|  |  |  |
| --- | --- | --- |
| [Previous](http://eagle.phys.utk.edu/guidry/android/applicationFundamentals.html" \t "_self) | | [Next](http://eagle.phys.utk.edu/guidry/android/applicationLifecycles.html) | | [Home](http://eagle.phys.utk.edu/guidry/android/index.html) |

**http://eagle.phys.utk.edu/guidry/android/logo.png Android User Interfaces**

Since a user will typically interact visually with an Android application, it is important to understand how Android lays out user interfaces. In this section we give a brief overview. See the Android [User Interface](http://developer.android.com/guide/topics/ui/index.html) document for more detail.

**A Hierarchy of Views**

Visually, an Android window is a hierarchy of *views*, which are objects derived from the [*View*](http://developer.android.com/reference/android/view/View.html) and[*ViewGroup*](http://developer.android.com/reference/android/view/ViewGroup.html) classes.

1. *View* is the base class for *[android.widget](http://developer.android.com/reference/android/widget/package-summary.html" \t "_new)* subclasses, which instantiate fully-implemented UI objects. For example, *Button* is a subclass of *View*.
2. *ViewGroup* is the base class for "layout" subclasses, which implement different kinds of UI layout architecture. (A *ViewGroup* is a special form of *View* that can contain other *Views*.) For example,
   * The class *[LinearLayout](http://developer.android.com/reference/android/widget/LinearLayout.html" \t "_new)*, which implements a layout that arranges its children in a single column or a single row.
   * The class *[TableLayout](http://developer.android.com/reference/android/widget/TableLayout.html" \t "_new)*, which arranges its children in rows and columns like a table.
   * The class *[RelativeLayout](http://developer.android.com/reference/android/widget/RelativeLayout.html" \t "_new)*, where the positions of the children can be described relative to each other or to the parent.
   * The class *[FrameLayout](http://developer.android.com/reference/android/widget/FrameLayout.html" \t "_new)*, which blocks out an area on the screen to display a single item. Multiple children may be added to a FrameLayout and all children will be pegged to the top left of the screen (this, for example, is a way to overlay a set of buttons over an image).
   * The class *[GridView](http://developer.android.com/reference/android/widget/GridView.html" \t "_new)*, which displays items in a two-dimensional scrolling grid.

All of these layout classes extend *ViewGroup* or a subclass of *ViewGroup*.

Each view in the hierarchy controls a particular rectangular subspace within the window.

* *Parent views* contain and organize the layout of their children.
* *Leaf views* are those at the bottom of the hierarchy; they draw in the rectangles they control and respond to user actions within those rectangles.

The following figure illustrates a view hierarchy

|  |
| --- |
| http://eagle.phys.utk.edu/guidry/android/figs/viewgroup.png |

**Placing the View Hierarchy in an Android Window**

The *[setContentView(View view)](http://developer.android.com/reference/android/app/Activity.html" \l "setContentView(android.view.View)" \t "_new)* method of *Activity* is used to place a view hierarchy within an activity window, where the content *view* is the *View* object at the root of the view hierarchy (the topmost *ViewGroup* in the preceding figure). Android handles the drawing of the view hierarchy but the *Activity* must supply the root node as the starting point. The essential procedure is that the root node of the hierarchy requests that its child nodes draw themselves, and in turn each view group node requests that its children draw themselves. Children may request a size and location within the parent, but the parent object has the final say-so on size and position for the child objects. This will be adequate introduction for our initial purposes; a more extensive discussion of rendering the view hierarchy within an Android window may be found in [How Android Draws Views](http://developer.android.com/guide/topics/ui/how-android-draws.html).

**XML Layouts**

The recommended way to lay out the view hierarchy is with an XML layout file. The name of an XML element in this file is in one-to-one correspondence with a Java class that it represents: a <TextView> element creates a *TextView* in your UI, a <LinearLayout> element creates a *LinearLayout* view group, and so on. Here is an example of a simple XML layout file:

<?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="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>

This layout would create a *TextView* and a *Button*, arrayed vertically in a linear layout in the Android window (we will explain the details of such a listing later). As long as you follow the rules, you do not have to worry about managing the relationship between XML layout of the UI and the Java code that will reference it. Basically you must just place correctly written code in a standardized set of directories and Android will do the rest. When you load a layout resource, Android automatically initializes these run-time objects, corresponding to the elements in your layout.

**Menus**

Menus are another important part of the UI but, as we describe below, layout and event handling are dealt with somewhat differently for menus than for other UI widgets in Android and the nature of the Menu user interface has evolved between early and current versions of Android.

***Evolution of the Menu***

Before Android 3.0 (API level 11) Android devices commonly had a dedicated Menu button. Following API level 11, Android devices were no longer required to have a Menu button and app developers were encouraged to move the functionality of the Menu button into the [Action Bar](http://developer.android.com/guide/topics/ui/actionbar.html). Even more recently, the action bar has evolved into the [Toolbar](https://developer.android.com/reference/android/support/v7/widget/Toolbar.html), which is a more flexible version of the Action Bar.

Even though the best-practices user display of menu items has changed following Android 3.0, the *functionality* of the Menu APIs remains largely the same. Since many modern Android devices do not have an explicit Menu button, apps designed for them will still use much of the Menu API functionality, but your app should not assume the presence of a Menu button to initiate that functionality (use an Action Bar instead). Furthermore, because of Google the backward-compatibility [Support Library](http://developer.android.com/tools/support-library/index.html?utm_content=buffer11f41&utm_source=buffer&utm_medium=twitter&utm_campaign=Buffer), it is possible to implement an Action Bar even on older versions of Android. This complicates the discussion even further if one is supporting both new and older versions of Android. We shall assume programming consistent with devices running Android 4.0 or later and not deal much with compatibility issues for older versions here.

|  |
| --- |
| By compatibility in preceding statements, we mostly mean compatibility of newer options in the API with devices running earlier versions of the API that did not implement those options. Android is for the most part forward and backward compatible without doing anything special. Programs written in Android 1.6 adhering to Android best practices will commonly work either as-is, or with a few small tweaks, under Android 4.4, and programs written under Android 4.4 will often work fine on devices running earlier versions of Android, unless those programs use explicitly features of the API that were introduced after the earlier versions. Even in this case, by using the [Support Library](http://developer.android.com/tools/support-library/index.html?utm_content=buffer11f41&utm_source=buffer&utm_medium=twitter&utm_campaign=Buffer), it is often possible to deploy features introduced in later versions of the API on devices running older versions. The [Action Bar](http://developer.android.com/guide/topics/ui/actionbar.html) one such feature; [Fragments](http://developer.android.com/reference/android/support/v4/app/Fragment.html) are another. |

***Types of Menus***

There are three general types of menus in Android:

1. *Options menu and action bar:* An *Activity* can be programmed so that if the MENU key on the device is pressed while the *Activity* is being displayed a menu appears at the bottom of the screen. Such menus are often used for preferences, settings, and other options for an entire activity or application. As we have noted, modern Android apps will typically implement similar functionality through the Action Bar and do not assume the presence of a Menu button. See [Creating an Options Menu](http://developer.android.com/guide/topics/ui/menus.html#options-menu) for further information.
2. *Context menu and contextual action mode:* Displayed when the user presses and holds down on (long presses) an item. Such menus are often useful for supplying further information or options for a specific item. For further information, see [Creating Contextual Menus](http://developer.android.com/guide/topics/ui/menus.html#context-menu)
3. *Popup menu:* displays a modal menu anchored to the view from which the menu was invoked. It appears below the anchor view if there is sufficient room, or above the view otherwise. It is commonly used to provide an overflow of actions or to provide options for a second part of a command. Additional information may be found in [Creating a Popup Menu](http://developer.android.com/guide/topics/ui/menus.html#PopupMenu)

In the following we shall give a brief overview of the Menu prior to Android 3.0 *(oldschool)* and the modern Action Bar *(newschool)*. For a more extensive discussion, consult the Android [Menus](http://developer.android.com/guide/topics/ui/menus.html" \t "_new)document.

***Menus: Oldschool***

The left figure below illustrates an options menu produced when the MENU key is pressed for an older Android phone on its default screen and the right figure below shows a context menu produced on the same phone after a long press on an entry in the Contacts list.

|  |  |
| --- | --- |
| http://eagle.phys.utk.edu/guidry/android/figs/mainMenu.png | http://eagle.phys.utk.edu/guidry/android/figs/contextMenu.png |

* The *View* hierarchy corresponding to menus is not handled directly by the user. Instead the user defines the[*onCreateOptionsMenu()*](http://developer.android.com/reference/android/app/Activity.html#onCreateOptionsMenu(android.view.Menu)) or *[onCreateContextMenu()](http://developer.android.com/reference/android/app/Activity.html" \l "onCreateContextMenu(android.view.ContextMenu,%20android.view.View,%20android.view.ContextMenu.ContextMenuInfo)" \t "_new)* callback methods for an Activity and specifies in them the items to be included in the menu (the menu items can also be declared in an XML resource file). When the menu is needed, Android will automatically create the *View* hierarchy for the menu and draw the user-specified menu items in it.
* Android menus also handle their own events so (unlike for other widgets, as described below) the user does not need to register event listeners on the menu items. The *[onOptionsItemSelected()](http://developer.android.com/reference/android/app/Activity.html" \l "onOptionsItemSelected(android.view.MenuItem)" \t "_new)* or *[onContextItemSelected()](http://developer.android.com/reference/android/app/Activity.html" \l "onContextItemSelected(android.view.MenuItem)" \t "_new)* methods will be called automatically by the Android framework when a menu item is selected, so the user puts logic in those methods to deal with the menu events.

|  |
| --- |
| The Android [Menus](http://developer.android.com/guide/topics/ui/menus.html) document contains further information about creating and using various kinds of menus in Android. |

***Menus: Newschool***

In Android 3.0 and beyond it is preferred that the options menu be handled in the [ActionBar](http://developer.android.com/reference/android/app/ActionBar.html" \t "_new) or [Toolbar](https://developer.android.com/reference/android/support/v7/widget/Toolbar.html" \t "_new)classes.  An action bar is a dedicated piece of real estate at the top of a screen that generally persists through the screens of the app. It identifies the app and the user location in the app, and provides user actions and navigation modes. It provides a standardized interface across applications that is adapted automatically by the system for different screen environments. Generically, an Action Bar is organized as in the following figure,

|  |
| --- |
| http://eagle.phys.utk.edu/guidry/android/figs/action_bar_basics.png |

and has the basic pieces labeled by the numbers in the above figure:

1. An icon that establishes the identity of the app.
2. A view control that allows users to switch between views (optional).
3. Action buttons that enable the most important actions of your app. Actions that don't fit on the action bar are moved automatically to the Action overflow described below.
4. Action overflow, which generates a popup upon tapping that contains less often used actions.

The figure below illustrates an actual action bar, showing (1) the app icon and label, (2) two action items, and (3) the action overflow button (the three vertical dots on the right) that pops up a menu with additional choices.

|  |
| --- |
| http://eagle.phys.utk.edu/guidry/android/figs/actionbarAt2x.png |

The general features of action bars are discussed in [Action Bar](http://developer.android.com/guide/topics/ui/actionbar.html) and design issues for action bars are elaborated in this [document](http://developer.android.com/design/patterns/actionbar.html).

**Adapters and Data Binding**

In dynamical applications you will often want to populate view groups with non-static data from some external data source. For example, a display of the current temperature in several different cities must display temperature data supplied continuously from an external source. This is termed *data binding* to the View. In Android this is handled by using an *[AdapterView](http://developer.android.com/reference/android/widget/AdapterView.html" \t "_new)* as the *ViewGroup*, with each child *View* populated by an [*Adapter*](http://developer.android.com/reference/android/widget/Adapter.html). An *AdapterView* object is a *ViewGroup* that determines its child views based on a given *Adapter* object. The *Adapter* acts as a bridge between the data source and the *AdapterView*, which displays it.

|  |
| --- |
| The binding of data to views is described more extensively in this AdapterView [document](http://developer.android.com/guide/topics/ui/binding.html). |

**Styles and Themes**

Good design is an important component of a user-friendly and aesthetically appealing user interface. One important aspect of good design is achieving a uniform look and feel for applications, which has positive implications for both aesthetics and functionality. One way to achieve this is through *styles* and *themes*:

1. *Style:* a collection of properties that specify the look and format for a *View* or window. In Android a style is defined in an XML resource separate from the XML layout resource.
2. *Theme:* a style applied to an entire Activity or application, rather than an individual View.

Styles and themes allow design to be separated from content (thus they function in a way similar to Cascading Style Sheets in webpage design), which facilitates maintenance and extension of projects.

Android provides some default style and theme resources out of the box, and you can also declare your own custom styles and themes. The following figures taken from the [Themes, Styles, & Preferences](http://eagle.phys.utk.edu/guidry/android/themesDemo.html)project illustrate the use of Android styles and themes.

|  |  |
| --- | --- |
| http://eagle.phys.utk.edu/guidry/android/figs/themesScreen1Black.png | http://eagle.phys.utk.edu/guidry/android/figs/themesScreen1Holo.png |

In this example styles and themes have been used to separate the content from the design: the left and right figures have exactly the same content, but they have different looks, and the programmer does not have to write any additional code to accomplish this, once the theme has been defined.

|  |
| --- |
| A more extensive discussion of using existing styles and themes, and constructing your own, may be found in the Android [Styles and Themes](http://developer.android.com/guide/topics/ui/themes.html) document and in the [Themes, Styles, & Preferences](http://eagle.phys.utk.edu/guidry/android/themesDemo.html)project. |

**Handling Events**

The layout of UI widgets described above will not be very useful unless we define event handlers to respond to events associated with the widgets. We shall give various specific examples later but remark here that this can be implemented in one of two basic ways.

1. The most common way that we will implement event handlers for widgets in our layout is to use Java to define an event handler and then register it with the appropriate view. The *View* class has a set of interfaces named *On<something>Listener*, each with a callback method called *on<something>()*. Examples include
   * *[View.OnClickListener](http://developer.android.com/reference/android/view/View.OnClickListener.html" \t "_new)*, which handles "clicks" on a *View*.
   * [*View.OnTouchListener*](http://developer.android.com/reference/android/view/View.OnTouchListener.html), which handles touch screen events in a *View*.
   * [*View.OnKeyListener*](http://developer.android.com/reference/android/view/View.OnKeyListener.html), which handles key presses within a *View*

For example, to cause your *View* to be notified when it is "clicked", (1) implement *OnClickListener*, (2) define its *[onClick(View v)](http://developer.android.com/reference/android/view/View.OnClickListener.html" \l "onClick(android.view.View)" \t "_new)* callback method implementing actions to be performed when an event is detected, and (3) register it to the *View* with *[setOnClickListener()](http://developer.android.com/reference/android/view/View.html" \l "setOnClickListener(android.view.View.OnClickListener)" \t "_new)*. (We shall see later how to determine which widget within the *View* was clicked when there is more than one.)

1. If none of the widgets or layouts already defined in Android are adequate to the task, you may wish to create your own View subclass (see [Custom Components](http://developer.android.com/guide/topics/ui/custom-components.html)). Then you will typically handle events by overriding an existing callback method for the *View*.

|  |
| --- |
| A more extensive discussion of handling UI events within *Views* may be found in [Input Events](http://developer.android.com/guide/topics/ui/ui-events.html). |

*Last modified: July 25, 2016*

|  |  |  |
| --- | --- | --- |
| [Previous](http://eagle.phys.utk.edu/guidry/android/applicationFundamentals.html) | | [Next](http://eagle.phys.utk.edu/guidry/android/applicationLifecycles.html) | | [Home](http://eagle.phys.utk.edu/guidry/android/index.html) |

http://eagle.phys.utk.edu/guidry/android/androidUserInterface.html