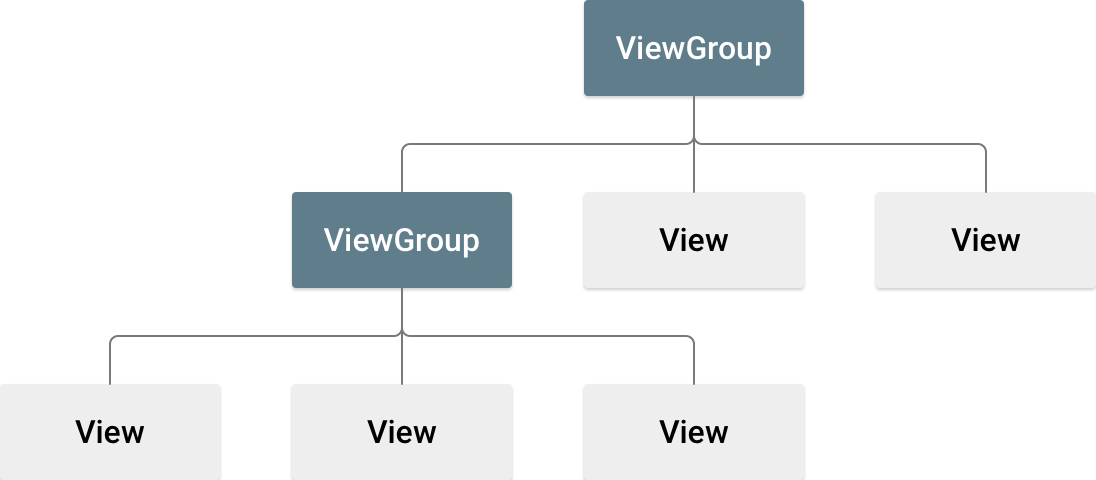
# **UI Overview**

All user interface elements in an Android app are built using [View](https://developer.android.com/reference/android/view/View.html) and [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) objects. A [View](https://developer.android.com/reference/android/view/View.html) is an object that draws something on the screen that the user can interact with. A [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) is an object that holds other [View](https://developer.android.com/reference/android/view/View.html) (and [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html)) objects in order to define the layout of the interface.

Android provides a collection of both [View](https://developer.android.com/reference/android/view/View.html) and [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) subclasses that offer you common input controls (such as buttons and text fields) and various layout models (such as a linear or relative layout).

## User Interface Layout

The user interface for each component of your app is defined using a hierarchy of [View](https://developer.android.com/reference/android/view/View.html) and [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) objects, as shown in figure 1. Each view group is an invisible container that organizes child views, while the child views may be input controls or other widgets that draw some part of the UI. This hierarchy tree can be as simple or complex as you need it to be (but simplicity is best for performance).



**Figure 1.** Illustration of a view hierarchy, which defines a UI layout.

To declare your layout, you can instantiate [View](https://developer.android.com/reference/android/view/View.html) 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](https://developer.android.com/reference/android/widget/TextView.html) widget in your UI, and a <LinearLayout> element creates a [LinearLayout](https://developer.android.com/reference/android/widget/LinearLayout.html) view group.

For example, a simple vertical layout with a text view and a button looks like this:

<?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>

When you load a layout resource in your app, Android initializes each node of the layout into a runtime object you can use to define additional behaviors, query the object state, or modify the layout.

For a complete guide to creating a UI layout, see [XML Layouts](https://developer.android.com/guide/topics/ui/declaring-layout.html).

## User Interface Components

You don't have to build all of your UI using [View](https://developer.android.com/reference/android/view/View.html) and [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) objects. Android provides several app components that offer a standard UI layout for which you simply need to define the content. These UI components each have a unique set of APIs that are described in their respective documents, such as [Adding the App Bar](https://developer.android.com/training/appbar/index.html), [Dialogs](https://developer.android.com/guide/topics/ui/dialogs.html), and [Status Notifications](https://developer.android.com/guide/topics/ui/notifiers/notifications.html).

# **Layouts**

A layout defines the visual structure for a user interface, such as the UI for an [activity](https://developer.android.com/guide/components/activities.html) or [app widget](https://developer.android.com/guide/topics/appwidgets/index.html). 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.

The Android framework gives you the flexibility to use either or both of these methods for declaring and managing your application's UI. For example, you could declare your application's default layouts in XML, including the screen elements that will appear in them and their properties. You could then add code in your application that would modify the state of the screen objects, including those declared in XML, at run time.

* You should also try the [Hierarchy Viewer](https://developer.android.com/tools/debugging/debugging-ui.html#hierarchyViewer) tool, for debugging layouts — it reveals layout property values, draws wireframes with padding/margin indicators, and full rendered views while you debug on the emulator or device.
* The [layoutopt](https://developer.android.com/tools/debugging/debugging-ui.html" \l "layoutopt) tool lets you quickly analyze your layouts and hierarchies for inefficiencies or other problems.

The advantage to declaring your UI in XML is that it enables you to better separate the presentation of your application from the code that controls its behavior. Your UI descriptions are external to your application code, which means that you can modify or adapt it without having to modify your source code and recompile. For example, you can create XML layouts for different screen orientations, different device screen sizes, and different languages. Additionally, declaring the layout in XML makes it easier to visualize the structure of your UI, so it's easier to debug problems. As such, this document focuses on teaching you how to declare your layout in XML. If you're interested in instantiating View objects at runtime, refer to the [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) and [View](https://developer.android.com/reference/android/view/View.html) class references.

In general, the XML vocabulary for declaring UI elements closely follows the structure and naming of the classes and methods, where element names correspond to class names and attribute names correspond to methods. In fact, the correspondence is often so direct that you can guess what XML attribute corresponds to a class method, or guess what class corresponds to a given XML element. However, note that not all vocabulary is identical. In some cases, there are slight naming differences. For example, the EditText element has a text attribute that corresponds to EditText.setText().

**Tip:** Learn more about different layout types in [Common Layout Objects](https://developer.android.com/guide/topics/ui/layout-objects.html).

## Write the XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout. For example, here's an XML layout that uses a vertical [LinearLayout](https://developer.android.com/reference/android/widget/LinearLayout.html) to hold a [TextView](https://developer.android.com/reference/android/widget/TextView.html) and a [Button](https://developer.android.com/reference/android/widget/Button.html):

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
              android:layout\_width="match\_parent"  
              android:layout\_height="match\_parent"  
              android:orientation="vertical" >  
    <TextView android:id="@+id/text"  
              android:layout\_width="wrap\_content"  
              android:layout\_height="wrap\_content"  
              android:text="Hello, I am a TextView" />  
    <Button android:id="@+id/button"  
            android:layout\_width="wrap\_content"  
            android:layout\_height="wrap\_content"  
            android:text="Hello, I am a Button" />  
</LinearLayout>

After you've declared your layout in XML, save the file with the .xml extension, in your Android project's res/layout/ directory, so it will properly compile.

More information about the syntax for a layout XML file is available in the [Layout Resources](https://developer.android.com/guide/topics/resources/layout-resource.html) document.

## Load the XML Resource

When you compile your application, each XML layout file is compiled into a [View](https://developer.android.com/reference/android/view/View.html) resource. You should load the layout resource from your application code, in your [Activity.onCreate()](https://developer.android.com/reference/android/app/Activity.html" \l "onCreate(android.os.Bundle)) callback implementation. Do so by calling [setContentView()](https://developer.android.com/reference/android/app/Activity.html" \l "setContentView(int)), passing it the reference to your layout resource in the form of: R.layout.layout\_file\_name. For example, if your XML layout is saved as main\_layout.xml, you would load it for your Activity like so:

public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main\_layout);  
}

The onCreate() callback method in your Activity is called by the Android framework when your Activity is launched (see the discussion about lifecycles, in the [Activities](https://developer.android.com/guide/components/activities.html#Lifecycle) document).

## Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, TextView supports the textSize attribute), but these attributes are also inherited by any View objects that may extend this class. Some are common to all View objects, because they are inherited from the root View class (like the id attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

### ID

Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the application is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the id attribute. This is an XML attribute common to all View objects (defined by the [View](https://developer.android.com/reference/android/view/View.html) class) and you will use it very often. The syntax for an ID, inside an XML tag is:

android:id="@+id/my\_button"

The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource. The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the R.java file). There are a number of other ID resources that are offered by the Android framework. When referencing an Android resource ID, you do not need the plus-symbol, but must add the android package namespace, like so:

android:id="@android:id/empty"

With the android package namespace in place, we're now referencing an ID from the android.R resources class, rather than the local resources class.

In order to create views and reference them from the application, a common pattern is to:

1. Define a view/widget in the layout file and assign it a unique ID:

<Button android:id="@+id/my\_button"  
        android:layout\_width="wrap\_content"  
        android:layout\_height="wrap\_content"  
        android:text="@string/my\_button\_text"/>

1. Then create an instance of the view object and capture it from the layout (typically in the [onCreate()](https://developer.android.com/reference/android/app/Activity.html" \l "onCreate(android.os.Bundle)) method):

Button myButton = (Button) findViewById(R.id.my\_button);

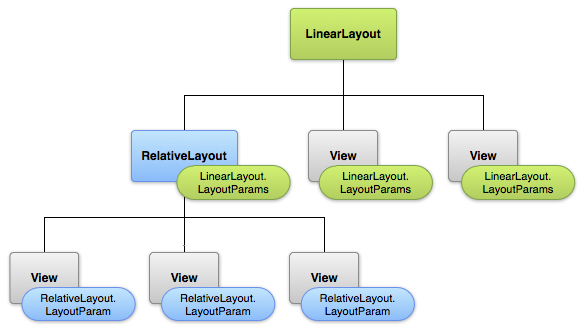
Defining IDs for view objects is important when creating a [RelativeLayout](https://developer.android.com/reference/android/widget/RelativeLayout.html). In a relative layout, sibling views can define their layout relative to another sibling view, which is referenced by the unique ID.

An ID need not be unique throughout the entire tree, but it should be unique within the part of the tree you are searching (which may often be the entire tree, so it's best to be completely unique when possible).

### Layout Parameters

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

Every ViewGroup class implements a nested class that extends [ViewGroup.LayoutParams](https://developer.android.com/reference/android/view/ViewGroup.LayoutParams.html). This subclass contains property types that define the size and position for each child view, as appropriate for the view group. As you can see in figure 1, the parent view group defines layout parameters for each child view (including the child view group).



**Figure 1.** Visualization of a view hierarchy with layout parameters associated with each view.

Note that every LayoutParams subclass has its own syntax for setting values. Each child element must define LayoutParams that are appropriate for its parent, though it may also define different LayoutParams for its own children.

All view groups include a width and height (layout\_width and layout\_height), and each view is required to define them. Many LayoutParams also include optional margins and borders.

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** tells your view to become as big as its parent view group will allow.

In general, specifying a layout width and height using absolute units such as pixels is not recommended. Instead, using relative measurements such as density-independent pixel units (**dp**), **wrap\_content**, or **match\_parent**, is a better approach, because it helps ensure that your application will display properly across a variety of device screen sizes. The accepted measurement types are defined in the [Available Resources](https://developer.android.com/guide/topics/resources/available-resources.html#dimension) document.

## 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()](https://developer.android.com/reference/android/view/View.html" \l "getLeft()) and [getTop()](https://developer.android.com/reference/android/view/View.html" \l "getTop()). The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view. 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.

In addition, several convenience methods are offered to avoid unnecessary computations, namely [getRight()](https://developer.android.com/reference/android/view/View.html" \l "getRight()) and [getBottom()](https://developer.android.com/reference/android/view/View.html" \l "getBottom()). These methods return the coordinates of the right and bottom edges of the rectangle representing the view. For instance, calling [getRight()](https://developer.android.com/reference/android/view/View.html" \l "getRight()) is similar to the following computation: getLeft() + getWidth().

## Size, Padding and Margins

The size of a view is expressed with a width and a height. A view actually possess two pairs of width and height values.

The first pair is known as measured width and measured height. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling [getMeasuredWidth()](https://developer.android.com/reference/android/view/View.html" \l "getMeasuredWidth()) and [getMeasuredHeight()](https://developer.android.com/reference/android/view/View.html" \l "getMeasuredHeight()).

The second pair is simply known as width and height, or sometimes drawing width and drawing height. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling [getWidth()](https://developer.android.com/reference/android/view/View.html" \l "getWidth()) and [getHeight()](https://developer.android.com/reference/android/view/View.html" \l "getHeight()).

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific number of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the [setPadding(int, int, int, int)](https://developer.android.com/reference/android/view/View.html" \l "setPadding(int, int, int, int)) method and queried by calling [getPaddingLeft()](https://developer.android.com/reference/android/view/View.html#getPaddingLeft()), [getPaddingTop()](https://developer.android.com/reference/android/view/View.html#getPaddingTop()), [getPaddingRight()](https://developer.android.com/reference/android/view/View.html#getPaddingRight()) and [getPaddingBottom()](https://developer.android.com/reference/android/view/View.html#getPaddingBottom()).

Even though a view can define a padding, it does not provide any support for margins. However, view groups provide such a support. Refer to [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html)and [ViewGroup.MarginLayoutParams](https://developer.android.com/reference/android/view/ViewGroup.MarginLayoutParams.html) for further information.

For more information about dimensions, see [Dimension Values](https://developer.android.com/guide/topics/resources/more-resources.html#Dimension).

## Common Layouts

Each subclass of the [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.

**Note:** Although you can nest one or more layouts within another layout to acheive your UI design, you should strive to keep your layout hierarchy as shallow as possible. Your layout draws faster if it has fewer nested layouts (a wide view hierarchy is better than a deep view hierarchy).

#### [**Linear Layout**](https://developer.android.com/guide/topics/ui/layout/linear.html)

[](https://developer.android.com/guide/topics/ui/layout/linear.html)

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**](https://developer.android.com/guide/topics/ui/layout/relative.html)

[](https://developer.android.com/guide/topics/ui/layout/relative.html)

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**](https://developer.android.com/guide/webapps/webview.html)

[](https://developer.android.com/guide/webapps/webview.html)

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](https://developer.android.com/reference/android/widget/AdapterView.html) to populate the layout with views at runtime. A subclass of the [AdapterView](https://developer.android.com/reference/android/widget/AdapterView.html) class uses an [Adapter](https://developer.android.com/reference/android/widget/Adapter.html) to bind data to its layout. The [Adapter](https://developer.android.com/reference/android/widget/Adapter.html) behaves as a middleman between the data source and the [AdapterView](https://developer.android.com/reference/android/widget/AdapterView.html) layout—the [Adapter](https://developer.android.com/reference/android/widget/Adapter.html) 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](https://developer.android.com/reference/android/widget/AdapterView.html) layout.

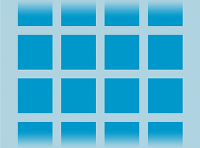
Common layouts backed by an adapter include:

#### [**List View**](https://developer.android.com/guide/topics/ui/layout/listview.html)

[](https://developer.android.com/guide/topics/ui/layout/listview.html)

Displays a scrolling single column list.

#### [**Grid View**](https://developer.android.com/guide/topics/ui/layout/gridview.html)

[](https://developer.android.com/guide/topics/ui/layout/gridview.html)

Displays a scrolling grid of columns and rows.

### Filling an adapter view with data

You can populate an [AdapterView](https://developer.android.com/reference/android/widget/AdapterView.html) such as [ListView](https://developer.android.com/reference/android/widget/ListView.html) or [GridView](https://developer.android.com/reference/android/widget/GridView.html) by binding the [AdapterView](https://developer.android.com/reference/android/widget/AdapterView.html) instance to an [Adapter](https://developer.android.com/reference/android/widget/Adapter.html), which retrieves data from an external source and creates a [View](https://developer.android.com/reference/android/view/View.html) that represents each data entry.

Android provides several subclasses of [Adapter](https://developer.android.com/reference/android/widget/Adapter.html) that are useful for retrieving different kinds of data and building views for an [AdapterView](https://developer.android.com/reference/android/widget/AdapterView.html). The two most common adapters are:

[ArrayAdapter](https://developer.android.com/reference/android/widget/ArrayAdapter.html)

Use this adapter when your data source is an array. By default, [ArrayAdapter](https://developer.android.com/reference/android/widget/ArrayAdapter.html) creates a view for each array item by calling [toString()](https://developer.android.com/reference/java/lang/Object.html" \l "toString()) on each item and placing the contents in a [TextView](https://developer.android.com/reference/android/widget/TextView.html).

For example, if you have an array of strings you want to display in a [ListView](https://developer.android.com/reference/android/widget/ListView.html), initialize a new [ArrayAdapter](https://developer.android.com/reference/android/widget/ArrayAdapter.html) using a constructor to specify the layout for each string and the string array:

ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,  
        android.R.layout.simple\_list\_item\_1, myStringArray);

The arguments for this constructor are:

* Your app [Context](https://developer.android.com/reference/android/content/Context.html)
* The layout that contains a [TextView](https://developer.android.com/reference/android/widget/TextView.html) for each string in the array
* The string array

Then simply call [setAdapter()](https://developer.android.com/reference/android/widget/AdapterView.html" \l "setAdapter(T)) on your [ListView](https://developer.android.com/reference/android/widget/ListView.html):

ListView listView = (ListView) findViewById(R.id.listview);  
listView.setAdapter(adapter);

To customize the appearance of each item you can override the [toString()](https://developer.android.com/reference/java/lang/Object.html" \l "toString()) method for the objects in your array. Or, to create a view for each item that's something other than a [TextView](https://developer.android.com/reference/android/widget/TextView.html) (for example, if you want an [ImageView](https://developer.android.com/reference/android/widget/ImageView.html) for each array item), extend the [ArrayAdapter](https://developer.android.com/reference/android/widget/ArrayAdapter.html) class and override [getView()](https://developer.android.com/reference/android/widget/ArrayAdapter.html" \l "getView(int, android.view.View, android.view.ViewGroup)) to return the type of view you want for each item.

[SimpleCursorAdapter](https://developer.android.com/reference/android/widget/SimpleCursorAdapter.html)

Use this adapter when your data comes from a [Cursor](https://developer.android.com/reference/android/database/Cursor.html). When using [SimpleCursorAdapter](https://developer.android.com/reference/android/widget/SimpleCursorAdapter.html), you must specify a layout to use for each row in the [Cursor](https://developer.android.com/reference/android/database/Cursor.html) and which columns in the [Cursor](https://developer.android.com/reference/android/database/Cursor.html) should be inserted into which views of the layout. For example, if you want to create a list of people's names and phone numbers, you can perform a query that returns a [Cursor](https://developer.android.com/reference/android/database/Cursor.html) containing a row for each person and columns for the names and numbers. You then create a string array specifying which columns from the [Cursor](https://developer.android.com/reference/android/database/Cursor.html) you want in the layout for each result and an integer array specifying the corresponding views that each column should be placed:

String[] fromColumns = {ContactsContract.Data.DISPLAY\_NAME,  
                        ContactsContract.CommonDataKinds.Phone.NUMBER};  
int[] toViews = {R.id.display\_name, R.id.phone\_number};

When you instantiate the [SimpleCursorAdapter](https://developer.android.com/reference/android/widget/SimpleCursorAdapter.html), pass the layout to use for each result, the [Cursor](https://developer.android.com/reference/android/database/Cursor.html) containing the results, and these two arrays:

SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,  
        R.layout.person\_name\_and\_number, cursor, fromColumns, toViews, 0);  
ListView listView = getListView();  
listView.setAdapter(adapter);

The [SimpleCursorAdapter](https://developer.android.com/reference/android/widget/SimpleCursorAdapter.html) then creates a view for each row in the [Cursor](https://developer.android.com/reference/android/database/Cursor.html) using the provided layout by inserting each fromColumns item into the corresponding toViews view.

.

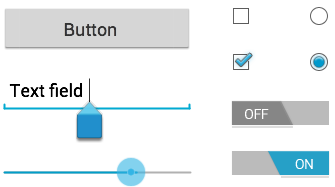
If, during the course of your application's life, you change the underlying data that is read by your adapter, you should call [notifyDataSetChanged()](https://developer.android.com/reference/android/widget/ArrayAdapter.html" \l "notifyDataSetChanged()). This will notify the attached view that the data has been changed and it should refresh itself.

### Handling click events

You can respond to click events on each item in an [AdapterView](https://developer.android.com/reference/android/widget/AdapterView.html) by implementing the [AdapterView.OnItemClickListener](https://developer.android.com/reference/android/widget/AdapterView.OnItemClickListener.html) interface. For example:

// Create a message handling object as an anonymous class.  
private OnItemClickListener mMessageClickedHandler = new OnItemClickListener() {  
    public void onItemClick(AdapterView parent, View v, int position, long id) {  
        // Do something in response to the click  
    }  
};  
  
listView.setOnItemClickListener(mMessageClickedHandler);

# **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.

Adding an input control to your UI is as simple as adding an XML element to your [XML layout](https://developer.android.com/guide/topics/ui/declaring-layout.html). For example, here's a layout with a text field and button:

<?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="horizontal">  
    <EditText android:id="@+id/edit\_message"  
        android:layout\_weight="1"  
        android:layout\_width="0dp"  
        android:layout\_height="wrap\_content"  
        android:hint="@string/edit\_message" />  
    <Button android:id="@+id/button\_send"  
        android:layout\_width="wrap\_content"  
        android:layout\_height="wrap\_content"  
        android:text="@string/button\_send"  
        android:onClick="sendMessage" />  
</LinearLayout>

Each input control supports a specific set of input events so you can handle events such as when the user enters text or touches a button.

## Common Controls

Here's a list of some common controls that you can use in your app. Follow the links to learn more about using each one.

**Note:** Android provides several more controls than are listed here. Browse the [android.widget](https://developer.android.com/reference/android/widget/package-summary.html) package to discover more. If your app requires a specific kind of input control, you can build your own [custom components](https://developer.android.com/guide/topics/ui/custom-components.html).

|  |  |
| --- | --- |
| Control Type | Description |
| [Button](https://developer.android.com/guide/topics/ui/controls/button.html) | A push-button that can be pressed, or clicked, by the user to perform an action. |
| [Text field](https://developer.android.com/training/keyboard-input/index.html) | An editable text field. You can use the AutoCompleteTextView widget to create a text entry widget that provides auto-complete suggestions |
| [Checkbox](https://developer.android.com/guide/topics/ui/controls/checkbox.html) | An on/off switch that can be toggled by the user. You should use checkboxes when presenting users with a group of selectable options that are not mutually exclusive. |
| [Radio button](https://developer.android.com/guide/topics/ui/controls/radiobutton.html) | Similar to checkboxes, except that only one option can be selected in the group. |
| [Toggle button](https://developer.android.com/guide/topics/ui/controls/togglebutton.html) | An on/off button with a light indicator. |
| [Spinner](https://developer.android.com/guide/topics/ui/controls/spinner.html) | A drop-down list that allows users to select one value from a set. |
| [Pickers](https://developer.android.com/guide/topics/ui/controls/pickers.html) | A dialog for users to select a single value for a set by using up/down buttons or via a swipe gesture. Use a DatePickercode> widget to enter the values for the date (month, day, year) or a TimePickerwidget to enter the values for a time (hour, minute, AM/PM), which will be formatted automatically for the user's locale. |

Input Events

On Android, there's more than one way to intercept the events from a user's interaction with your application. When considering events within your user interface, the approach is to capture the events from the specific View object that the user interacts with. The View class provides the means to do so.

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. These methods are called by the Android framework when the respective action occurs on that object. For instance, when a View (such as a Button) is touched, the onTouchEvent() method is called on that object. However, in order to intercept this, you must extend the class and override the method. However, extending every View object in order to handle such an event would not be practical. This is why the View class also contains a collection of nested interfaces with callbacks that you can much more easily define. These interfaces, called [event listeners](https://developer.android.com/guide/topics/ui/ui-events.html#EventListeners), are your ticket to capturing the user interaction with your UI.

While you will more commonly use the event listeners to listen for user interaction, there may come a time when you do want to extend a View class, in order to build a custom component. Perhaps you want to extend the [Button](https://developer.android.com/reference/android/widget/Button.html) class to make something more fancy. In this case, you'll be able to define the default event behaviors for your class using the class [event handlers](https://developer.android.com/guide/topics/ui/ui-events.html#EventHandlers).

Event Listeners

An event listener is an interface in the [View](https://developer.android.com/reference/android/view/View.html) 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.

Included in the event listener interfaces are the following callback methods:

onClick()

From [View.OnClickListener](https://developer.android.com/reference/android/view/View.OnClickListener.html). 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](https://developer.android.com/reference/android/view/View.OnLongClickListener.html). 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).

onFocusChange()

From [View.OnFocusChangeListener](https://developer.android.com/reference/android/view/View.OnFocusChangeListener.html). This is called when the user navigates onto or away from the item, using the navigation-keys or trackball.

onKey()

From [View.OnKeyListener](https://developer.android.com/reference/android/view/View.OnKeyListener.html). This is called when the user is focused on the item and presses or releases a hardware key on the device.

onTouch()

From [View.OnTouchListener](https://developer.android.com/reference/android/view/View.OnTouchListener.html). 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](https://developer.android.com/reference/android/view/View.OnCreateContextMenuListener.html). 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](https://developer.android.com/guide/topics/ui/menus.html#context-menu) developer guide.

These methods are the sole inhabitants of their respective interface. To define one of these methods and handle your events, implement the nested interface in your Activity or define it as an anonymous class. Then, pass an instance of your implementation to the respective View.set...Listener()method. (E.g., call [setOnClickListener()](https://developer.android.com/reference/android/view/View.html" \l "setOnClickListener(android.view.View.OnClickListener)) and pass it your implementation of the [OnClickListener](https://developer.android.com/reference/android/view/View.OnClickListener.html).)

The example below shows how to register an on-click listener for a Button.

// 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);  
    ...  
}

You may also find it more convenient to implement OnClickListener as a part of your Activity. This will avoid the extra class load and object allocation. For example:

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  
    }  
    ...  
}

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. For the few that do, here's why:

* [onLongClick()](https://developer.android.com/reference/android/view/View.OnLongClickListener.html#onLongClick(android.view.View)) - 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.
* [onKey()](https://developer.android.com/reference/android/view/View.OnKeyListener.html#onKey(android.view.View, int, android.view.KeyEvent)) - 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-key listeners.
* [onTouch()](https://developer.android.com/reference/android/view/View.OnTouchListener.html#onTouch(android.view.View, android.view.MotionEvent)) - 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.

Remember that hardware key events are always delivered to the View currently in focus. They are dispatched starting from the top of the View hierarchy, and then down, until they reach the appropriate destination. If your View (or a child of your View) currently has focus, then you can see the event travel through the [dispatchKeyEvent()](https://developer.android.com/reference/android/view/View.html" \l "dispatchKeyEvent(android.view.KeyEvent)) method. As an alternative to capturing key events through your View, you can also receive all of the events inside your Activity with [onKeyDown()](https://developer.android.com/reference/android/app/Activity.html" \l "onKeyDown(int, android.view.KeyEvent)) and [onKeyUp()](https://developer.android.com/reference/android/app/Activity.html" \l "onKeyUp(int, android.view.KeyEvent)).

Also, when thinking about text input for your application, remember that many devices only have software input methods. Such methods are not required to be key-based; some may use voice input, handwriting, and so on. Even if an input method presents a keyboard-like interface, it will generally **not** trigger the [onKeyDown()](https://developer.android.com/reference/android/app/Activity.html" \l "onKeyDown(int, android.view.KeyEvent)) family of events. You should never build a UI that requires specific key presses to be controlled unless you want to limit your application to devices with a hardware keyboard. In particular, do not rely on these methods to validate input when the user presses the return key; instead, use actions like [IME\_ACTION\_DONE](https://developer.android.com/reference/android/view/inputmethod/EditorInfo.html#IME_ACTION_DONE) to signal the input method how your application expects to react, so it may change its UI in a meaningful way. Avoid assumptions about how a software input method should work and just trust it to supply already formatted text to your application.

**Note:** Android will call event handlers first and then the appropriate default handlers from the class definition second. As such, returning true from these event listeners will stop the propagation of the event to other event listeners and will also block the callback to the default event handler in the View. So be certain that you want to terminate the event when you return true.

## Event Handlers

If you're building a custom component from View, then you'll be able to define several callback methods used as default event handlers. In the document about [Custom Components](https://developer.android.com/guide/topics/ui/custom-components.html), you'll learn see some of the common callbacks used for event handling, including:

* [onKeyDown(int, KeyEvent)](https://developer.android.com/reference/android/view/View.html#onKeyDown(int, android.view.KeyEvent)) - Called when a new key event occurs.
* [onKeyUp(int, KeyEvent)](https://developer.android.com/reference/android/view/View.html#onKeyUp(int, android.view.KeyEvent)) - Called when a key up event occurs.
* [onTrackballEvent(MotionEvent)](https://developer.android.com/reference/android/view/View.html#onTrackballEvent(android.view.MotionEvent)) - Called when a trackball motion event occurs.
* [onTouchEvent(MotionEvent)](https://developer.android.com/reference/android/view/View.html#onTouchEvent(android.view.MotionEvent)) - Called when a touch screen motion event occurs.
* [onFocusChanged(boolean, int, Rect)](https://developer.android.com/reference/android/view/View.html#onFocusChanged(boolean, int, android.graphics.Rect)) - Called when the view gains or loses focus.

There are some other methods that you should be aware of, which are not part of the View class, but can directly impact the way you're able to handle events. So, when managing more complex events inside a layout, consider these other methods:

* [Activity.dispatchTouchEvent(MotionEvent)](https://developer.android.com/reference/android/app/Activity.html#dispatchTouchEvent(android.view.MotionEvent)) - This allows your [Activity](https://developer.android.com/reference/android/app/Activity.html) to intercept all touch events before they are dispatched to the window.
* [ViewGroup.onInterceptTouchEvent(MotionEvent)](https://developer.android.com/reference/android/view/ViewGroup.html#onInterceptTouchEvent(android.view.MotionEvent)) - This allows a [ViewGroup](https://developer.android.com/reference/android/view/ViewGroup.html) to watch events as they are dispatched to child Views.
* [ViewParent.requestDisallowInterceptTouchEvent(boolean)](https://developer.android.com/reference/android/view/ViewParent.html#requestDisallowInterceptTouchEvent(boolean)) - Call this upon a parent View to indicate that it should not intercept touch events with [onInterceptTouchEvent(MotionEvent)](https://developer.android.com/reference/android/view/ViewGroup.html" \l "onInterceptTouchEvent(android.view.MotionEvent)).

## Touch Mode

When a user is navigating a user interface with directional keys or a trackball, it is necessary to give focus to actionable items (like buttons) so the user can see what will accept input. If the device has touch capabilities, however, and the user begins interacting with the interface by touching it, then it is no longer necessary to highlight items, or give focus to a particular View. Thus, there is a mode for interaction named "touch mode."

For a touch-capable device, once the user touches the screen, the device will enter touch mode. From this point onward, only Views for which[isFocusableInTouchMode()](https://developer.android.com/reference/android/view/View.html#isFocusableInTouchMode()) is true will be focusable, such as text editing widgets. Other Views that are touchable, like buttons, will not take focus when touched; they will simply fire their on-click listeners when pressed.

Any time a user hits a directional key or scrolls with a trackball, the device will exit touch mode, and find a view to take focus. Now, the user may resume interacting with the user interface without touching the screen.

The touch mode state is maintained throughout the entire system (all windows and activities). To query the current state, you can call [isInTouchMode()](https://developer.android.com/reference/android/view/View.html" \l "isInTouchMode())to see whether the device is currently in touch mode.

## Handling Focus

The framework will handle routine focus movement in response to user input. This includes changing the focus as Views are removed or hidden, or as new Views become available. Views indicate their willingness to take focus through the [isFocusable()](https://developer.android.com/reference/android/view/View.html" \l "isFocusable()) method. To change whether a View can take focus, call [setFocusable()](https://developer.android.com/reference/android/view/View.html" \l "setFocusable(boolean)). When in touch mode, you may query whether a View allows focus with [isFocusableInTouchMode()](https://developer.android.com/reference/android/view/View.html" \l "isFocusableInTouchMode()). You can change this with [setFocusableInTouchMode()](https://developer.android.com/reference/android/view/View.html" \l "setFocusableInTouchMode(boolean)).

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 of these attributes to the View from which the focus is leaving. Define the value of the attribute to be the id of the View to which focus should be given. 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 this vertical layout, navigating up from the first Button would not go anywhere, nor would navigating down from the second Button. Now that the top Button has defined the bottom one as the **nextFocusUp** (and vice versa), 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 your layout declaration. Set the value **true**. You can also declare a View as focusable while in Touch Mode with android:focusableInTouchMode.

To request a particular View to take focus, call [requestFocus()](https://developer.android.com/reference/android/view/View.html" \l "requestFocus()).

To listen for focus events (be notified when a View receives or looses focus), use [onFocusChange()](https://developer.android.com/reference/android/view/View.OnFocusChangeListener.html" \l "onFocusChange(android.view.View, boolean)), as discussed in the [Event Listeners](https://developer.android.com/guide/topics/ui/ui-events.html#EventListeners) section, above.