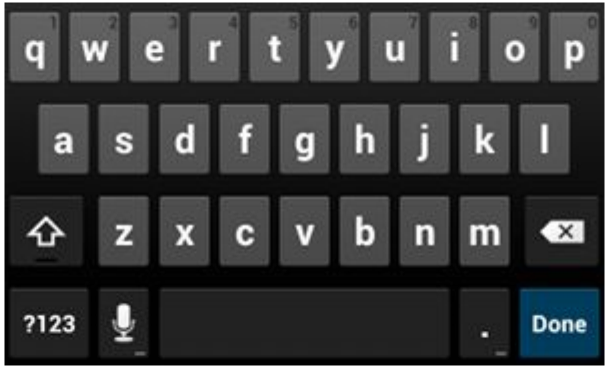


Specifying the Keyboard Type

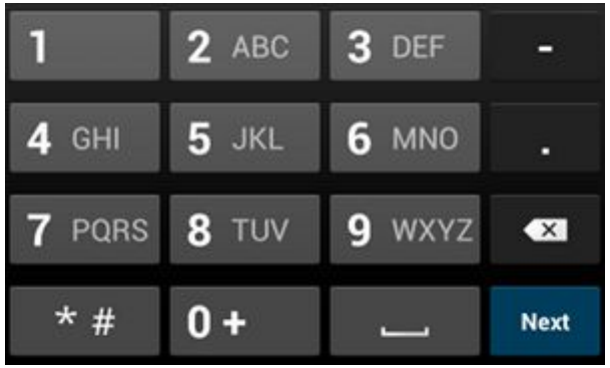
```
<EditText
    android:id="@+id/email_address"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/email_hint"
    android:inputType="textEmailAddress" />
```



The textEmailAddress input type



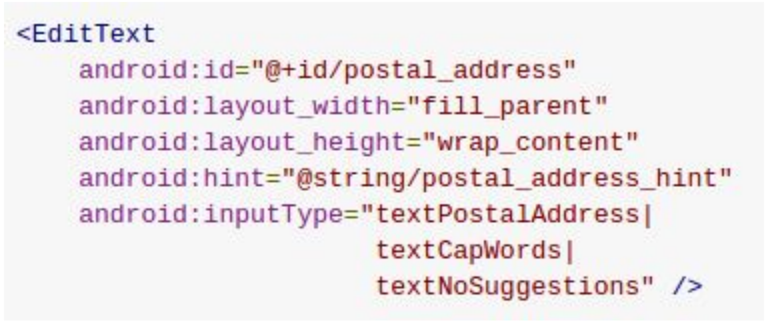
The default text input type.



The phone input type.

Controlling other behaviors

The **android:inputType** also allows you **to specify certain keyboard behaviors**, such as **whether to capitalize all new words or use features like auto-complete and spelling suggestions**.



```
<EditText
    android:id="@+id/postal_address"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/postal_address_hint"
    android:inputType="textPostalAddress|
                       textCapWords|
                       textNoSuggestions" />
```

Controlling other behaviors

"**textCapSentences**": Normal text keyboard that capitalizes the first letter for each new sentence.

"**textCapWords"**: Normal text keyboard that capitalizes every word. Good for titles or person names.

"**textAutoCorrect"**: Normal text keyboard that corrects commonly misspelled words.

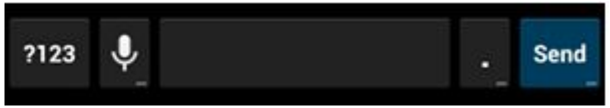
"textPassword": Normal text keyboard, but the characters entered turn into dots.

"**textMultiLine"**: Normal text keyboard that allow users to input long strings of text that include line breaks (carriage returns).

Specifying Keyboard Actions

In addition to changing the keyboard's input type, Android allows you **to specify an action to be made when users have completed their input**. The action specifies the button that appears **in place of the carriage return** key and the action to be made, such as "Search" or "Send."

```
<EditText
    android:id="@+id/search"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/search_hint"
    android:inputType="text"
    android:imeOptions="actionSend" />
```



If you declare`android:imeOptions="actionSend"`, the keyboard includes the Send action

Responding to action button events

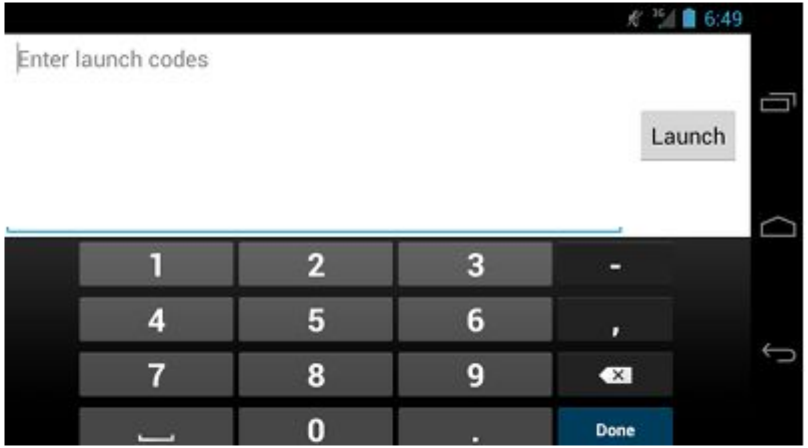
```
<EditText
    android:id="@+id/search"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/search_hint"
    android:inputType="text"
    android:imeOptions="actionSend" />
```

```
EditText editText = (EditText) findViewById(R.id.search);
editText.setOnEditorActionListener(new OnEditorActionListener() {
    @Override
    public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
        boolean handled = false;
        if (actionId == EditorInfo.IME_ACTION_SEND) {
            sendMessage();
            handled = true;
        }
        return handled;
    }
});
```

Setting a custom action button label

If the keyboard is too large to reasonably share space with the underlying application (such as when a handset device is in landscape orientation) then fullscreen ("extract mode") is triggered. In this mode, a labeled action button is displayed next to the input. You can customize the text of this button by setting the android:imeActionLabel attribute:

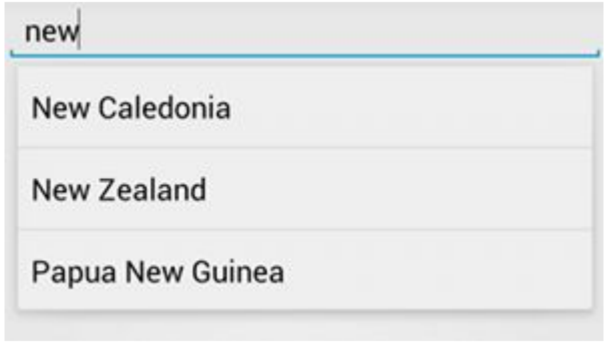
```
<EditText
    android:id="@+id/launch_codes"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/enter_launch_codes"
    android:inputType="number"
    android:imeActionLabel="@string/launch" />
```

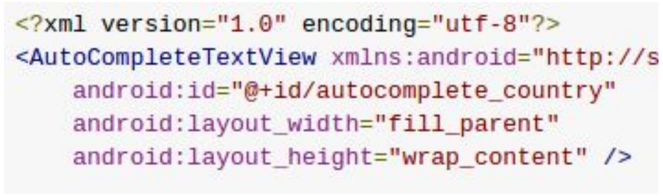


Providing Auto-complete Suggestions

If you want to **provide suggestions to users as they type**, you can use a **subclass of EditText called AutoCompleteTextView**.

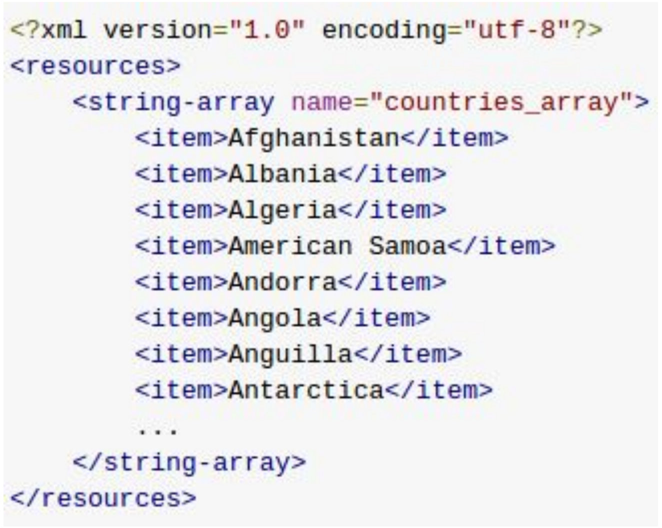
To implement auto-complete, **you must specify an Adapter** that provides the text suggestions.





```xml
<?xml version="1.0" encoding="utf-8"?>
<AutoCompleteTextView xmlns:android="http://s
    android:id="@+id/autocomplete_country"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

Define the array that contains all text suggestions. For example, here's an array of country names that's defined in an XML resource file (**res/values/strings.xml**):

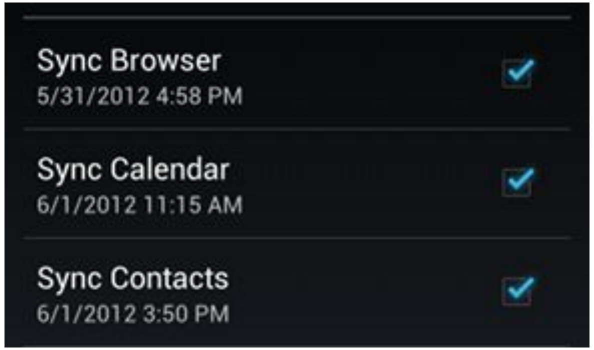


```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="countries_array">
        <item>Afghanistan</item>
        <item>Albania</item>
        <item>Algeria</item>
        <item>American Samoa</item>
        <item>Andorra</item>
        <item>Angola</item>
        <item>Anguilla</item>
        <item>Antarctica</item>
        ...
    </string-array>
</resources>
```

```java
// Get a reference to the AutoCompleteTextView in the layout
AutoCompleteTextView textView = (AutoCompleteTextView)
findViewById(R.id.autocomplete_country);
// Get the string array
String[] countries = getResources().getStringArray(R.array.countries_array);
// Create the adapter and set it to the AutoCompleteTextView
ArrayAdapter<String> adapter =
        new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, countries);
textView.setAdapter(adapter);
```

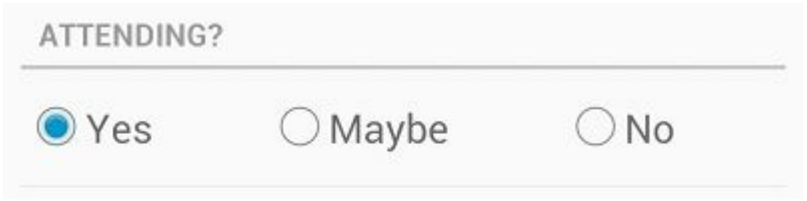
Checkboxes

Checkboxes allow the **user to select one or more options from a set**. Typically, you should present each checkbox option in a vertical list.


Sync Browser
5/31/2012 4:58 PM
Sync Calendar
6/1/2012 11:15 AM
Sync Contacts
6/1/2012 3:50 PM

Radio Buttons

Radio buttons **allow the user to select one option from a set**. You should use radio buttons for optional sets that are **mutually exclusive** if you think that the user needs to see all available options side-by-side. If it's not necessary to show all options side-by-side, use a spinner instead.



Responding to Click Events

The **RadioGroup** is a subclass of **LinearLayout** that has a vertical orientation by default.

```xml
<?xml version="1.0" encoding="utf-8"?>
<RadioGroup xmlns:android="........."
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <RadioButton android:id="@+id/radio_pirates"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/pirates"
        android:onClick="onRadioButtonClicked"/>
    <RadioButton android:id="@+id/radio_ninjas"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/ninjas"
        android:onClick="onRadioButtonClicked"/>
</RadioGroup>
```

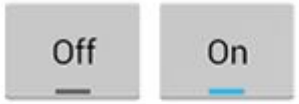
```java
public void onRadioButtonClicked(View view) {
    // Is the button now checked?
    boolean checked = ((RadioButton) view).isChecked();

    // Check which radio button was clicked
    switch(view.getId()) {
        case R.id.radio_pirates:
            if (checked)
                // Pirates are the best
            break;
        case R.id.radio_ninjas:
            if (checked)
                // Ninjas rule
            break;
    }
}
```

Toggle Buttons

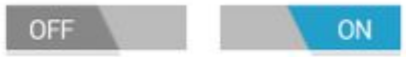
A toggle button allows the **user to change a setting between two states**.

Android 4.0 (API level 14) introduces another kind of toggle button called a **switch** that provides a slider control, which you can add with a Switch object.

If you need to change a button's state yourself, you can use the CompoundButton.**setChecked()** or CompoundButton.**toggle()** methods.



Toggle buttons



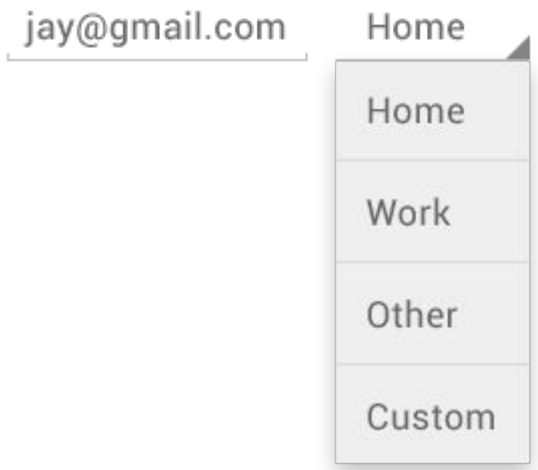
Switches (in Android 4.0+)

Responding to Button Presses

```
ToggleButton toggle = (ToggleButton) findViewById(R.id.togglebutton);
toggle.setOnCheckedChangeListener(new
CompoundButton.OnCheckedChangeListener() {
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked)
{
        if (isChecked) {
            // The toggle is enabled
        } else {
            // The toggle is disabled
        }
    }
});
```

Spinners

Spinners provide **a quick way to select one value from a set**. In the **default state,** a spinner **shows its currently selected value**. Touching the spinner displays a dropdown menu with all other available values, from which the user can select a new one.



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

Populate the Spinner with User Choices

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="planets_array">
        <item>Mercury</item>
        <item>Venus</item>
        <item>Earth</item>
        <item>Mars</item>
        <item>Jupiter</item>
        <item>Saturn</item>
        <item>Uranus</item>
        <item>Neptune</item>
    </string-array>
</resources>
```

```java
Spinner spinner = (Spinner) findViewById(R.id.spinner);
// Create an ArrayAdapter using the string array and a default spinner layout
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
        R.array.planets_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
spinner.setAdapter(adapter);
```

Responding to User Selections

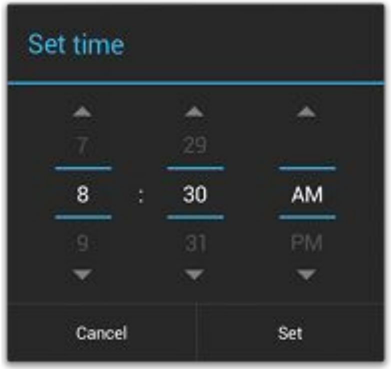
```
public class SpinnerActivity extends Activity implements OnItemSelectedListener
{
    ...

    public void onItemSelected(AdapterView<?> parent, View view,
            int pos, long id) {
        // An item was selected. You can retrieve the selected item using
        // parent.getItemAtPosition(pos)
    }

    public void onNothingSelected(AdapterView<?> parent) {
        // Another interface callback
    }
}
```

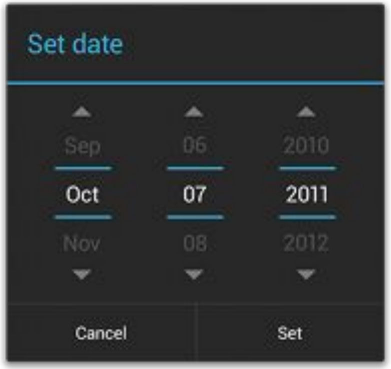
```
Spinner spinner = (Spinner) findViewById(R.id.spinner);
spinner.setOnItemSelectedListener(this);
```

Pickers

Android provides controls for **the user to pick a time or pick a date as ready-to-use dialogs**. Each picker provides controls for selecting each part of the time (hour, minute, AM/PM) or date (month, day, year)

We **recommend** that you use **DialogFragment** to host each time or date picker. The DialogFragment manages the dialog lifecycle for you and allows you to display the pickers in different layout configurations, such as in a basic dialog on handsets or as an embedded part of the layout on large screens.





DialogFragment

Although **DialogFragment** was first added to the platform in **Android 3.0 (API level 11)**, if your app supports versions of Android older than 3.0—even as low as Android 1.6—you can use the DialogFragment class that's available in the **support library for backward compatibility**.

Creating a Time Picker

To define a DialogFragment for a TimePickerDialog, you must:

- Define the **onCreateDialog()** method to **return an instance of TimePickerDialog**

- **Implement** the **TimePickerDialog.OnTimeSetListener** interface to receive a callback when the user sets the time.

Creating a Time Picker

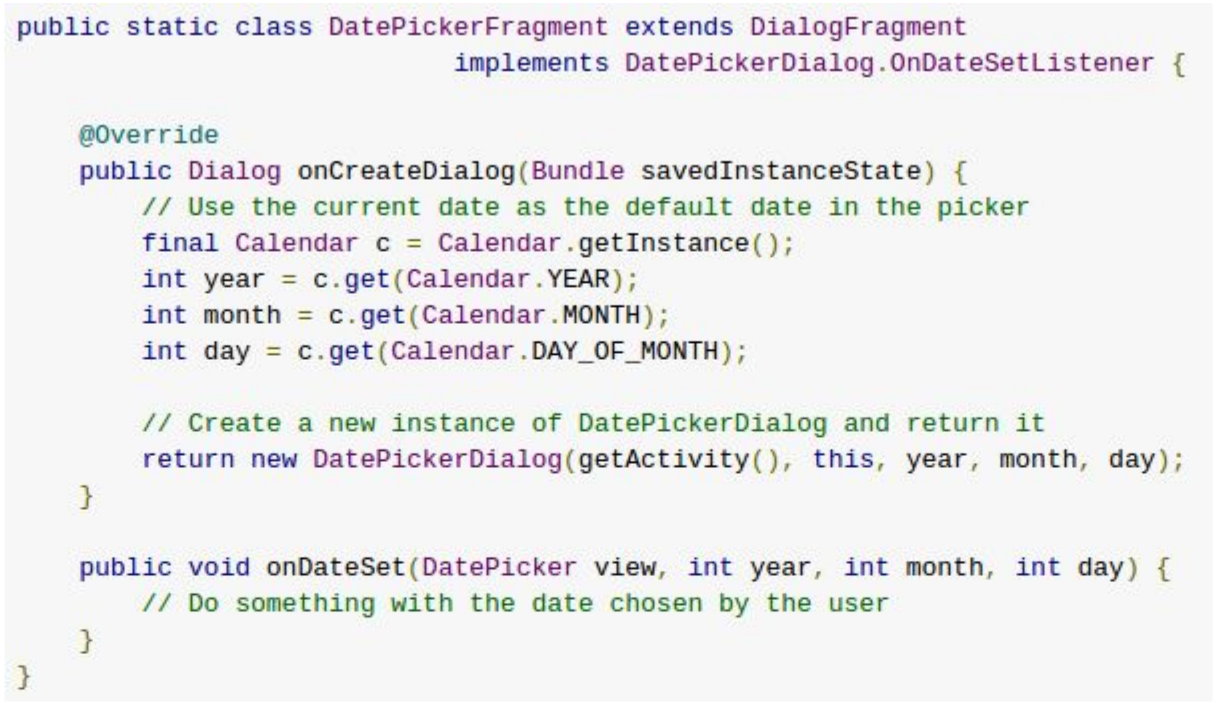
```java
public static class TimePickerFragment extends DialogFragment
                            implements TimePickerDialog.OnTimeSetListener {

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

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

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

Creating a Time Picker

```xml
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/pick_time"
    android:onClick="showTimePickerDialog" />
```

```java
public void showTimePickerDialog(View v) {
    DialogFragment newFragment = new TimePickerFragment();
    newFragment.show(getSupportFragmentManager(), "timePicker");
}
```

Creating a Date Picker



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

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the current date as the default date in the picker
        final Calendar c = Calendar.getInstance();
        int year = c.get(Calendar.YEAR);
        int month = c.get(Calendar.MONTH);
        int day = c.get(Calendar.DAY_OF_MONTH);

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

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

Creating a Date Picker

```xml
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/pick_date"
    android:onClick="showDatePickerDialog" />
```

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

Input Events

Within the various View classes that you'll use to compose your layout, you may notice several **public callback methods** that look useful for **UI events**

An **event listener** is an **interface** in the **View** class that contains a single callback method. These methods will be **called** by the **Android framework** when the View to which the listener has been registered is triggered by user interaction with the item in the UI.

Input Events

onClick(): From View.OnClickListener. This is called when the user either **touches the item** (when in touch mode), or **focuses** upon the item with the navigation-keys or trackball and presses the suitable "enter" key or presses down on the trackball

onLongClick(): From View.OnLongClickListener. This is called when the user either **touches and holds the item** (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses and holds the suitable "enter" key or presses and holds down on the trackball (for one second)

Input Events

onFocusChange(): From View.OnFocusChangeListener. This is called when the **user navigates onto or away from the item**, using the navigation-keys or trackball

onKey(): From View.OnKeyListener. This is called when the user is focused on the item and **presses or releases a hardware key** on the device

Input Events

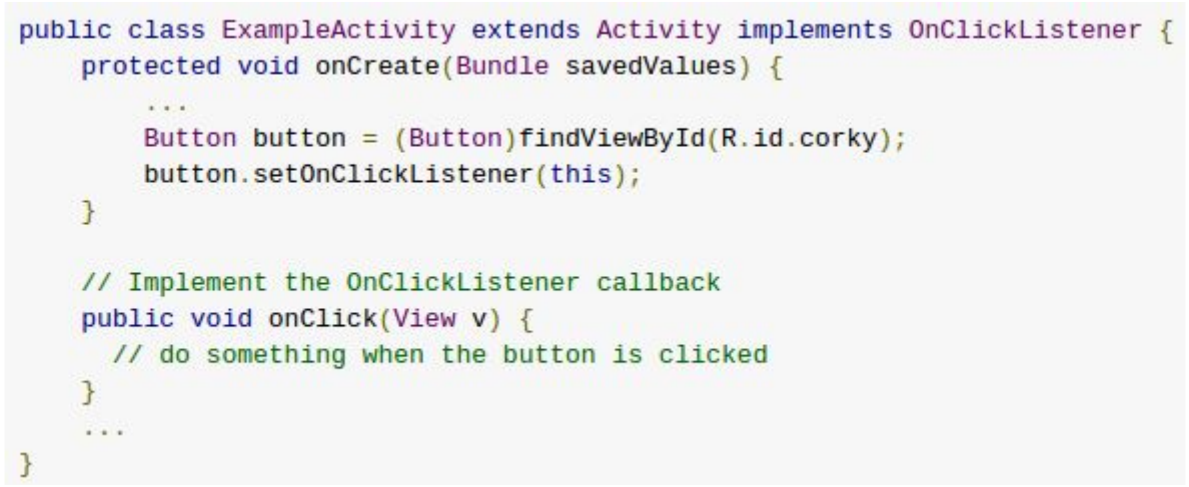
onTouch(): From View.OnTouchListener. This is called when the user **performs an action qualified as a touch event, including a press, a release, or any movement gesture on the screen** (within the bounds of the item)

onCreateContextMenu(): From View.OnCreateContextMenuListener. This is called when a **Context Menu** is being built (as the result of a sustained "long click"). See the discussion on context menus in the Menus developer guide.

Event Listeners

```java
// Create an anonymous implementation of OnClickListener
private OnClickListener mCorkyListener = new OnClickListener() {
    public void onClick(View v) {
      // do something when the button is clicked
    }
};

protected void onCreate(Bundle savedValues) {
    ...
    // Capture our button from layout
    Button button = (Button)findViewById(R.id.corky);
    // Register the onClick listener with the implementation above
    button.setOnClickListener(mCorkyListener);
    ...
}
```

Event Listeners



```java
public class ExampleActivity extends Activity implements OnClickListener {
    protected void onCreate(Bundle savedValues) {
        ...
        Button button = (Button)findViewById(R.id.corky);
        button.setOnClickListener(this);
    }

    // Implement the OnClickListener callback
    public void onClick(View v) {
      // do something when the button is clicked
    }
    ...
}
```

Event Listeners

Notice that the **onClick()** callback in the above example **has no return value**, but some other event listener methods must return a boolean. The reason depends on the event.

o**nLongClick()** and **onKey()** - 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.

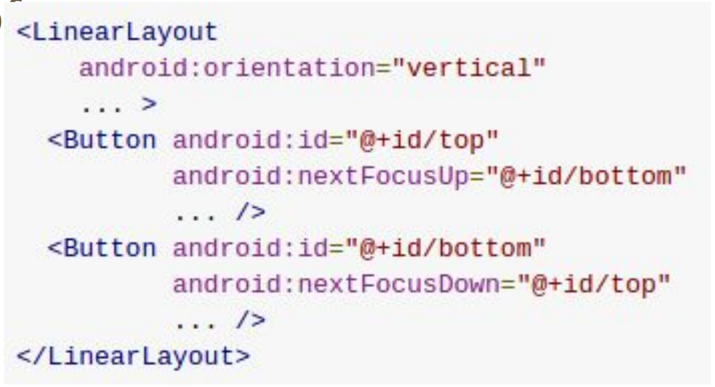
Event Listeners

onTouch() - This returns a boolean to indicate whether your listener consumes this event. **The important thing is that this event can have multiple actions that follow each other**. So, if you return false when the **down action** event is received, you indicate that you have not consumed the event and are also not interested in subsequent actions from this event. Thus, you will not be called for any other actions within the event, such as a **finger gesture**, or the eventual **up action event**.

Handling Focus

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

In rare cases, the default algorithm may not match the intended behavior of the developer. In these situations, you can provide explicit overrides with the following XML attributes in the layout file: **nextFocusDown, nextFocusLeft, nextFocusRight, and nextFocusUp**. Add one o
these attributes to the View from which the
focus is leaving.



```
<LinearLayout
    android:orientation="vertical"
    ... >
  <Button android:id="@+id/top"
          android:nextFocusUp="@+id/bottom"
          ... />
  <Button android:id="@+id/bottom"
          android:nextFocusDown="@+id/top"
          ... />
</LinearLayout>
```

Menus

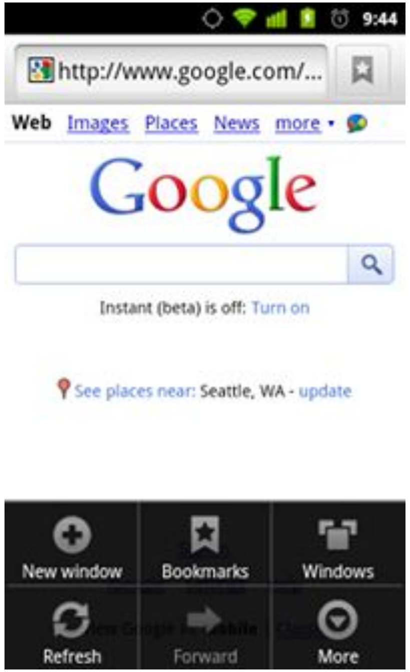
Menus are a common user interface component in many types of applications

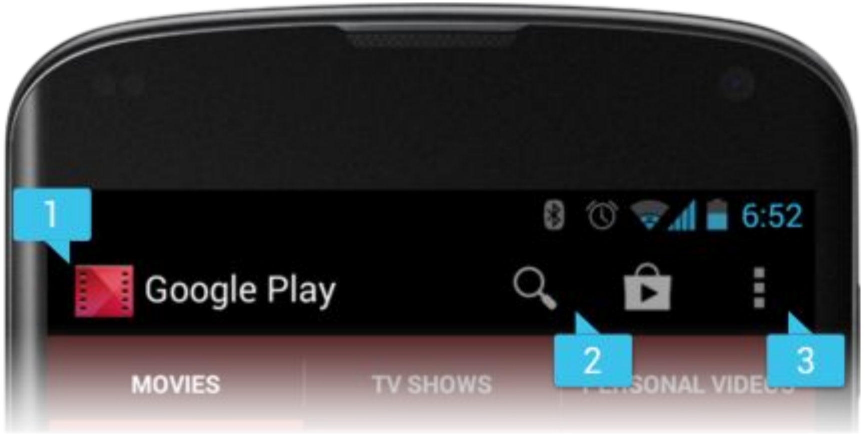
To provide a familiar and consistent user experience, you should use the **Menu APIs** to present user actions and other options in your activities.

three fundamental types of menus or action presentations on all versions of Android:

- Options menu and app bar
- Context menu and contextual action mode
- Popup menu

Options Menu


9:44
http://www.google.com/...
Web
Images
Places
News
more
Google
Instant (beta) is off: Turn on
See places near: Seattle, WA - update
New window
Bookmarks
Windows
Refresh
Forward
More



Options Menu

The options menu is the **primary collection of menu items for an activity**. It's where you should place actions that have a global impact on the app, such as "Search," "Compose email," and "Settings."

- If you're developing for **Android 2.3 or lower**, users can reveal the options menu panel by **pressing the Menu button**

- On **Android 3.0 and higher**, items from the **options menu are presented by the app bar** as a combination of on-screen action items and overflow options. Beginning with Android 3.0, the Menu button is deprecated (some devices don't have one), so you should migrate toward using the action bar to provide access to actions and other option

Defining a Menu in XML

For all menu types, **Android provides a standard XML format to define menu items**

Instead of building a menu in your activity's code, **you should define a menu and all its items in an XML menu resource**

- ➔ It separates the content for the menu from your application's behavioral code
- ➔ It allows you to create alternative menu configurations for different platform versions, screen sizes, and other configurations by leveraging the app resources framework

You can then **inflate the menu resource** (load it as a Menu object) in your activity or fragment

Defining a Menu in XML

To define the menu, create an XML file inside your project's **res/menu/** directory and build the menu with the following elements:

- **<menu>** : Defines a Menu, **which is a container for menu items**. A <menu> element **must be the root node for the file** and can hold one or more <item> and <group> elements.

- **<item>** : Creates a MenuItem, which **represents a single item in a menu**. This element may contain a nested <menu> element in order to create a submenu

- **<group>** : An **optional**, invisible container for <item> elements. It allows you to **categorize menu items** so they share properties such as active state and visibility

an example menu

Here's an example menu named **game_menu.xml**

- **android:icon** A reference to a drawable to use as the item's icon
- **android:title** A reference to a string to use as the item's title
- **android:showAsAction** Specifies when and how this item should appear as an action item in the app bar

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/new_game"
          android:icon="@drawable/ic_new_game"
          android:title="@string/new_game"
          android:showAsAction="ifRoom"/>
    <item android:id="@+id/help"
          android:icon="@drawable/ic_help"
          android:title="@string/help" />
</menu>
```

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.game_menu, menu);
    return true;
}
```

Handling click events

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/new_game"
          android:icon="@drawable/ic_new_game"
          android:title="@string/new_game"
          android:showAsAction="ifRoom"/>
    <item android:id="@+id/help"
          android:icon="@drawable/ic_help"
          android:title="@string/help" />
</menu>
```

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle item selection
    switch (item.getItemId()) {
        case R.id.new_game:
            newGame();
            return true;
        case R.id.help:
            showHelp();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Tip

Android 3.0 adds the ability for you to define the **on-click** behavior for a **menu item in XML**, using the **android:onClick** attribute. The value for the attribute must be the name of a method defined by the activity using the men

if your application **contains multiple activities** and **some of them provide the same options menu**, consider **creating an activity that implements nothing except the onCreateOptionsMenu() and onOptionsItemSelected()** methods. **Then extend this class for each activity** that should share the same options menu.

Changing menu items at runtime

After the system calls **onCreateOptionsMenu()**, it retains an instance of the Menu you populate and will not call onCreateOptionsMenu() again unless the menu is invalidated for some reason. However, **you should use onCreateOptionsMenu() only to create the initial menu state and not to make changes during the activity lifecycle**

If you want to **modify the options menu based on events** that occur during the activity lifecycle, you can do so in the **onPrepareOptionsMenu()** method. This method passes you the Menu object as it currently exists so you can modify it, such as **add, remove, or disable items**. (Fragments also provide an onPrepareOptionsMenu() callback.)

Changing menu items at runtime

On **Android 2.3.x and lower**, the **system calls onPrepareOptionsMenu() each time the user opens the options menu** (presses the Menu button).

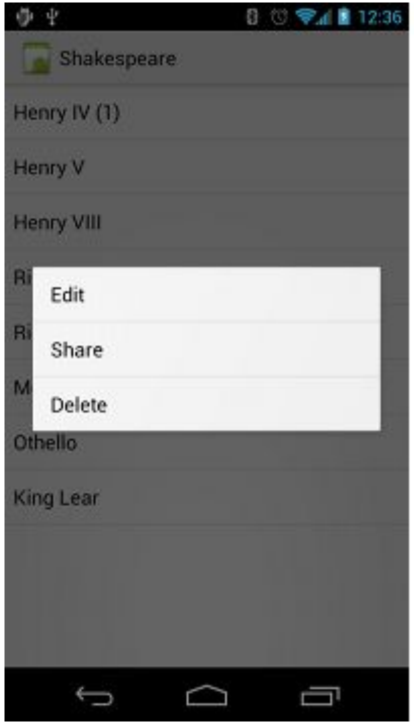
On **Android 3.0 and higher**, the options menu is **considered to always be open when menu items are presented in the app bar**. When an event occurs and you want to perform a menu update, **you must call invalidateOptionsMenu() to request that the system call onPrepareOptionsMenu()**

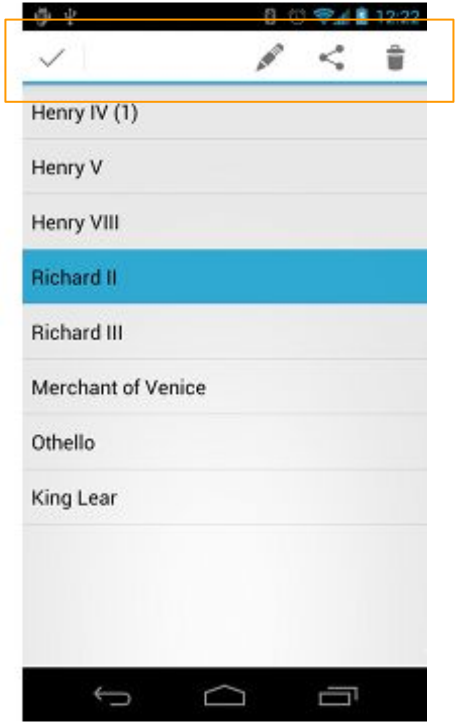
You should **never change items in the options menu based on the View** currently in focus. If you want to provide menu items that are **context-sensitive to a View, use a Context Menu**.

Context menu and contextual action mode

A context menu is a **floating menu that appears when the user performs a long-click on an element**. It provides actions that affect the selected content or context frame

You can provide a context menu for any view, but they are **most often used for items in a ListView, GridView**, or other view collections in which the user can perform direct actions on each item



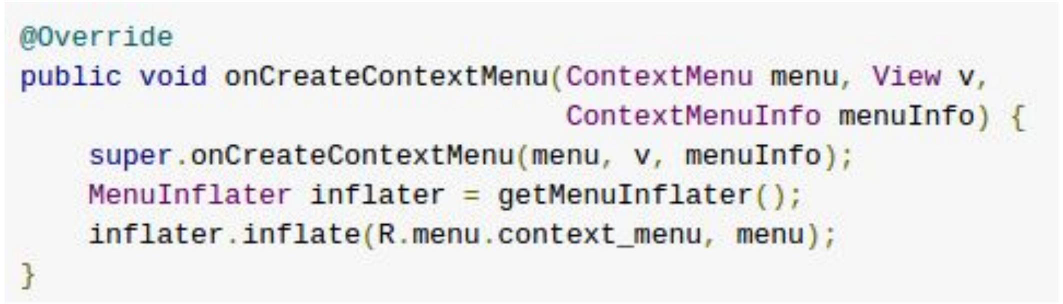


Creating a floating context menu

1. **Register the View to which the context menu should be associated by calling registerForContextMenu() and pass it the View**
   If your activity uses a ListView or GridView and you want each item to provide the same context menu, register all items for a context menu by passing the ListView or GridView to registerForContextMenu()
2. **Implement the onCreateContextMenu() method in your Activity or Fragment**. When the registered view receives a long-click event, the system calls your onCreateContextMenu() method. This is where you define the menu items, usually by inflating a menu resource.

Creating a floating context menu

MenuInflater allows you to inflate the context menu from a menu resource. The callback method parameters include the View that the user selected and a ContextMenu.ContextMenuInfo object that provides additional information about the item selected. If your activity has several views that each provide a different context menu, you might use these parameters to determine which context menu to inflate.



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

Creating a floating context menu

3. Implement **onContextItemSelected()**: When the user selects a menu item, the system calls this method so you can perform the appropriate action.

```java
@Override
public boolean onContextItemSelected(MenuItem item) {
    AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
    switch (item.getItemId()) {
        case R.id.edit:
            editNote(info.id);
            return true;
        case R.id.delete:
            deleteNote(info.id);
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Using the contextual action mode

Implement the **ActionMode.Callback** interface. In its callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other lifecycle events for the action mode

Call **startActionMode() when you want to show the bar** (such as when the user long-clicks the view)

```
private ActionMode.Callback mActionModeCallback = new ActionMode.Callback() {

    // Called when the action mode is created; startActionMode() was called
    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        // Inflate a menu resource providing context menu items
        MenuInflater inflater = mode.getMenuInflater();
        inflater.inflate(R.menu.context_menu, menu);
        return true;
    }

    // Called each time the action mode is shown. Always called after onCreate
    // may be called multiple times if the mode is invalidated.
    @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.menu_share:
                shareCurrentItem();
                mode.finish(); // Action picked, so close the CAB
                return true;
            default:
                return false;
        }
    }
```

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

user long-clicks the view

the mActionMode is used to ensure that the ActionMode instance is not recreated if it's already active, by checking whether the member is null before starting the action mode.

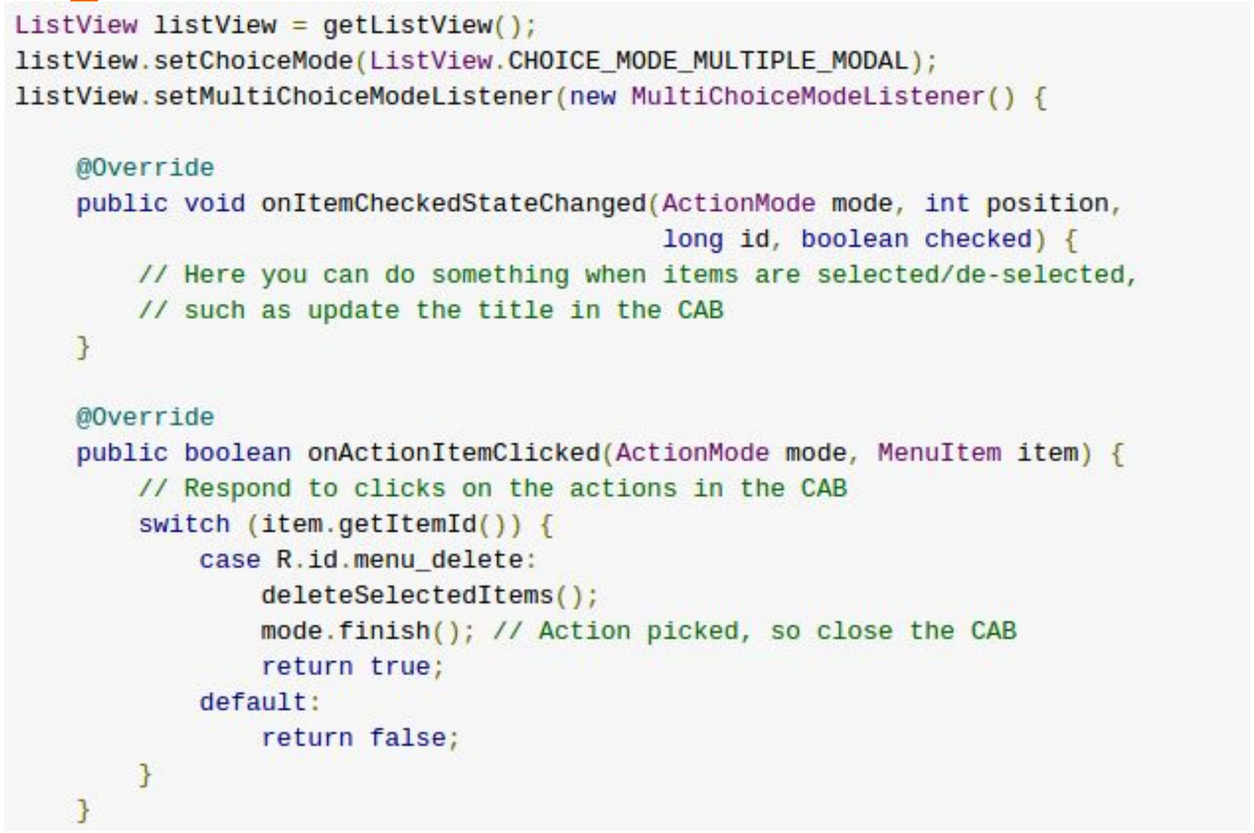
```java
someView.setOnLongClickListener(new View.OnLongClickListener() {
    // Called when the user long-clicks on someView
    public boolean onLongClick(View view) {
        if (mActionMode != null) {
            return false;
        }

        // Start the CAB using the ActionMode.Callback defined above
        mActionMode = getActivity().startActionMode(mActionModeCallback);
        view.setSelected(true);
        return true;
    }
});
```

Enabling batch contextual actions in a ListView or GridView

If you have a collection of items in a ListView or GridView (or another extension of AbsListView) and want to allow users to perform batch actions, you should:

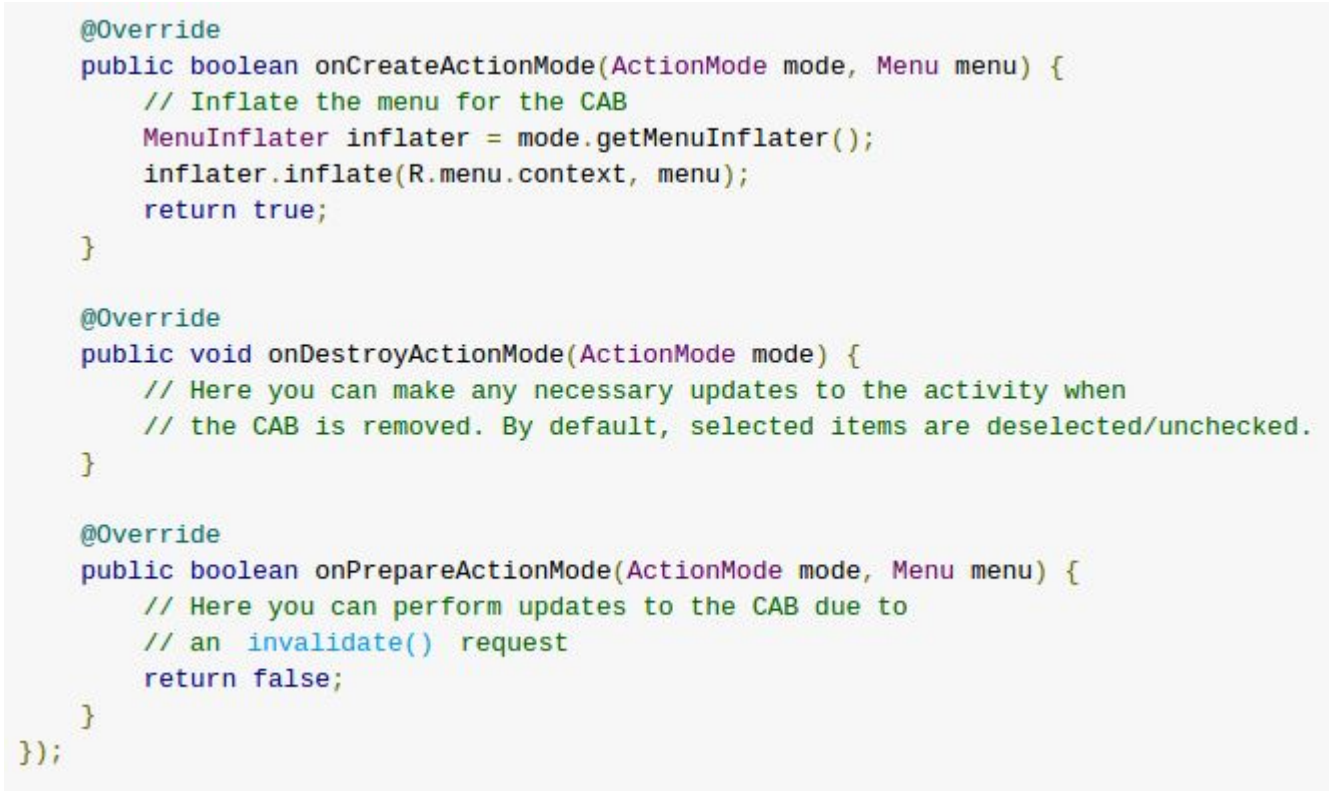
- Implement the AbsListView.MultiChoiceModeListener interface and set it for the view group with setMultiChoiceModeListener()
- Call setChoiceMode() with the CHOICE_MODE_MULTIPLE_MODAL argument

Enabling batch contextual actions in a ListView



```java
ListView listView = getListView();
listView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
listView.setMultiChoiceModeListener(new MultiChoiceModeListener() {

    @Override
    public void onItemCheckedStateChanged(ActionMode mode, int position,
                                          long id, boolean checked) {
        // Here you can do something when items are selected/de-selected,
        // such as update the title in the CAB
    }

    @Override
    public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
        // Respond to clicks on the actions in the CAB
        switch (item.getItemId()) {
            case R.id.menu_delete:
                deleteSelectedItems();
                mode.finish(); // Action picked, so close the CAB
                return true;
            default:
                return false;
        }
    }
```

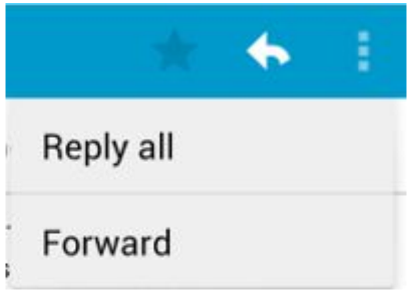
Enabling batch contextual actions in a ListView



```
    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        // Inflate the menu for the CAB
        MenuInflater inflater = mode.getMenuInflater();
        inflater.inflate(R.menu.context, menu);
        return true;
    }

    @Override
    public void onDestroyActionMode(ActionMode mode) {
        // Here you can make any necessary updates to the activity when
        // the CAB is removed. By default, selected items are deselected/unchecked.
    }

    @Override
    public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
        // Here you can perform updates to the CAB due to
        // an  invalidate()  request
        return false;
    }
});
```

Popup menu

A popup menu displays a list of items in a vertical list that's **anchored to the view that invoked the menu**. It's good for providing an overflow of actions that relate to specific content or to **provide options for a second part of a command**. *Actions in a popup menu should not directly affect the corresponding content*—that's what contextual actions are for. Rather, the popup menu is for extended actions that relate to regions of content in your activity



Creating a Popup Menu

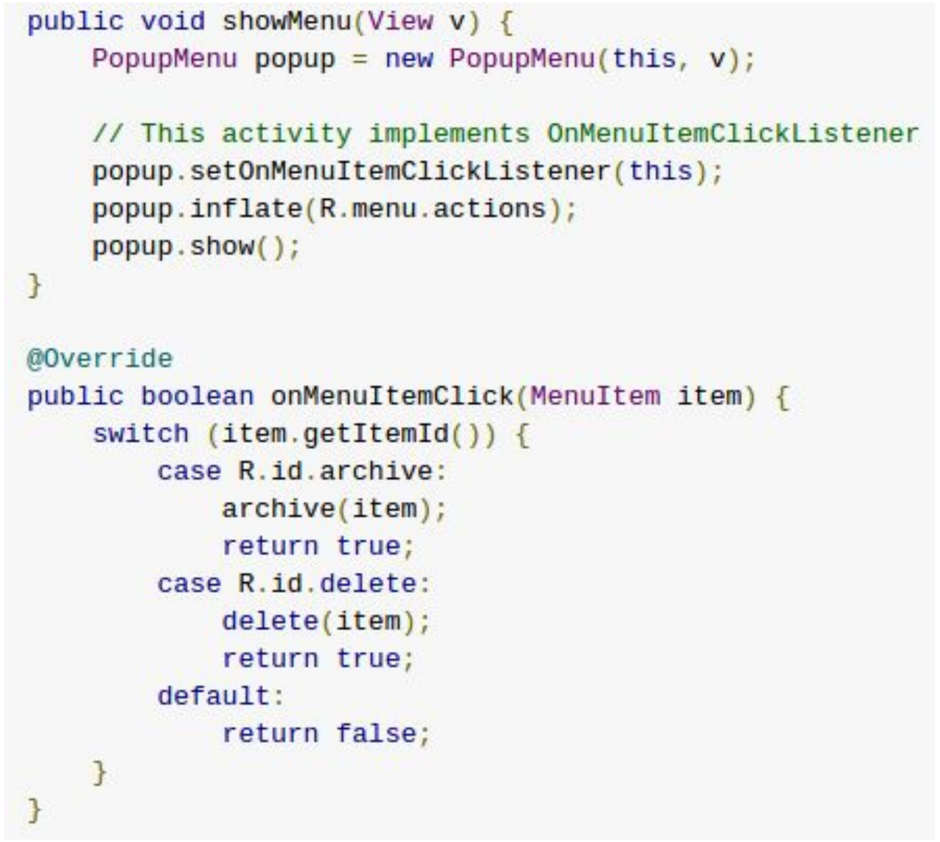
In API level 14 and higher, you can combine the two lines that inflate the menu with PopupMenu.inflate().

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_overflow_holo_dark"
    android:contentDescription="@string/descr_overflow_button"
    android:onClick="showPopup" />
```

```
public void showPopup(View v) {
    PopupMenu popup = new PopupMenu(this, v);
    MenuInflater inflater = popup.getMenuInflater();
    inflater.inflate(R.menu.actions, popup.getMenu());
    popup.show();
}
```

Handling click events

To perform an action when the user selects a menu item, you must implement the **PopupMenu.OnMenuItemClickListener interface** and register it with your PopupMenu by calling setOnMenuItemclickListener(). When the user selects an item, the system calls the onMenuItemClick() callback in your interface.



```
public void showMenu(View v) {
    PopupMenu popup = new PopupMenu(this, v);

    // This activity implements OnMenuItemClickListener
    popup.setOnMenuItemClickListener(this);
    popup.inflate(R.menu.actions);
    popup.show();
}

@Override
public boolean onMenuItemClick(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.archive:
            archive(item);
            return true;
        case R.id.delete:
            delete(item);
            return true;
        default:
            return false;
    }
}
```

Creating Menu Groups

A menu group is a collection of menu items that share certain traits. With a group, you can:

- **Show** or **hide** all items with setGroupVisible()
- **Enable** or **disable** all items with setGroupEnabled()
- Specify whether all items are **checkable** with setGroupCheckable()

Creating Menu Groups

single: Only one item from the group can be checked (radio buttons)
all: All items can be checked (checkboxes)
none: No items are checkable

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_save"
          android:icon="@drawable/menu_save"
          android:title="@string/menu_save" />
    <!-- menu group -->
    <group android:id="@+id/group_delete">
        <item android:id="@+id/menu_archive"
              android:title="@string/menu_archive" />
        <item android:id="@+id/menu_delete"
              android:title="@string/menu_delete" />
    </group>
</menu>
```

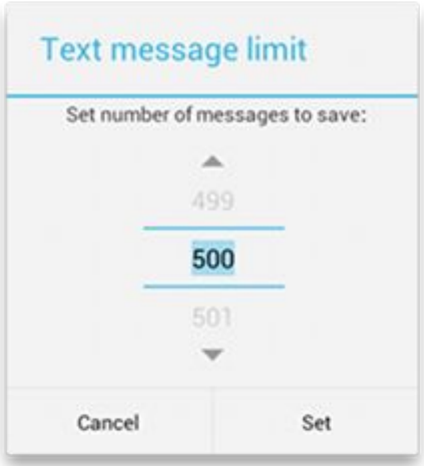
```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/
    <group android:checkableBehavior="single">
        <item android:id="@+id/red"
              android:title="@string/red" />
        <item android:id="@+id/blue"
              android:title="@string/blue" />
    </group>
</menu>
```

Creating Menu Groups

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.vibrate:
        case R.id.dont_vibrate:
            if (item.isChecked()) item.setChecked(false);
            else item.setChecked(true);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Dialogs

A **dialog** is a small **window** that prompts the user to make a decision or enter additional information. A dialog does not fill the screen and is normally used for **modal events** that require users to take an action before they can proceed.


Text message limit

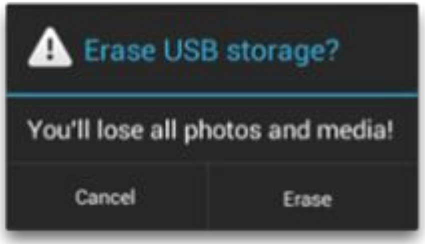
Set number of messages to save:

499

500

501

Cancel Set


Erase USB storage?

You'll lose all photos and media!

Cancel Erase

Dialogs

The Dialog class is the base class for dialogs, **but you should avoid instantiating Dialog directly**. Instead, 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 allows the user to select a date or time**.

Avoid ProgressDialog

Android includes another dialog class called **ProgressDialog** that shows a dialog with a progress bar. **However, if you need to indicate loading or indeterminate progress**, you should instead follow the design guidelines for **Progress & Activity** and use a ProgressBar in your layout.

DialogFragment

These classes define the style and structure for your dialog, but **you should use a DialogFragment as a container for your dialog**. The DialogFragment class **provides all the controls you need** to create your dialog and manage its appearance, instead of calling methods on the Dialog object.

Using DialogFragment to manage the dialog ensures that it **correctly handles lifecycle events such as when the user presses the Back button or rotates the screen**.

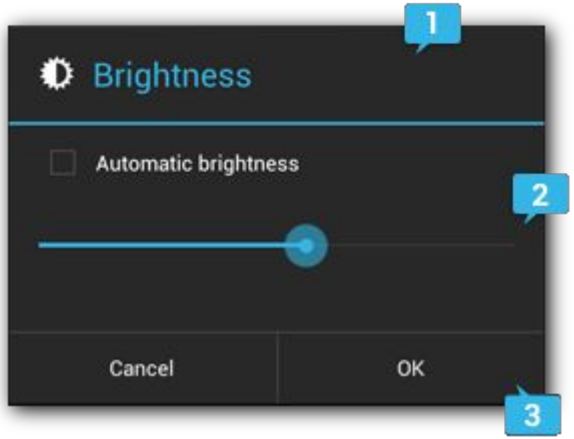
Building an Alert Dialog

The AlertDialog class allows you to build a variety of dialog designs and is often the only dialog class you'll need

Title: This is *optional* and *should be used only when the content area is occupied* by a detailed message, a list, or custom layout. If you need to state a simple message or question, you don't need a title.

Content area: This can *display a message*, a *list*, or other *custom layout*.

Action buttons: There should be *no more than three action buttons in a dialog*.



AlertDialog.Builder

```
// 1. Instantiate an AlertDialog.Builder with its constructor
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());

// 2. Chain together various setter methods to set the dialog characteristics
builder.setMessage(R.string.dialog_message)
       .setTitle(R.string.dialog_title);

// 3. Get the AlertDialog from create()
AlertDialog dialog = builder.create();
```

Adding buttons

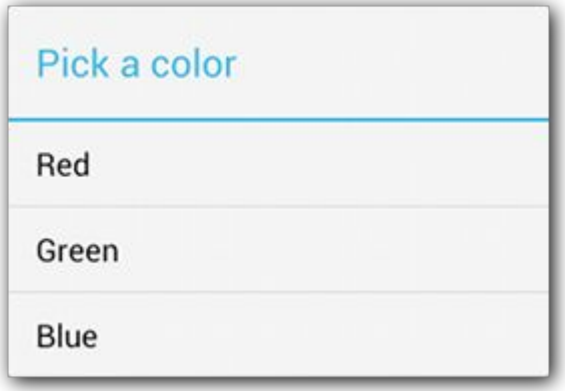
Positive (OK), Negative(Cancel), Neutral(Remind me later)

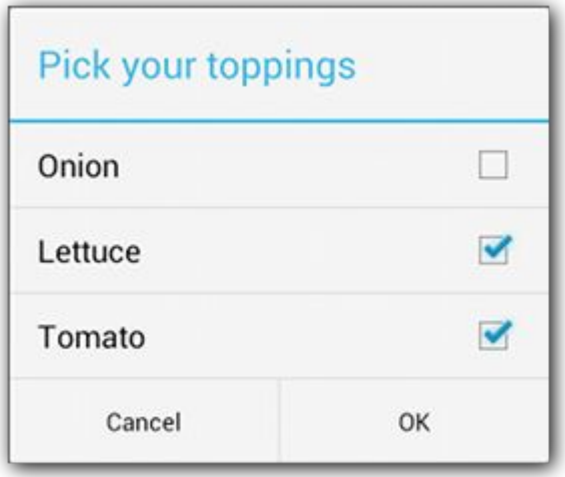
```
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
// Add the buttons
builder.setPositiveButton(R.string.ok, new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {
               // User clicked OK button
           }
       });
builder.setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id) {
               // User cancelled the dialog
           }
       });
// Set other dialog properties
...

// Create the AlertDialog
AlertDialog dialog = builder.create();
```

Adding a list

There are **three kinds of lists available** with the AlertDialog APIs:

A traditional **single-choice list**
A persistent **single-choice list** (**radio buttons**)
A persistent **multiple-choice list** (**checkboxes**)





Single-choice list

```java
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    builder.setTitle(R.string.pick_color)
           .setItems(R.array.colors_array, new DialogInterface.OnClickListener() {
               public void onClick(DialogInterface dialog, int which) {
               // The 'which' argument contains the index position
               // of the selected item
           }
    });
    return builder.create();
}
```

To specify the items for the list, call **setItems()**, **passing an array**. Alternatively, you can specify **a list using setAdapter()**. This allows you to back the list with dynamic data (such as from a database) using a ListAdapter.

tip

If you choose to back your list with a ListAdapter, always **use a Loader so that the content loads asynchronously**. This is described further in Building Layouts with an **Adapter** and the **Loaders** guide.

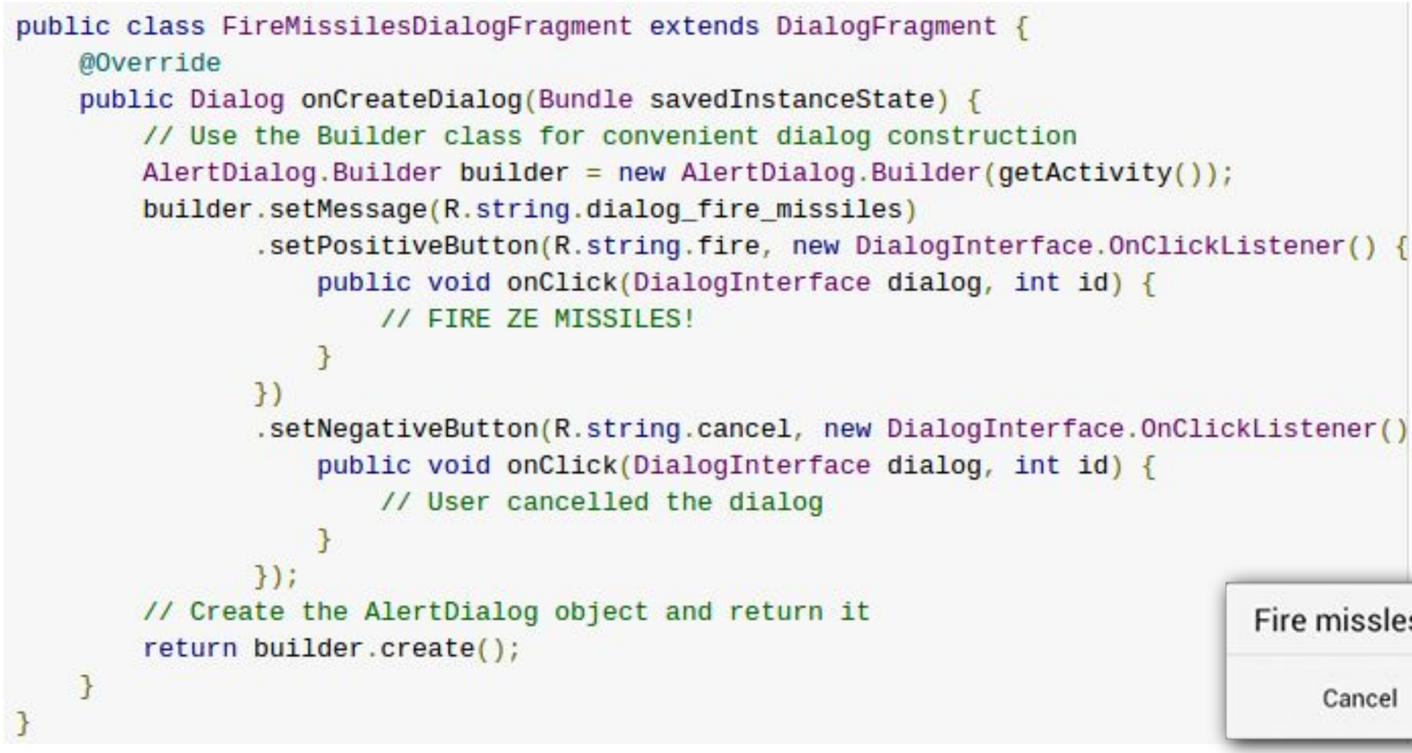
By default, **touching a list item dismisses the dialog**, unless you're using one of the following persistent choice lists.

Adding a persistent multiple-choice or single-choice list

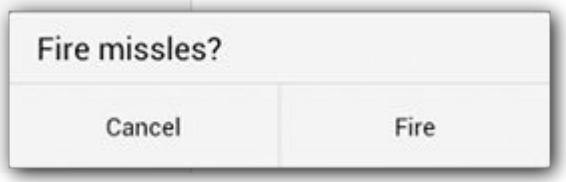
To add a list of multiple-choice items (checkboxes) or single-choice items (radio buttons), use the **setMultiChoiceItems()** or **setSingleChoiceItems()** methods, respectively.

Although both a **traditional list** and a **list with radio buttons** provide a **"single choice" action**, **you should use setSingleChoiceItems() if you want to persist the user's choice**. That is, if opening the dialog again later should indicate what the user's current choice is, then you create a list with radio buttons.

Creating a Dialog Fragment

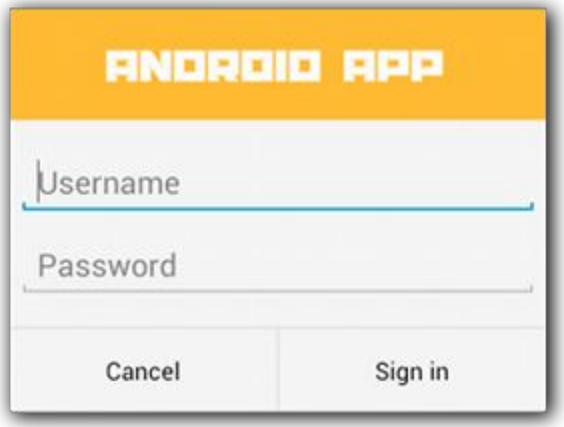


```
public class FireMissilesDialogFragment extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Use the Builder class for convenient dialog construction
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setMessage(R.string.dialog_fire_missiles)
               .setPositiveButton(R.string.fire, new DialogInterface.OnClickListener() {
                   public void onClick(DialogInterface dialog, int id) {
                       // FIRE ZE MISSILES!
                   }
               })
               .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener()
                   public void onClick(DialogInterface dialog, int id) {
                       // User cancelled the dialog
                   }
               });
        // Create the AlertDialog object and return it
        return builder.create();
    }
}
```



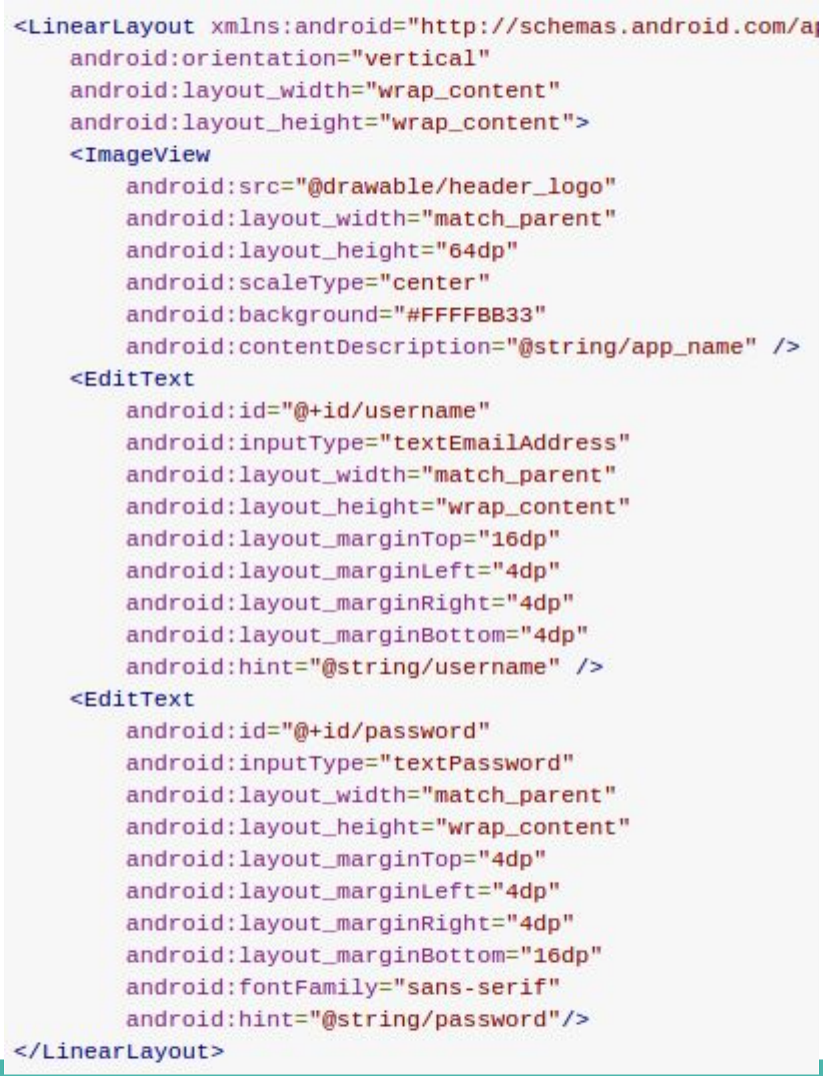
Creating a Custom Layout

If you want a custom layout in a dialog, create a layout and add it to an **AlertDialog** by calling **setView()** on your AlertDialog.Builder object.



Creating a Custom Layout

res/layout/dialog_signin.xml



```
<LinearLayout xmlns:android="http://schemas.android.com/ap
    android:orientation="vertical"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <ImageView
        android:src="@drawable/header_logo"
        android:layout_width="match_parent"
        android:layout_height="64dp"
        android:scaleType="center"
        android:background="#FFFFBB33"
        android:contentDescription="@string/app_name" />
    <EditText
        android:id="@+id/username"
        android:inputType="textEmailAddress"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:layout_marginLeft="4dp"
        android:layout_marginRight="4dp"
        android:layout_marginBottom="4dp"
        android:hint="@string/username" />
    <EditText
        android:id="@+id/password"
        android:inputType="textPassword"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="4dp"
        android:layout_marginLeft="4dp"
        android:layout_marginRight="4dp"
        android:layout_marginBottom="16dp"
        android:fontFamily="sans-serif"
        android:hint="@string/password"/>
</LinearLayout>
```

Creating a Custom Layout

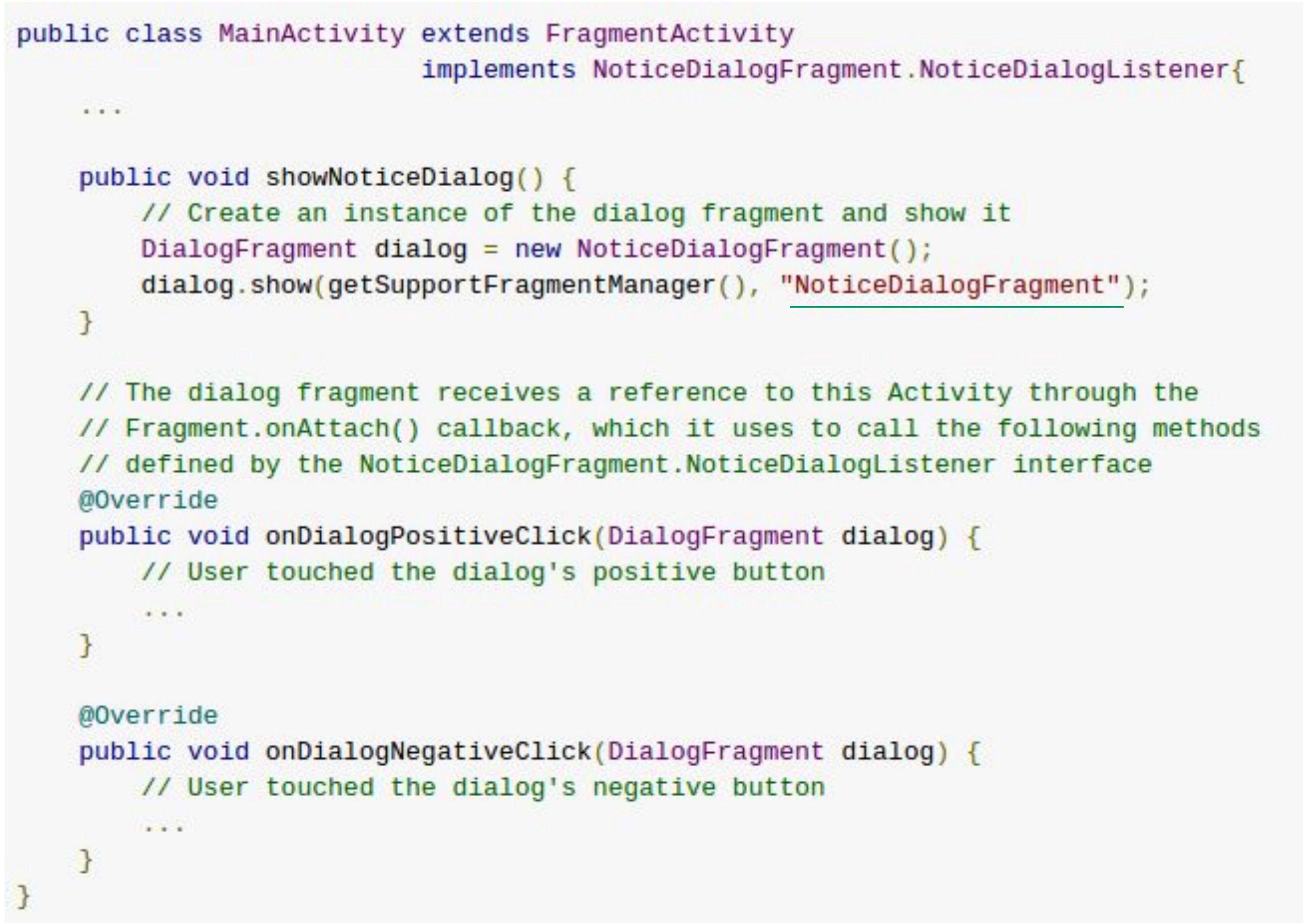
```java
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    // Get the layout inflater
    LayoutInflater inflater = getActivity().getLayoutInflater();

    // Inflate and set the layout for the dialog
    // Pass null as the parent view because its going in the dialog layout
    builder.setView(inflater.inflate(R.layout.dialog_signin, null))
    // Add action buttons
           .setPositiveButton(R.string.signin, new DialogInterface.OnClickListener() {
               @Override
               public void onClick(DialogInterface dialog, int id) {
                   // sign in the user ...
               }
           })
           .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
               public void onClick(DialogInterface dialog, int id) {
                   LoginDialogFragment.this.getDialog().cancel();
               }
           });
    return builder.create();
}
```

Passing Events Back to the Dialog's Host

When the user touches one of the dialog's action buttons or selects an item from its list, your DialogFragment might perform the necessary action itself, but often you'll want to deliver the event to the activity or fragment that opened the dialog. To do this, define an interface with a method for each type of click event. Then implement that interface in the host component that will receive the action events from the dialog.

```java
public class NoticeDialogFragment extends DialogFragment {

    /* The activity that creates an instance of this dialog fragment must
     * implement this interface in order to receive event callbacks.
     * Each method passes the DialogFragment in case the host needs to query it. */
    public interface NoticeDialogListener {
        public void onDialogPositiveClick(DialogFragment dialog);
        public void onDialogNegativeClick(DialogFragment dialog);
    }

    // Use this instance of the interface to deliver action events
    NoticeDialogListener mListener;

    // Override the Fragment.onAttach() method to instantiate the NoticeDialogListener
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        // Verify that the host activity implements the callback interface
        try {
            // Instantiate the NoticeDialogListener so we can send events to the host
            mListener = (NoticeDialogListener) activity;
        } catch (ClassCastException e) {
            // The activity doesn't implement the interface, throw exception
            throw new ClassCastException(activity.toString()
                    + " must implement NoticeDialogListener");
        }
    }
    ...
}
```

The second argument, is a unique tag name that the system uses to save and restore the fragment state when necessary. The tag also allows you to get a handle to the fragment by calling findFragmentByTag().



```
public class MainActivity extends FragmentActivity
                          implements NoticeDialogFragment.NoticeDialogListener{
    ...

    public void showNoticeDialog() {
        // Create an instance of the dialog fragment and show it
        DialogFragment dialog = new NoticeDialogFragment();
        dialog.show(getSupportFragmentManager(), "NoticeDialogFragment");
    }

    // The dialog fragment receives a reference to this Activity through the
    // Fragment.onAttach() callback, which it uses to call the following methods
    // defined by the NoticeDialogFragment.NoticeDialogListener interface
    @Override
    public void onDialogPositiveClick(DialogFragment dialog) {
        // User touched the dialog's positive button
        ...
    }

    @Override
    public void onDialogNegativeClick(DialogFragment dialog) {
        // User touched the dialog's negative button
        ...
    }
}
```

```
public class NoticeDialogFragment extends DialogFragment {
    ...

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Build the dialog and set up the button click handlers
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setMessage(R.string.dialog_fire_missiles)
               .setPositiveButton(R.string.fire, new DialogInterface.OnClickListener() {
                   public void onClick(DialogInterface dialog, int id) {
                       // Send the positive button event back to the host activity
                       mListener.onDialogPositiveClick(NoticeDialogFragment.this);
                   }
               })
               .setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener()
                   public void onClick(DialogInterface dialog, int id) {
                       // Send the negative button event back to the host activity
                       mListener.onDialogNegativeClick(NoticeDialogFragment.this);
                   }
               });
        return builder.create();
    }
}
```

Dismissing a Dialog

When the **user touches any of the action buttons** created with an AlertDialog.Builder(or **presses the Back button**, t**ouches the screen outside the dialog area**, or if you **explicitly call cancel()** on the Dialog), the system **dismisses the dialog** for you.

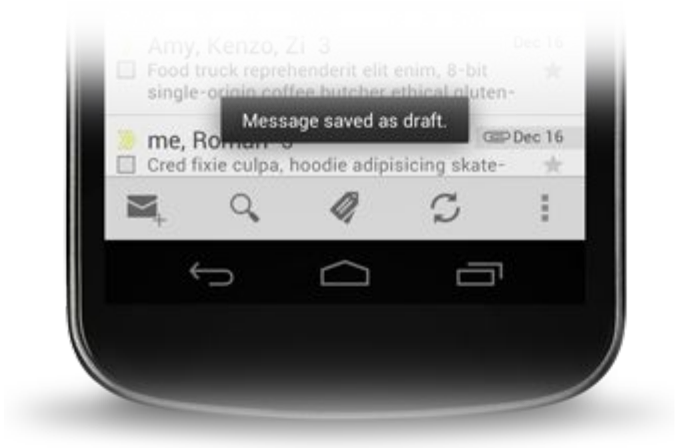
The system also dismisses the dialog when the user touches an item in a dialog list, except when the list uses radio buttons or checkboxes.

In case you need to perform certain actions when the dialog goes away, you can implement the **onDismiss()** method in your DialogFragment.

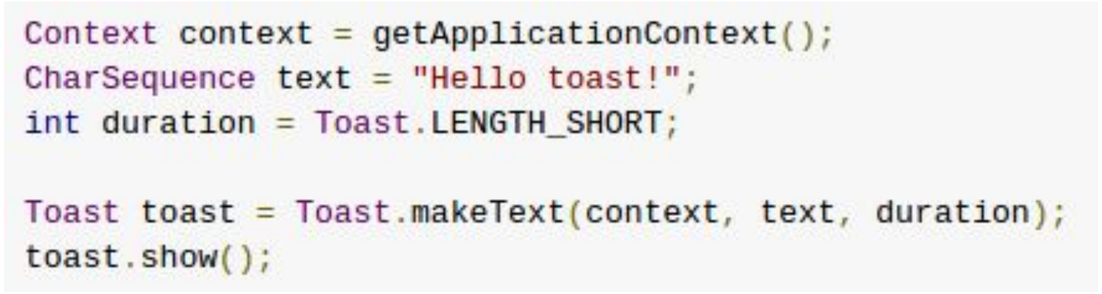
The system calls onDismiss() upon each event that invokes the onCancel() callback.

Toasts

A toast provides simple feedback about an operation in a small popup. It only fills the amount of space required for the message and the current activity remains visible and interactive.


Amy, Kenzo, Zi 3
Food truck reprehenderit elit enim, 8-bit
single-origin coffee butcher ethical gluten-
Message saved as draft.
me, Roman 3
Dec 16
Cred fixie culpa, hoodie adipisicing skate-

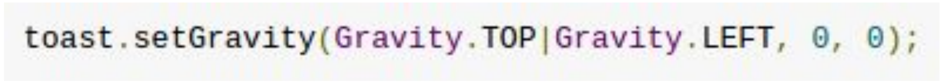
The Basics



```java
Context context = getApplicationContext();
CharSequence text = "Hello toast!";
int duration = Toast.LENGTH_SHORT;

Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

Positioning your Toast

A standard toast notification appears near the bottom of the screen, centered horizontally. You can change this position with the setGravity(int, int, int) method. This accepts three parameters: a Gravity constant, an x-position offset, and a y-position offset.



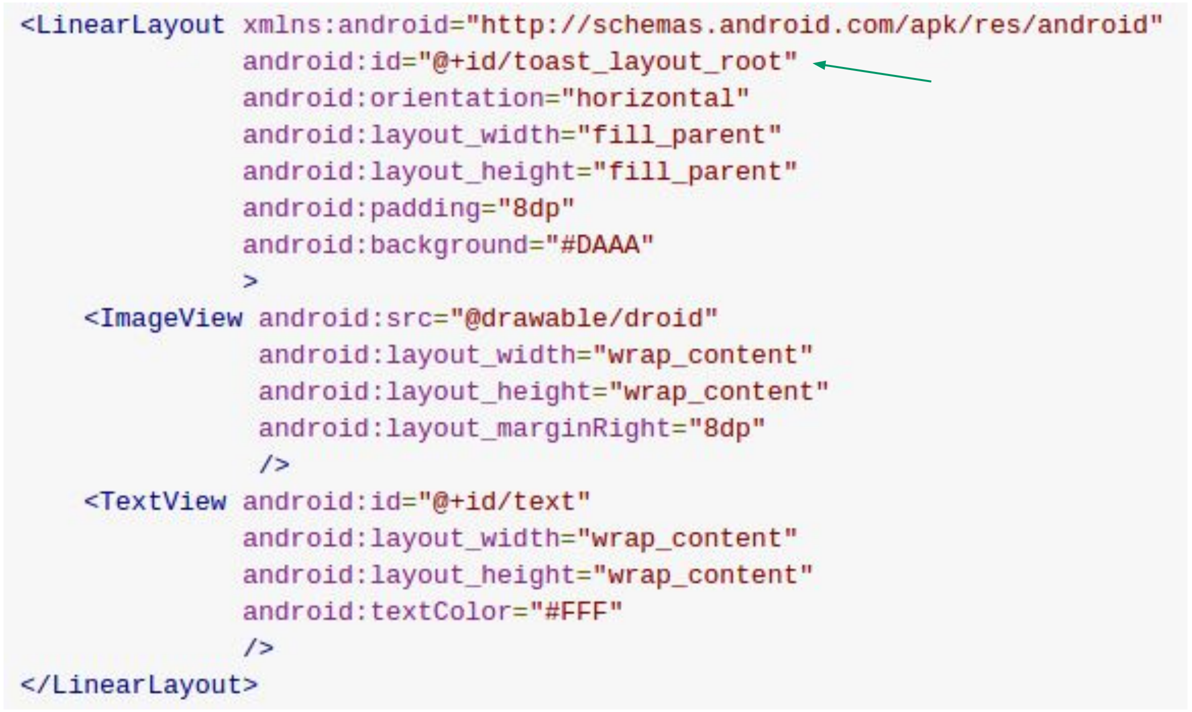
```
toast.setGravity(Gravity.TOP|Gravity.LEFT, 0, 0);
```

Creating a Custom Toast View

If a simple text message isn't enough, you can create a customized layout for your toast notification. To create a custom layout, define a View layout, in XML or in your application code, and pass the root View object to the setView(View) method.

Creating a Custom Toast View

toast_layout.xml



```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:id="@+id/toast_layout_root"
              android:orientation="horizontal"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:padding="8dp"
              android:background="#DAAA"
              >
    <ImageView android:src="@drawable/droid"
               android:layout_width="wrap_content"
               android:layout_height="wrap_content"
               android:layout_marginRight="8dp"
               />
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:textColor="#FFF"
              />
</LinearLayout>
```

Creating a Custom Toast View

```
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.custom_toast,
                               (ViewGroup) findViewById(R.id.toast_layout_root));

TextView text = (TextView) layout.findViewById(R.id.text);
text.setText("This is a custom toast");

Toast toast = new Toast(getApplicationContext());
toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(layout);
toast.show();
```

Styles and Themes

A style is a **collection of properties** that **specify the look and format for a View or window**. A style can specify properties such as height, padding, font color, font size, background color, and much more. *A style is defined in an XML resource that is separate from the XML that specifies the layout*.

Styles in Android share a similar philosophy to cascading stylesheets in web design—they allow you to **separate the design from the content**.

Defining Styles

To create a set of styles, save an XML file in the **res/values/** directory of your project. The name of the XML file is arbitrary, but it must use the .xml extension and be saved in the res/values/ folder.

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="#00FF00"
    android:typeface="monospace"
    android:text="@string/hello" />
```

```
<TextView
    style="@style/CodeFont"
    android:text="@string/hello" />
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CodeFont" parent="@android:style/TextAppearance.Medium">
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

Inheritance

You can inherit from styles that you've created yourself or from styles that are built into the platform

```
<style name="GreenText" parent="@android:style/TextAppearance">
    <item name="android:textColor">#00FF00</item>
</style>
```

If you want to **inherit from styles that you've defined yourself**, you **do not have to use the parent attribute**. Instead, just **prefix the name of the style you want to inherit to the name of your new style, separated by a period**. For example, to create a new style that inherits the CodeFont style defined above, but make the color red, you can author the new style like this

```
<style name="CodeFont.Red">
    <item name="android:textColor">#FF0000</item>
</style>
```