Android Architecture

android architecture or **Android software stack** is categorized into five parts:

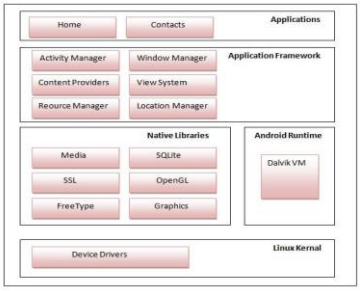
1) Linux kernel - It is the heart of android architecture that exists at the root of android architecture. **Linux kernel** is responsible for device drivers, power management, memory management, device management and resource access.

2) Native Libraries - On the top of linux kernel, their are **Native libraries** such as WebKit, OpenGL, FreeType, SQLite, Media, C runtime library (libc) etc.The WebKit library is responsible for browser support, SQLite is for database, FreeType for font support, Media for playing and recording audio and video formats.

3) Android Runtime - In android runtime, there are core libraries and DVM (Dalvik Virtual Machine) which is responsible to run android application. DVM is like JVM but it is optimized for mobile devices. It consumes less memory and provides fast performance.

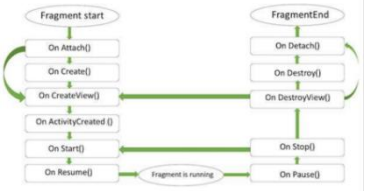
4) Android Framework - On the top of Native libraries and android runtime, there is android framework. Android framework includes **Android API's** such as UI (User Interface), telephony, resources, locations, Content Providers (data) and package managers. It provides a lot of classes and interfaces for android application development.

5) Applications - On the top of android framework, there are applications. All applications such as home, contact, settings, games, browsers are using android framework that uses android runtime and libraries. Android runtime and native libraries are using linux kernal.



Fragment Lifecycle

A fragment is very dependent on the activity in which it lives and can go through multiple steps while its activity goes through one. Instance Description onAttach() .The fragment instance is associated with an activity instance. The fragment and the activity are not fully initialized. In this method a reference to the fragment for further initialization works. onCreate() The system calls this method when creating the fragment. The onCreate() method is called after the onCretae() method on the activity but before the onCreateView() method of the fragment. We should initialize essential component of the fragments that we want to retain when the fragment is paused or stopped then resumed. onCreateView() This method creates user interface. In this method we should not interactive with the activity. To draw a UI for our fragment we must turn a view from this method. If we return null, the fragment does not provide a UI. onActivityCreated() This is the point where host activity gets created. This is important to notice that this method is called after onCreateView() to indicate that the activity's onCreate() has completed. onStart() This is called once the fragment gets visible. onResume() From this point onwards fragment becomes active. onPause() The system calls this method as the first indication that the user is leaving the fragment. The fragment is visible but becomes not active any more. If any other activity is animated on top of the activity which contains the fragment. onStop() The fragment becomes not visible and going to stopped onDestroyView() It destroys the view of the fragment. This is the counter part of onCreateView() where we set up the UI. If the fragment is recreated from the back stack then onCreateView() is called. onDestroy() This method does the final cleanup related to the fragment state. But is not guaranteed to get called by Android Platform onDitach() It is called after onDistroy() to notify that the fragment has been disassociated from its hosting activity



FRAGMENT LIFECYCLE

ADAPTER VIEWS

Now that we've been introduced to adapters, it is time to put them to work for us, providing data for list controls. In this section, we're going to first cover the basic list control, the ListView. Then, we'll describehow to create our own custom adapter, and finally, we'll describe the other types of list controls: GridViews, spinners, and the gallery.

LIST VIEW

The ListView control displays a list of items vertically. That is, if we've got a list of items to view and the number of items extends beyond what we can currently see in the display, we can scroll to see the rest of the items. We generally use a ListView by writing a new activity that extends android.app.ListActivity. ListActivity contains a ListView, and we set the data for the ListView by calling the setListAdapter() method. As we described previously, adapters link list controls to the data and help prepare the child views for the list control. Items in a ListView can be clicked to take immediate action or selected to act on the set of selected.

Android supports 4 types dialog boxes:

AlertDialog : An alert dialog box supports 0 to 3 buttons and a list of selectable elements, including check boxes and radio buttons. Among the other dialog boxes, the most suggested dialog box in the alert dialog box. **ProgressDialog** : This dialog box displays a progress wheel or progress bar. It is an extension of AlertDialog and supports adding buttons. **DatePickerDialog** : This dialog box is used for selecting a date by the user. **TimePickerDialog** : This dialog box is used for selecting a time by the user.

USING DIALOGS IN ANDROID

The Android SDK offers extensive support for dialogs. A dialog is a smaller window that pops up in front of the current window to show an urgent message, to prompt the user for a piece of input, or to show some sort of status like the progress of a download. The user is generally expected to interact with the dialog and then return to the window underneath to continue with the application. Android allows a dialog fragment to also be embedded within an activity's layout. Dialogs that are explicitly supported in Android include the alert, prompt, pick-list, single-choice, multiple-choice,progress, time-picker, and date-picker dialogs. Dialogs in Android are asynchronous, which provides flexibility. However, if we are accustomed to a programming framework where dialogs are primarily synchronous (such as Microsoft Windows, or JavaScript dialogs in web pages), we might find asynchronous dialogs a bit unintuitive. With a synchronous dialog, the line of code after the dialog is shown does not run until the dialog has been dismissed. This means the next line of code could interrogate which button was pressed, or what text was typed into the dialog. In Android however, dialogs are asynchronous. As soon as the dialog has been shown, the next line of code runs, even though the user hasn't touched the dialog yet. Our application has deal with this fact by implementing callbacks from the dialog, to allow the application to be notified of user interaction with the dialog. This also means our application has the ability to dismiss the dialog from code, which is powerful. If the dialog is displaying a busy message because our application is doing something, as soon as our application has completed that task, it can dismiss the dialog from code.

DIALOG FRAGMENTS

The use of dialog fragments is to present a simple alert dialog and a custom dialog that is used to collect prompt text. Dialog-related functionality uses a class called DialogFragment. A DialogFragment is derived from the class Fragment and behaves much like a fragment. We will then use the DialogFragment as the base class for our dialogs. Once we have a derived dialog from this class such as public class MyDialogFragment extends DialogFragment { ... } we can then show this dialog fragment MyDialogFragment as a dialog using a fragment transaction. Following example shows a code snippet to do this. Listing 9–1. Showing a Dialog Fragment

```
SomeActivity
{ //....other activity functions
public void showDialog()
{
//construct MyDialogFragment
MyDialogFragment mdf =
MyDialogFragment.newInstance(arg1,arg2);
FragmentManager fm = getFragmentManager();
FragmentTransaction ft = fm.beginTransaction();
mdf.show(ft,"my-dialog-tag");
}
//....other activity functions
}
```

The steps to show a dialog fragment are as follows:

1. Create a dialog fragment.
2. Get a fragment transaction.
3. Show the dialog using the fragment transaction from step 2. **FUNDAMENTAL COMPONENTS**

When we build applications for Android we need to understand JavaServer Pages (JSP) and servlets in order to write Java 2 Platform, Enterprise Edition (J2EE) applications. Similarly, we need to understand views, activities, fragments, intents, content providers, services, and the AndroidManifest.xml file. **1. View** Views are user interface (UI) elements that form the basic building blocks of a user interface. A view can be a button, a label, a text field, or many other UI elements. Views are also used as containers for views, which mean there's usually a hierarchy of views in the UI. **2. Activity** An activity is a UI concept that usually represents a single screen in our application. It generally contains one or more views, but it doesn't have to. An activity is pretty much like it sounds—something that helps the user do one thing, which could be viewing data, creating data, or editing data.Most Android applications have several activities within them. **3. Fragment** When a screen is large, it becomes difficult to manage all of its functionality in a single activity. Fragments are like sub-activities, and an activity can display one or more fragments on the screen at the same time. When a screen is small, an activity is more likely to contain just one fragment, and that fragment can be the same one used within larger screens**. 4. Intent** An intent generically defines an "intention" to do some work. Intents encapsulate several concepts, so the best approach to understanding them is to see examples of their use. We can use intents to perform the following tasks: Broadcast a message., Start a service., Launch an activity., Display a web page or a list of contacts., Dial a phone number or answer a phone call. Intents are not always initiated by our application—they're also used by the system to notify our application of specific events (such as the arrival of a text message). Intents can be explicit or implicit. If we simply say that we want to display a URL, the system decides what component will fulfill the intention. We can also provide specific information about what should handle the intention. **5. Content Provider** Data sharing among mobile applications on a device is common. Therefore, Android defines a standard mechanism for applications to share data (such as a list of contacts) without exposing the underlying storage, structure, and implementation. Through content providers, we can expose our data and have our applications use data from other applications. **6. Service** Services in Android resemble services we see in Windows or other platforms—they're background processes that can potentially run for a long time. Android defines two types of services: local services and remote services. Local services are components that are only accessible by the application that is hosting the service. Conversely, remote services are services that are meant to be accessed remotely by other applications running on the device**. 7. AndroidManifest.xml** AndroidManifest.xml, which is similar to the web.xml file in the J2EE world, defines the contents and behavior of our application. For example, it lists our application's activities and services, along with the permissions and features the application needs to run.

UNDERSTANDING LAYOUT MANAGERS

Android offers a collection of view classes that act as containers for views. These container classes are called layouts (or layout managers), and each implements a specific strategy to manage the size and position of its children. For example, the LinearLayout class lays out its children either horizontally or vertically, one after the other. All layout managers derive from the View class; therefore we can nest layout managers inside of one another. **ANDROID LAYOUT MANAGERS**

Layout Manager Description

LinearLayout Organizes its children either horizontally or vertically **TableLayout** Organizes its children in tabular form **RelativeLayout** Organizes its children relative to one another or to the parent **FrameLayout** Allows us to dynamically change the control(s) in the layout **GridLayout** Organizes its children in a grid arrangement

LINEAR LAYOUT MANAGER

The LinearLayout layout manager is the most basic. This layout manager organizes its children either horizontally or vertically based on the value of the orientation property.We can create a vertically oriented LinearLayout by setting the value of orientation to vertical. Because layout managers can be nested, we could, for example, construct a vertical layout manager that contained horizontal layout managers to create a fill-in form, where each row had a label next to an EditText control. Each row would be its own horizontal layout, but the rows as a collection would be organized vertically.Layout managers extend android.widget.ViewGroup, as do many control-based container classes such as ListView. Although the layout managers and control-based containers extend the same class, the layout manager classes strictly deal with the sizing and position of controls and not user interaction with child controls. For example, compare LinearLayout to the ListView control. On the screen, they look similar in that both can organize children vertically. But the ListView control provides APIs for the user to make selections, whereas LinearLayout does not In other words, the control-based container (ListView) supports user interaction with the items in the container, whereas the layout manager (LinearLayout) addresses sizing and positioning only. Using the LinearLayout layout manager **TABLE LAYOUT MANAGER**

The TableLayout layout manager is an extension of LinearLayout. This layout manager structures its child controls into rows and columns. To use this layout manager, we create an instance of TableLayout and place TableRow elements within it. These TableRow elements contain the controls of the table. Because the contents of a TableLayout are defined by rows as opposed to columns, Android determines the number of columns in the table by finding the row with the most cells. Android creates a table with two rows and three columns. The last column of the first row is an empty cell. **RELATIVE LAYOUT MANAGER**

Another interesting layout manager is RelativeLayout. As the name suggests, this layout manager implements a policy where the controls in the container are laid out relative to either the container or another control in the container. As shown, the UI looks like a simple login form. The username label is pinned to the top of the container, because we set android:layout_alignParent Top to true. Similarly, the Username input field is positioned below the Username label because we set android:layout_below. The Password label appears below the Username label, and the Password input field appears below the Password label. The disclaimer label is pinned to the bottom of the container because we set android:layout_alignParentBottom to true. Besides these three layout attributes, we can also specify layout_above, layout_toRightOf, layout_toLeftOf , layout_centerInParent, and several more. Working with RelativeLayout is fun due to its simplicity. In fact, once we start using it, it'll become our favorite layout manager we'll find our self going back to it over and over again.

FRAME LAYOUT MANAGER

The layout managers that we've discussed so far implement various layout strategies. In other words, each one has a specific way that it positions and orients its children on the screen. With these layout managers, we can have many controls on the screen at one time, each taking up a portion of the screen. Android also offers a layout manager that is mainly used to display a single item: FrameLayout. We mainly use this utility layout class to dynamically display a single view, but we can populate it with many items, setting one to visible while the others are invisible. As we said earlier, we generally use FrameLayout when we need to dynamically set the content of a view to a single control. Although this is the general practice, the control will accept many children, as we demonstrated. Adds two controls to the layout but has one of the controls visible at a time. FrameLayout, however, does not force us to have only one control visible at a time. If we add many controls to the layout, FrameLayout will simply stack the controls, one on top of the other, with the last one on top. This can create an interesting UI. Another interesting aspect of the FrameLayout is that if we add more than one control to the layout, the size of the layout is computed as the size of the largest item in the container, the top image is actually much smaller than the image behind it, but because the size of the layout is computed based on the largest control, the image on top is stretched. Also note that if we put many controls inside a FrameLayout with one or more of them invisible to start, we might want to consider using setMeasureAllChildren(true) on our frameLayout.

Because the largest child dictates the layout size, you'll have a problem if the largest child is invisible to begin with: when it becomes visible, it is only partially visible. To ensure that all items are rendered properly, call setMeasureAllChildren() and pass it a value of true. The equivalent XML attribute for FrameLayout is **android:measureAllChildren="true".**

GRID LAYOUT MANAGER

Android 4.0 brought with it a new layout manager called GridLayout. As we might expect, it lays out views in a grid pattern of rows and columns, somewhat like TableLayout. However, it's easier to use than TableLayout. With a GridLayout, we can specify a row and column value for a view, and that's where it goes in the grid. This means we don't need to specify a view for every cell, just those that we want to hold a view. Views can span multiple grid cells. We can even put more than one view into the same grid cell. When laying out views, we must not use the weight attribute, because it does not work in child views of a GridLayout. We can use the layout_gravity attribute instead. Other interesting attributes we can use with GridLayout child views include layout_column and layout_columnSpan to specify the left-most column and the number of columns the view takes up, respectively. Similarly, there are layout_row and layout_rowSpan attributes. Interestingly, we do not need to specify layout_height and layout_width for GridLayout child views; they default to WRAP_CONTENT.

GRID VIEW

Android has a GridView control that can display data in the form of a grid. We use the term data here; the contents of the grid can be text, images, and so on. The GridView control displays information in a grid. The usage pattern for the GridView is to define the grid in the XML layout and then bind the data to the grid using an android.widget.ListAdapter. Don't forget to add the uses-permission tag to the AndroidManifest .xml file to make this example work. We've no doubt noticed that the adapter used by the grid is a ListAdapter. Lists are generally one-dimensional, whereas grids are two-dimensional. We can conclude, then, that the grid actually displays list-oriented data. And it turns out that the list is displayed by rows. That is, the list goes across the first row, then across the second row, and so on. As before, we have a list control that works with an adapter to handle the data management, and the generation of the child views. The same techniques we used before should work just fine with GridViews. One exceptionrelates to making selections: there is no way to specify multiple choices in a GridView.

IMAGE VIEW CONTROL

This is used to display an image, where the image can come from a file, a content provider, or a resource such as a drawable. We can even specify just a color, and the ImageView will display that color.

SPINNER CONTROL

The Spinner control is like a drop-down menu. It is typically used to select from a relatively short list of choices. If the choice list is too long for the display, a scrollbar is automatically added for us. Although a spinner is technically a list control, it will appear to us more like a simple TextView control. In other words, only one value will be displayed when the spinner is at rest. The purpose of the spinner is to allow the user to choose from a set of predetermined values: when the user clicks the small arrow, a list is displayed, and the user is expected to pick a new value. Populating this list is done in the same way as the other list controls: with an adapter. Because a spinner is often used like a drop-down menu, it is common to see the adapter get the list choices from a resource file. Notice the new attribute called android:prompt for setting a prompt at the top of the list to choose from. The actual text for our spinner prompt is in our /res/values /strings.xml file. As we should expect, the Spinner class has a method for setting the prompt in code as well.

ICON MENU

Android supports not only text but also images or icons as part of its menu repertoire. We can use icons to represent menu items instead of and in addition to text. Note a few limitations when it comes to using icon menus. 1) we can't use icon menus for expanded menus. This restriction may be lifted in the future, depending on device size and SDK support. Larger devices may allow this functionality, whereas smaller devices may keep the restriction. 2) Icon menu items do not support menu item check marks. 3) If the text in an icon menu item is too long, it's truncated after a certain number of characters, depending on the size of the display. (This last limitation applies to text based menu items also). Creating an icon menu item is straightforward. We create a regular text-based menu item as before, and then we use the setIcon() method on the MenuItem class to set the image. We need to use the image's resource ID, so we must generate it first by placing the image or icon in the /res/drawable directory. For example, if the icon's file name is balloons, then the resource ID is R.drawable.balloons.

SUB MENU

A Menu object can have multiple SubMenu objects. Each SubMenuobject is added to the Menu object through a call to the Menu.addSubMenu() method. We add menu items to a submenu the same way that we add menu items to a menu. This is because SubMenu is also derived from a Menu object. However, we cannot add additional submenus to a submenu

CONTEXT MENU

In Windows applications, for example, we can access a context menu by right-clicking a UI element. Android supports the same idea of context menus through an action called a long click. A long click is a mouse click held down slightly longer than usual on any Android view. On handheld devices such as cell phones, mouse clicks are implemented in a number of ways, epending on the navigation mechanism. If our phone has a wheel to move the cursor, a press of the wheel serves as the mouse click. Or if the device has a touch pad, a tap or a press is equivalent to a mouse click. Or we might have a set of arrow buttons for movement and a selection button in the middle; clicking that button is equivalent to clicking the mouse. Regardless of how a mouse click is implemented on our device, if we hold the mouse click a bit longer, we realize Although a context menu is owned by a view, the method to populate context menus resides in the Activity class. This method is called activity.onCreateContextMenu(), and its role resembles that of the activity.onCreateOptionsMenu() method. This callback method also carries with it (as an argument to the method) the view for which the context menu items are to be populated. The steps to implement a context menu: 1) Register a view for a context menu in an activity's onCreate() method. 2) Populate the context menu using onCreateContextMenu(). We must complete step 1 before this callback method is invoked by Android. 3) Respond to context menu clicks

DYNAMIC MENUS

If we want to create dynamic menus, use the onPrepareOptionsMenu() method that Android provides on an activity class. This method resembles onCreateOptionsMenu() except that it is called every time a menu is invoked. We should use onPrepareOptionsMenu() if we want to disable some menu items or menu groups based on what we are displaying. For 3.0 and above, we have to explicitly call a new provisioned method called invalidateOptions Menu(), which in turn invokes the onPrepareOptions Menu(). We can call this method any time something changes in our application state that would require a change to the menu.

POPUP MENUS

SDK 4.0 enhanced this slightly by adding a couple of utility methods (for example, PopupMenu.inflate) to the PopupMenu class. A pop-up menu can be invoked against any view in response to a UI event. An example of a UI event is abutton click or a click on an image view.

DIFFERENT RESOURCES

Resources play a key role in Android architecture. A resource in Android is a file or a value that is bound to an executable application. These files and values are bound to the executable in such a way that we can change them or provide alternatives without recompiling the application. Examples of resources include strings, colors, bitmaps, and layouts. Instead of hard-coding strings in an application, resources allow us to use their IDs instead. This indirection lets change the text of the string resource without changing the source code.

STRING RESOURCES

Android allows us to define strings in one or more XML resource files. These XML files containing string-resource definitions reside in the /res/values subdirectory. The names of the XML files are arbitrary, although we commonly see the file name as string s.xml. Listing 3–1 shows an example of a string resource file.Listing 3–1. **Example strings.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="hello">hello</string>
<string name="app_name">hello appname</string>
</resources>
```

Even the first line of the file indicating that it is an XML file with a certain encoding is optional. When this file is created or updated, the Eclipse ADT plug-in automatically creates or updates a Java class in our application's root package called R.java with unique IDs for the two string resources specified. Regardless of the number of resource files, there is only one R.java file.

LAYOUT RESOURCES

In Android, the view for a screen is often loaded from an XML file as a resource. This is very similar to an HTML file describing the content and layout of a web page. These XML files are called layout resources. A layout resource is a key resource used in Android UI programming. Consider the code segment in Listing 3–4 for a sample Android activity. Listing 3–4. **Using a Layout File**

```
public class HelloWorldActivity extends Activity
{
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
TextView tv = (TextView)this.findViewById(R.id.text1);
tv.setText("Try this text instead");
```

The line setContentView(R.layout.main) points out that there is a static class called R.layout, and within that class, there is a constant called main (an integer) pointing to a View defined by an XML layout resource file. The name of the XML file is main.xml, which needs to be placed in the resources' layout subdirectory. In other words, this statement expects the programmer to create the file /res/layout/main.xml and place the necessary layout definition in that file.

COLOR RESOURCES

As we can with string resources, we can use reference identifiers to indirectly reference colors. Doing this enables Android to localize colors and apply themes. Once we've defined and identified colors in resource files, we can access them in Java code through their IDs. Whereas string-resource IDs are available under the <ourpackage>. R.string namespace, the color IDs are available under the <ourpackage>. R.color namespace.Android also defines a base set of colors in its own resource files. These IDs, by extension, are accessible through the Android android.R.color Namespace.

DIMENSION RESOURCES

Pixels, inches, and points are all examples of dimensions that can play a part in XML layouts or Java code. We can use these dimension resources to style and localize Android UIs without changing the source code. We can specify the dimensions in any of the following units: px: Pixels in: Inches mm: Millimeters pt: Points dp: Density-independent pixels based on a 160dpi (pixel density per inch) screen (dimensions adjust to screen density) sp: Scale-independent pixels (dimensions that allow for user sizing; helpful for use in fonts).

IMAGE RESOURCES

Android generates resource IDs for image files placed in the /res/drawable subdirectory. The supported image types include .gif, .jpg, and .png. Each image file in this directory generates a unique ID from its base file name. If the image file name is sample_image.jpg, for example, then the resource ID generated is R.drawable.sample_image.

TextView

A TextView displays text to the user and optionally allows them to edit it. It is a complete text editor but the basic class is not allowing to editing. The TextView control knows how to display text but does not allow editing. This might lead us to conclude that the control is essentially a dummy label. We can set the autoLink property to emai or web, and the control will find and highlight any e-mail addresses and URLs. TextView is utilizing the android.text.util.Linkify class. The main attributes are id, hint, maxHeight, maxWidth, password, minHeight, minWidth, text, phoneNumber, textColor, textStyle etc.

EditText:

The EditText control is a subclass of TextView. As suggested by the name, the EditText control allows for text editing. EditText is not as powerful as the text-editing controls that we find on the Internet, but users of Android-based devices probably won't type documents they'll type a couple paragraphs at most. Therefore, the class has limited but appropriate functionality and may even surprise us. For example, one of the most significant properties of an EditText is the inputType. We can set the inputType property to textAutoCorrect have the control correct common misspellings. We can set it to textCapWords to have the control capitalize words. Other options expect only phone numbers or passwords. The old default behavior of the EditText control is to display text on one line and expand as needed. The user to a single line by setting the singleLine property to true. The user will have to continue typing on the same line. With inputType, if we don't specify textMultiLine, the EditText will default to single-line only. So if we want the old default behavior of multiline typing, we need to specify inputType with textMultiLine.

AutoCompleteTextView

The AutoCompleteTextView control is a TextView with auto-complete functionality. In other words, as the user types in the TextView, the control can display suggestions for selection. For example, if the user types en, the control suggests English. If the user types gr, the control recommends Greek, and so.

MultiAutoCompleteTextView

The control offers suggestions only for the entire text in the text view. In other words, if we type a sentence, we don't get suggestions for each word. That's where MultiAutoCompleteTextView comes in. We can use the MultiAutoCompleteTextView to provide suggestions as the user types. For example, the user typed the word English followed by a comma, and then Ge, at which point the control suggested German. If the user were to continue, the control would offer additional suggestions. Using the MultiAutoCompleteTextView is like using the AutoCompleteTextView. The difference is that we have to tell the control where to start suggesting again.

CHECKBOX CONTROL

The CheckBox control is another two-state button that allows the user to toggle its state. The difference is that, for many situations, the users don't view it as a button that invokes immediate action. From Android's point of view, however, it is a button, and we can do anything with a check box that we can do with a button. We manage the state of a check box by calling setChecked() or toggle(). We can obtain the state by calling isChecked().If we need to implement specific logic when a check box is checked or unchecked, we can register for the on-checked event by calling setOnCheckedChangeListener() with an implementation of the OnCheckedChangeListener interface. We'll then have to implement the onCheckedChanged() method, which will be called when the check box is checked or unchecked. The nice part of setting up the OnCheckedChangeListener is that we are passed the new state of the CheckBox button. We could instead use the OnClickListener technique as we used with basic buttons. When the onClick() method is called, we would need to determine the new state of the button by casting it appropriately and then calling isChecked() on it.

RADIO BUTTON CONTROLS

RadioButton controls are an integral part of any UI toolkit. A radio button gives the users several choices and forces them to select a single item. To enforce this single-selection model, radio buttons generally belong to a group, and each group is forced to have only one item selected at a time. To create a group of radio buttons in Android, first create a RadioGroup, and then populate the group with radio buttons. Note that the radio buttons within the radio group are, by default, unchecked to begin with, although we can set one to checked in the XML definition. To set one of the radio buttons to the checked state programmatically, we can obtain a reference to the radio button and call setChecked().We can also use the toggle() method to toggle the state of the radio button. As withthe CheckBox control, we will be notified of on-checked or on-unchecked events if we call the setOnCheckedChangeListener() with an implementation of the OnCheckedChangeListener interface. There is a slight difference here, though. This is a different class than before. This time, it's technically the RadioGroup.OnCheckedChangeListener class, whereas before it was the CompoundButton.OnCheckedChangeListener class. The RadioGroup can also contain views other than the radio button.